PPIG2010

Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group

Edited by:
Joey Lawrance and Rachel Bellamy

Proceedings of the 22nd Annual Psychology of Programming Interest Group.
September 19-22, 2010. Universidad Carlos III de Madrid, Leganés, Spain.

ISBN 978-84-693-3416-4

Joseph Lawrance and Rachel Bellamy, editors.
Maria Paloma Díaz Pérez and Mary Beth Rosson, publishers.

Psychology of Programming Interest Group (PPIG) 2010

Table of Contents i

Editors’ Note

It is our honor to welcome you to the 22nd Annual Workshop of the Psychology of Programming Interest Group. This year's workshop will be co-located with the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), and hosted at Universidad Carlos III de Madrid.

We are excited to welcome Margaret Burnett as the keynote speaker to PPIG. Margaret Burnett is a Professor of Computer Science at Oregon State University, and has investigated how gender differences interact with purportedly gender-neutral end-user programming and end-user software engineering systems. Her work promises interventions that can help both males and females.

We are also pleased to provide PPIG attendees the opportunity to attend Ed Chi’s VL/HCC keynote address on Model-Driven Research in Human-Centric Computing. Ed Chi is the Area Manager and a Principal Scientist at Palo Alto Research Center's Augmented Social Cognition Group.

Many people have helped to make this event possible. We would especially like to thank the reviewers who offered constructive feedback to the authors of all the papers, and Maria Kutar who once again stepped-up to organize the doctoral consortium. We could not have organized this workshop more than 5,000 kilometers away without the help of the VL/HCC chairs Paloma Diaz and Mary Beth Rosson, and the VL/HCC local committee Ignacio Aedo. As always, too many others to mention offered advice and a helping hand along the way, we thank you all.

Thank you, all! Welcome to Madrid!
¡Gracias a todos! ¡Bienvenido a Madrid!

Rachel Bellamy, IBM Research
Joseph Lawrance, Wentworth Institute of Technology

Back cover photograph courtesy of: http://www.flickr.com/photos/beeldenzeggenmeer/405092064/

Gender HCI and Programming Tools

Margaret Burnett

Department of Electrical Engineering
and Computer Science
Oregon State University
Corvallis, Oregon
burnett@eecs.oregonstate.edu

Although there have been recent investigations into how to understand and ameliorate the low representation of females in computing, there has been little research into how software tools fit into the picture. We have been investigating how gender differences interact with purportedly gender-neutral software tools that aim at supporting people doing programming. For example, what if female end-user programmers' problem-solving effectiveness, when using end-user programming environments like Excel, would accelerate if the environment were changed to take gender differences into account? This talk reports the investigations my students and I have conducted into whether and how programming tools affect males' and females' performance differently, and describes the beginnings of work on promising interventions that help both males and females.

Margaret Burnett is a Professor of Computer Science at Oregon State University. Her research focuses on human issues of programming languages and environments, especially when the programming is done by males and females not trained as professional programmers. She has been investigating gender differences in the context of problem-solving software for several years, considering populations ranging from spreadsheet users to professional programmers. She is also a co-founder of the EUSES Consortium, a collaboration among Oregon State University, Carnegie Mellon University, Drexel University, Pennsylvania State University, University of Cambridge, University of Nebraska, University of Washington, and IBM, to help End Users Shape Effective Software (EUSES).

Liveness in Notation Use: From Music to Programming

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

Chris Nash
Computer Laboratory
Cambridge University
Christopher.Nash@cl.cam.ac.uk

Alan F. Blackwell
Computer Laboratory
Cambridge University
Alan.Blackwell@cl.cam.ac.uk

Abstract

In this paper we draw an analogy between musical systems and programming environments, concentrating on user experience associated with feedback and its implications for flow. We present a number of different analytical frames all of which, we suggest, influence the nature of this feedback and with it, the user experience. We introduce a new diagrammatic analysis format, and use it to explore the kinds of feedback loop present in musical systems, what such systems might teach us in the analytical description of programming languages, and vice versa.

Introduction

The value of user feedback when interacting with computer systems is well-known. The need for feedback is taught in standard textbooks (e.g. Preece, Sharp & Rogers 2007), undergraduate courses (e.g. Blackwell 2009), and professional best practice (e.g. Microsoft 2009). In mainstream user interface design, feedback is supported by the principles of direct manipulation (Shneiderman 1983). In the analysis of programming languages, as we discuss later, a variety of kinds of feedback are incorporated in Taminoto's characterisation of liveness (Tanimoto 1990). Furthermore, in recent analyses of domain specific programming for end-users, lack of feedback has been shown to be a key impediment to usability. (Church & Whitten 2009, Church et al. 2009)

However the concept of user feedback is not a simple continuum. By drawing an analogy between musical systems and programming IDEs (Integrated Development Environments), we can ask a number of questions about different aspects of feedback; the source of the feedback, the level of liveness of the feedback, and the trends over time as systems evolve.

We start this analysis with a discussion of feedback in the modern IDE, showing that Tanimoto's theoretical frame for types of liveness (Tanimoto 1990) is still relevant, and use it as the starting point of the analogy to musical systems. We then apply this analogy to liveness in programming and consider the different sources of feedback the programmer and the musician has access to.

Levels of liveness

Intuitively, liveness is an assessment of how 'responsive' a system is. When I perform an action, are my changes immediately apparent? Do I have to go through a number of auxiliary steps in order to understand the consequences of my actions? Liveness is a property both of the program notation, and also of its execution environment. (Tanimoto 1990)

- Level 1 liveness
  (informative; "ancillary")
  describes situations in which a visual representation is used as an aid to software design (Tanimoto was referring to a user document such as a flowchart, not a programming language). This provides a basic level of graphical representation, and can be made continuously visible, although mainly because of the fact that a paper document can be placed beside the screen, rather than on it.

- Level 2 liveness
  (informative, significant; "executable")
  describes situations in which the visual representation specifies a program that can be manually executed, possibly after compilation. This provides a basic kind of physical action mapping, in that modification of the representation will eventually change the behaviour of the program.
- Level 3 liveness
  (informative, significant, responsive; "edit-triggered")
  describes situations in which the representation responds to an edit with immediate feedback, automatically executing or applying the changes. This allows users to make rapid actions, and often (after noting the system response) an opportunity to quickly reverse an incorrect action.
- Level 4 liveness
  (informative, significant, responsive, live; "stream-driven")
  describes situations in which the environment is continually active, showing the results of program execution as changes are made to the program. This provides high visibility of the effect of actions.

Because it spans both notation and execution, the degree of liveness within a programming environment is not a single factor, but can vary significantly across different components. The user experience of feedback in a typical professional IDE, such as Visual Studio or Eclipse, is far from homogenous.

Some aspects have a high degree of live responsiveness which span multiple steps of the traditional programming and compilation cycle. For example, code completion (e.g. Visual Studio's *IntelliSense*), much loved by developers, offers real-time feedback on code, not only from its explicit feedback (suggesting options for code-completion) but from implicit feedback when the programmer notices errors in the code by virtue of the IntelliSense engine failing to correctly suggest possibilities. Whilst this is still very much feedback from the notation (the source code and the IDE) rather than the domain (the executing behaviour of the program) it is rapid, typically appearing in under a second, and brings some of the benefits of Level 3 liveness.

However, a professional programmer spends much of her time doing other things than entering new code. At the simplest level activities include compiling, debugging and deploying. The Level 2-live compile cycle of even a small application on a high-performance workstation can take in the order of 30s, potentially disrupting the programmers flow and requiring distracting context shifts of attention.

The story for debugging is equally variable, though step-through debuggers offer the ability to provide real-time feedback (Level 3 liveness), albeit at a much slower pace than typical execution. However the programmer may have to manually walk through a series of steps in order to put the system in the desired state for debugging. Tool support such as Inform 7's *Skein* (Nelson 2006) allows replaying of a set of pre-recorded steps, but this is still an unusual feature.

In order to look at the way the different levels of liveness affect the different aspects of the process of developing, we shall consider a domain where the importance of feedback has been appreciated for a long time, and where the boundaries of liveness levels are starker: music.

Feedback in music

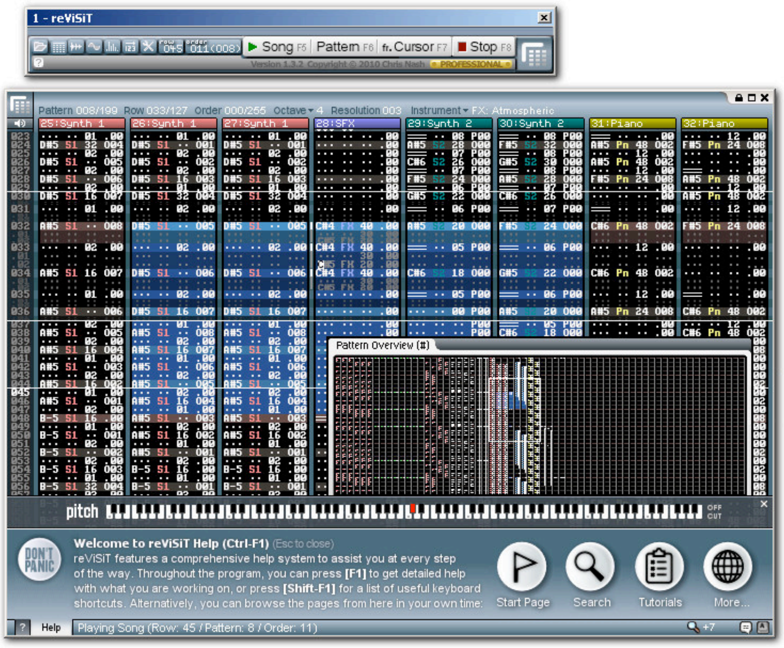
In music, feedback comes in many forms – tactile, haptic, visual and, most importantly, aural. In acoustic performance, such feedback occurs in realtime, but when you move to the digital domain, a latency is necessarily introduced, to enable efficient buffered DSP processing. Nonetheless, the tolerable delay is stricter than typical program feedback (e.g. Nielsen 1993); only a few milliseconds can be distracting (Walker 1999).

Modern music production software, such as the *sequencer* or *digital audio workstation (DAW)*, is centred on the idea of recording such a performance, either in MIDI or audio, and exploits the provisionality of the digital domain to support experimentation and improvisation. However, after capturing the initial realtime performance, editing is mediated through abstract visual notations, often entailing cumbersome WIMP interfaces (van Dam 1997) and overly-literalistic visual metaphors (Duignan 2004). Increasingly complex features and interactions slow down the feedback cycle, and interaction is driven by abstract feedback from the visual notation, rather than concrete feedback from the domain itself, such as sound.

The DAW can be seen as a reversal of the classical composition process, in which the composer creates the notation before the music is performed, and the musical structure may be sketched in abstract form before a more concrete instantiation is required (Sloboda 1985). Instead, sequencers are more suited to the scenario where the initial performance is more representative of the actual end product; the musical idea must be largely formed before the user interacts with the program. As such, it is the tight interaction and feedback cycle with the physical instrument (either acoustic or MIDI) that provides immediate musical feedback in the sequencer model, rather than with the notation itself.

Fluid and fluent user experiences have been identified as critical to the creative user experience (e.g. Norman 1993, Shneiderman et al 2005). Relatedly, the notion of 'flow' (Csikszenmihalyi 1990) has been closely linked with creativity, and advocates "direct and immediate feedback". Similarly, Leman (2007) argues that more "direct involvement" in the music can be afforded by fast feedback loops, which he calls *action-reaction cycles*.

Our own research has been investigating an alternative style of composition tool, which prioritises musical feedback over richer, graphical affordances. The *soundtracker* (e.g. Nash 2004) offers a spreadsheet-like interface, using a grid of text to describe musical phrases, where each cell has a fixed number of spaces to specify pitch, instrument, volume (or panning) and one of a variety of musical ornaments (or *effects*), for example: $\boxed{\textbf{C\#5 01 64 D01}}$ starts playing a note [C#] in octave [5]; instrument [01]; maximum volume [64]; with a slow [01] *diminuendo* [D]. Despite the unorthadox appearance of tracker notation (see Figure 1), the music produced by its users is quite conventional – from dance tracks and pop songs to film scores and orchestral symphonies. Using the computer keyboard, the musical text can be edited very efficiently, prompting some to liken the process to a form of "musical touch-typing" (MacDonald 2007).

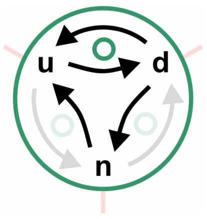


Welcome to reViSiT Help (Ctrl-F1) (Esc to close)
reViSiT features a comprehensive help system to assist you at every step of the way. Throughout the program, you can press **[F1]** to get detailed help with what you are working on, or press **[Shift-F1]** for a list of useful keyboard shortcuts. Alternatively, you can browse the pages from here in your own time:

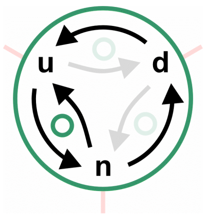
Playing Song (Row: 45 / Pattern: 8 / Order: 11)

Figure 1 – The reViSiT soundtracker, a text-based music composition tool

Unlike the performance-based approach of sequencers, trackers encourage interaction with the notation, but crucially keep sound feedback close-at-hand – notes, phrases, parts or the whole song can be auditioned instantly, with a single keypress. In Figure 2, we illustrate this difference between the approaches, as a function of the feedback loops they produce.



(A) *a performance-driven system, (e.g. a MIDI sequencer)*



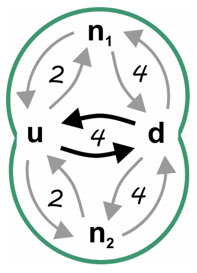
(B) *a manipulation-driven system, (e.g. a soundtracker)*

Figure 2 – Two computer music experiences, modelled as intrinsic feedback loops combining the user (u) and musical domain (d) with notation (n), connected by arcs representing common creative processes (performance, audition, data manipulation, visualisation, transcription and realisation).

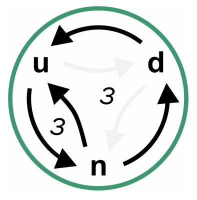
Performance-based systems as represented in Figure 2 (A) support flow by allowing episodes of tight feedback between the user (u) and the musical domain (d), and capturing the musical output of their instrument. Between these episodes, users must interact with some transcription of the performance, which may only represent a crude adumbration of the original performance (e.g. Figure 2 (A), in the case of DAWs). By contrast, manipulation-driven systems like trackers, as represented in Figure 2 (B), do not rely on such transcription, restricting the musical possibilities to those realisable in the notation, and driving interaction from sonic feedback from the encapsulated music, keeping the visual feedback between the user (u) and the notation (n) as direct, minimal, and tight as possible.

Adapting the earlier classification of liveness to our musical examples, we note that Level 4 liveness is that supported by "live" musical performance - where the effects of actions (on the instrument) are continuously and immediately *audible* - and, as such, we can clearly see its utility in sequencers and other performance-based music software. At the other end of the spectrum, Level 1 liveness can be seen in the offline notations used by composers, such as the sketching of ideas and musical processes on paper, and the ancillary visual representation presented by a sequencer's "Arrange Window".

The majority of other computer music scenarios are centred on editing a visual specification of what will happen in the music, as in both a sequencer/DAW's GUI and trackers, and thus such programs lie somewhere on a continuum between Level 2 and Level 3 liveness, based on the immediacy and quality of feedback provided. Sequencers are unable to sustain Level 4 liveness after the point at which the performance is captured, instead providing sub-devices (e.g. Arrange Window, Score Editor, Piano Roll, etc.) each allowing the visualisation and editing of specific and distinct aspects of the recorded data, where interaction is driven by visual feedback, and less frequently auditioned by spooling to the appropriate point and initiating playback. Although the music is then realised in realtime, the feedback is far from a continuous interpretation, and most often offers only Level 2 liveness, requiring the user to spool to an appropriate point and press play. Conversely, the rapid actions and prominent role of frequent sonic feedback, leads us to consider trackers more in terms of Level 3 liveness, where small edits are made and almost immediately auditioned - the execution is still triggered by the user, but the single keypress becomes *cognitively* automatic, following the edit.



(A) *a digital audio workstation (DAW)*



(B) *a soundtracker*

Figure 3 – Feedback loops in two music program types, annotated with levels of liveness.

In Figure 3, we annotate the feedback loops related to two scenarios in computer music interactions with the level of perceived liveness. DAWs (3A) do not have a central primary notation, instead offering numerous different tools to present different aspects a musical recording, splitting interaction between different editing features, views and windows, each using pointer-based direct manipulation and often triggering slow and a complex non-realtime ("offline") processes. By contrast, the tracker environment (3B) prioritises a single musical notation, and supports rapid sound feedback and a generally higher level of liveness, without supporting the Level 4 liveness of a direct performance.

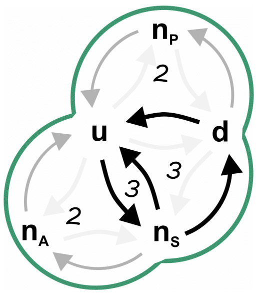
Towards Liveness in Programming

Level of Liveness	Programming	Music
1	flow chart, UML diagram	composer shorthand
2	code editor, compiler	score, sequencer/DAW GUI
3	code completion, syntax highlighting, realtime compilation, edit and continue	soundtracker, live coding
4	macro recording, Scratch, Data Canvas[1]	sequencer/DAW recording

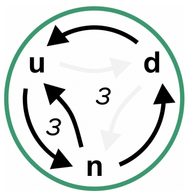
Table 1 - Examples of liveness levels in programming and music

Thus, in music, we can see that the tools strive to offer higher levels of liveness, supporting more fluid interactions, by tightening the feedback cycles between the notation and/or the domain. We can see the same trend in IDEs (as illustrated in Table 1). The introduction of *Edit-and-Continue* in Visual Studio, where a program can be paused, edited and then directly resumed, decreases the need for a full compile cycle after minor changes to the code, supporting Level 3 liveness. However, in other places we are moving in the opposite direction. The migration from compilers to in-line syntax checkers was an increase in liveness, but is now being supplemented by static analysis tools such as Coverity[2]. Currently, such tools tend to execute in a separate pass, being more a Level 2 tool.

Figure 4 shows the different cycles of feedback that occur in a typical modern IDE (4A) and a code editor (4B). We can see that the core interaction cycle between the users, the source code notation and the domain, is the same, with roughly the same degree of liveness, Level 3. However, the ancillary support tools, the profiler and the static analyser currently have lower liveness properties, Level 2.



(A) an integrated development environment (IDE) (e.g. Visual Studio)



(B) standalone code editor (e.g. EMACS)

Figure 4 – Two programming experiences, modelled as feedback loops combining the user (u), execution domain (d), source notation (n/n_s), profiler notation (n_p) and static analyser notation (n_a).

[1] introduced later
[2] www.coverity.com

There is a general theme here that when tools first get introduced they are at a relatively low level of liveness (e.g. Code Profilers are frequently found at Level 2 at the time of writing), but as technology progresses they migrate towards increasing liveness. This tendency lends weight to our suggestion that this is a crucial, but under-recognised, aspect of the user experience of programming, in that it seems to emerge in response to actual usage, rather than being designed-in at the outset.

Sources of Feedback: Notation, Domain

Previously, we have talked about the different sources of feedback that are available to the musician, the most obvious being the domain of sound itself. There is also feedback to be derived from the notation, in whatever form. This is similar to the experience of a programmer who gains feedback about their current task, say a programming or debugging activity, both from the executing behaviour of the program and from interactions with the notation, the source code and associated environment.

In most professional programming environments, these two worlds, the world of notation (source code) and of 'performance' (execution) are completely distinct. A common strategy for attempting to tighten the feedback loop is to batch-simulate the behaviour of small sections of the code in close to realtime, independent of the rest of the program. The Windows Presentation Foundation preview in Visual Studio does this, as does the Sprite view in Scratch (Malan & Leitner 2007).

Feedback in Scratch

The Scratch programming environment from MIT (ibid.), is one of the latest in a series of attempts to build educational programming environments for children, in which the main motivating element of the environment is that children are able to create their own videogames. There have been many other examples in the past of programming environments oriented toward videogame children, recently including Alice (Pausch et al 1995), Robertson and Good's AdventureAuthor (2005), and Microsoft's Kodu[3]. Scratch was explicitly motivated by a metaphor of media construction as musical improvisation, referring to the scratching techniques of turntable artists when they create new works from existing media. Many of these systems motivate children by providing rich libraries of media, artwork, and language primitives that can be rapidly composed into a satisfying result.

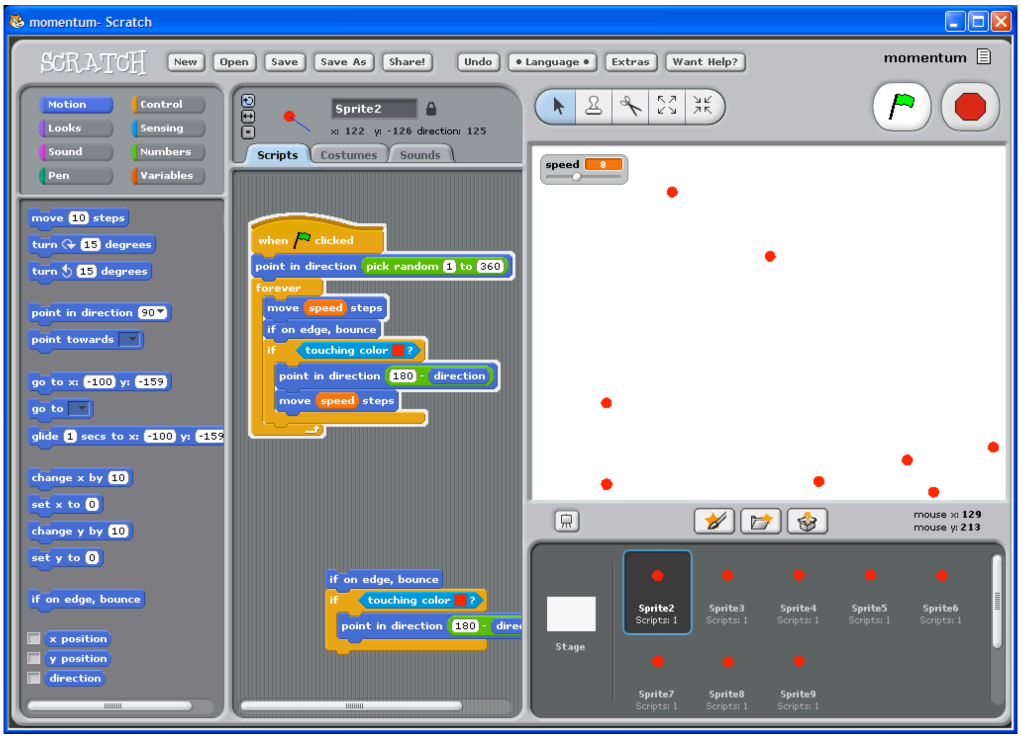
The feedback cycles in tools of this kind have two main effects. One is to maintain the level of motivation, by rewarding the child either with a functional product, or at least with a believable promise that a functional product is within reach. The other is to quickly correct faulty mental models of the system behaviour, in order that misconceptions about programming do not become entrenched. Both of these factors contribute to developing expertise. We believe that the same factors are likely to apply in adult development of expertise, although in children both are more dramatic (children generally have less patience, and are also more likely to acquire fundamental misconceptions).

A screenshot of the Scratch development environment can be seen in Figure 5. It shares the properties of many simple programming IDEs, and is very similar in overall structure to other recent instructional programming environments, such as Alice. It includes a) a live preview of the game display 'stage'; b) a list of those objects ('sprites') that appear on the stage; c) a canvas on which the behaviour of the sprite 'scripts' can be specified by assembling language primitives; and d) a navigation interface for finding and selecting primitives.

Most significant to our argument is the effort that has been devoted in this kind of environment to offering an experience in which the program is constantly 'running' (or using Tanimoto's term, 'live'). Any aspect of object behaviour can be evaluated at any time – whether a single language primitive or a whole script – to preview the effect it will have on the stage. An executing loop can be edited while it executes, and the next iteration of the loop will follow the new behaviour. All values can be inspected or modified at any time. No doubt it has taken substantial engineering effort to allow this much user freedom while maintaining a consistent execution state[4].

[3] http://research.microsoft.com/en-us/projects/kodu/

[4] From personal experience, and based on reports from its increasingly wide use in UK schools, Scratch is very robust.



```
move 10 steps
turn 15 degrees
turn 15 degrees
point in direction 90
point towards
go to x: -100 y: -159
go to
glide 1 secs to x: -100 y: -159
change x by 10
set x to 0
change y by 10
set y to 0
if on edge, bounce
x position
y position
direction
```

```
when clicked
point in direction pick random 1 to 360
forever
    move speed steps
    if on edge, bounce
    if touching color ?
        point in direction 180 - direction
        move speed steps
```

```
if on edge, bounce
if touching color ?
    point in direction 180 - dire
```

Figure 5 - The Scratch development environment

We find it interesting to reflect on the design principles that are apparent in this successful product. There are two kinds of direct visual feedback – sprites on the stage can be dragged with the mouse, and syntax elements can be moved around within the script window. However, both kinds of action also have effects on system state as a result of maintaining the consistency between the internal state of the execution engine and the direct manipulation interface. Dragging a sprite on the stage updates the current state of that sprite (potentially overriding state that resulted from script execution), but with the program continuing to run, so that it is possible to explore the effect of program execution from alternative screen states. Even if program execution is halted, the current position of a sprite is used as the default location for new operations added to the script – for example, allowing a character movement to be specified simply by dragging the sprite to the desired location, then inserting a move-to command in which the current location will have become the desired location.

These are examples of the close coupling between behaviour and notation that allow Scratch to provide feedback at multiple levels, and to maintain motivation during the acquisition of expertise.

Another challenge for abstraction

The increasingly close coupling of behaviour and notation that makes Scratch successful is, as we suggested earlier, part of a general progression of tools that work at the levels of the notation and the domain towards increasing liveness. An example is the range of refactoring tools, which started out as separate monolithic entities of the kind that many static analysis tools often are today, but have been progressively merged into mainstream interaction.

However, herein lays an interesting challenge for the psychology of programming. In many music production environments, a cumbersome visual metaphor fails to provide the kind of liveness experienced in trackers, where notation and domain are separate, but offer rapid feedback cycles. The abstraction manager/sub-device in an IDE constitutes an analogous obstacle. A common design manoeuvre in Cognitive Dimensions is to respond to viscosity by introducing an abstraction and an abstraction manager. However doing so typically encumbers the user interface and provides a slow rate of interaction, albeit more powerful interaction. Whilst initially from an Attention Investment (Blackwell 2002) point of view this may be a rational trade-off, we are suggesting that there is an experiential difference, notably in the ease of achieving flow, between a small number of abstraction driven interactions and a large number of micro-interactions, as supported by (e.g. Resnick et al 2005). An interesting objective for programming usability is to achieve the same flow interaction, just at the higher level of abstraction – or "high-level hacking", as Thomas Green calls it (Green, personal communications).

One step along this path is to use technological innovations to minimise the cost of every operation. This is the approach the Data Canvas[5] takes. This project uses extensive pre-computation and cluster computing to achieve operations over statistical amounts of data at interactive speed, thus enabling the user to view and manipulate a statistical profile of their data in real-time. This, we predict, will have two effects:

1. It will increase the likelihood of the programmer achieving flow, by preventing disruptive delays and mode switching (e.g. Data Canvas, like Scratch, contains no notion of a separate mode for debugging)
2. It will allow the programmer to interact with emergent behaviours of their data through a large number of very rapid micro-operations. This will help decrease the premature commitment risk associated with abstraction over unknown data. (Church & Whitten 2009)

Sources of Feedback: Other Worlds

We have talked so far about feedback from the notation and the domain. However there are other sources of feedback that are important to both the programmer and the musician. First we have hinted above that there are analysis tools that do not form a strict layering of feedback but rather a graph. These feedback arcs can be extended beyond the technical, into the social (the feedback to the developer from the user in participatory design).

So the level of liveness of a technical ecosystem can be broken down into considering the properties of each sub-device and operation. Previous attempts to extend the analytic purchase of Cognitive Dimensions have struggled with the need to describe different notational 'levels' - structured representations that contain the same information, and can potentially be translated from one to another, but have different notational properties. The music analogy emphasises the ways in which the user's experience constitutes a web of interconnections between artefactual, cognitive, social and cultural structures. All of these can be represented digitally, and all offer different kinds of feedback between the computer and a user. We suggest that making the same kind of description for programming systems as for music systems may be a productive avenue for future work.

Conclusion

In this paper we have considered the liveness of a number of the interactive elements of musical systems and IDEs. We have observed structural correspondences between the systems and drawn analogies between the ways in which they influence their respective experiences of use. We discussed the trend towards increasing liveness in such systems, considering Scratch in detail. We introduced a diagrammatic model of the different places in which tightly coupled feedback might occur, and the way that this can be used to explain different kinds of usability relationship between notations and experienced domains. We concluded with a suggestion as to how this analysis may fix a difficulty with the Cognitive Dimensions' notion of levels. We believe that considering different kinds of feedback loop within musical systems provides new and interesting possibilities for the analytical description of the experience of programming languages.

[5] www.riversofdata.com

References

Blackwell, A.F. (2001). Pictorial representation and metaphor in visual language design. J. Visual Lang. Comput. 12, 3, 223--252.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10.

Blackwell, A.F. (2006). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4), 490-530.

Blackwell (2009) Human-Computer Interaction (undergraduate course notes) http://www.cl.cam.ac.uk/teaching/0910/HCI/HCI2009.pdf

Carroll, J. M, Mazur, S. A. (1986) LisaLearning, Computer, v.19 n.11, p.35-49, Nov.

Church, L. and Whitten, A. (2009). Generative usability: security and user centered design beyond the appliance. In Proceedings of the 2009 Workshop on New Security Paradigms Workshop (Oxford, United Kingdom, September 08 - 11, 2009). NSPW '09. ACM, New York, NY, 51-58. DOI= http://doi.acm.org/10.1145/1719030.1719038

Church, L., Anderson, J., Bonneau, J., and Stajano, F. (2009). Privacy stories: confidence in privacy behaviors through end user programming. In Proceedings of the 5th Symposium on Usable Privacy and Security (Mountain View, California, July 15 - 17, 2009). SOUPS '09. ACM, New York, NY, 1-1. DOI= http://doi.acm.org/10.1145/1572532.1572559

Csikszentmihalyi, M. 1990. Flow: The Psychology of Optimal Experience. New York: Harper Perennial.

van Dam, A. 1997. “Post-WIMP User-Interfaces”, in Communications of the ACM, 40(2):63-67. Association of Computing Machinery.

Leman, M. 2008. Embodied Music Cognition and Mediation Techology. Cambridge, MA: MIT Press.

MacDonald, R. 2007. “Trackers!”, in Computer Music, 113:27-35. Bath, UK: Future Publishing, ltd.

Malan, D. J. and Leitner, H. H. 2007. Scratch for budding computer scientists. SIGCSE Bull. 39, 1 (Mar. 2007), 223-227. DOI= http://doi.acm.org/10.1145/1227504.1227388

Microsoft, 2009. User Experience and Interaction Guidelines for Windows 7 and Windows Vista. Available from: http://msdn.microsoft.com/en-us/library/aa511258.aspx

Nash, C. 2004. VSTrack: Tracking Software for VST Hosts. MPhil Thesis. Trinity College, Available from: http://vstrack.nashnet.co.uk.

Nelson, G. 2006. Inform 7. Available from: http://inform7.com/

Nelson, T.H. 1990. The right way to think about software design. In The Art of Human-Computer Interface Design. B. Laurel, ed. Addison Wesley, Reading, MA. 235-243.

Nielsen, J. 1993. Usability Engineering. Cambridge, MA: AP Professional.

Norman, D.A. 1993. Things That Make Us Smart. New York: Basic Books.

Pausch, R., Burnette, T. Capeheart, A. Conway, M. Cosgrove, D. DeLine R., Durbin, J. Gossweiler, R., Jeff S. K. Alice: Rapid Prototyping System for Virtual Reality White, IEEE Computer Graphics and Applications, May 1995

Preece, J., Sharp, H. Rogers Y. (2007) Interaction Design: Beyond human-computer interaction (2nd Edition).

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., Selker, T., and Eisenberg, M. 2005. NSF Workshop Report on Creativity Support Tools, Available: Workshop on Creativity Support Tools, http://www.cs.umd.edu/hcil/CST/ [Accessed: 13 Apr. 2010].

Robertson, J. and Good, J. (2005). Adventure Author: An Authoring Tool for 3D Virtual Reality Story Construction. In theProceedings of the AIED-05 Workshop on Narrative Learning Environments, pp. 63-69.

Shneiderman, B. 1983. "Direct Manipulation: A Step Beyond Programming Languages", in Computer, August 1983:57-69, Washington, DC: IEEE Computer Society.

Sloboda, J. 1985. The Musical Mind. Oxford, UK: Oxford Science Publications.

Tanimoto S.L. 1990. *VIVA*: A Visual Language for Image Processing. Journal of Visual Languages and Computing 1(2), 127-139.

Tanimoto S.L. 2003 Programming in a data factory. Proc. IEEE Symposium on Human Centric Computing Languages and Environments (*HCC*'03), pp.100-107.

Walker, M. 1999. "Mind the Gap: Dealing with Computer Audio Latency", in Sound On Sound, April 1999. Cambridge, UK: SOS Publications Group.

Usability requirements for interaction-oriented development tools

Catherine Letondal, Stéphane Chatty, W. Greg Phillips[1], Fabien André, and Stéphane Conversy

Université de Toulouse - ENAC
7 avenue Edouard Belin, 31400 Toulouse, France
firstname.name@enac.fr

[1]Royal Military College of Canada
Kingston, Ontario, Canada K7K 7B4
greg.phillips@rmc.ca

Abstract. Building interactive software is a notoriously complex task, for which many programming tools have been proposed over the years. Although the research community has sporadically identified usability requirements for such tools, tool proponents rarely document their design processes and there is no established reference for comparing tools with requirements. Furthermore, the design of most tools is strongly influenced by the design of their underlying general purpose programming languages. These in turn were designed from their own set of little-documented requirements, which adds to the confusion. In this paper, we provide a review and classification of the requirements and properties expected of interactive development tools. We review how designers of APIs and toolkits for interaction-oriented systems set the usability requirements for the programming interface of their systems. We relate our analysis to other studies in related domains such as end-user programming, natural programming, and teaching.

1 Introduction

Throughout the last several decades, programming interactive software has consistently been documented as complex and costly [59]. Dozens of tools, languages, architecture patterns and formal models have been proposed to address aspects of this complexity. However, these have seldom had a significant or lasting impact on programming practices. Most interactive software is still written in languages whose evolution was driven by other forces, and commercial user interface programming frameworks make little use of principles deriving from research – for example, few frameworks include constraints, data-flow, or standardised architecture patterns. In an era when most computers are used for interaction, it is striking that programming languages are still based on requirements derived from computation and that user interface programming still comes as an afterthought in language design.

Several reasons can be suggested for this state of affairs. First, user interfaces are a moving target: for example, principles proposed for programming command languages became partly obsolete with the advent of direct manipulation [60]. It is also possible that researchers have proposed solutions that differ too radically from industry practices. For instance, interpreted languages are thought to facilitate iterative design, but their use conflicts with intellectual property protection. However, it may be simply that *the proposed designs have failed to properly capture and address the domain's true requirements.*

As far back as the early 1970s, Weinberg suggested that we analyse programming tools in terms of their usability, proposed a simple framework for analysing the corresponding requirements, and observed that language designers rarely document the requirements they address [95]. Since then little progress has been made on the articulation of explicit usability requirements by designers, neither for mainstream programming languages nor for interactive software tools. Therefore, when addressing particular requirements, researchers may have produced solutions that ignore or even conflict with other important requirements.

We wish to explore the problem of interactive software tools from this angle. More specifically, we wish to explore the relationship between the usability requirements of programming tools for two classes of software: interaction-oriented software and computation-oriented software. Do they differ, complement, or conflict? Is one a subclass of the other? How can one combine requirements, and thus solutions? To carry out this investigation, we need a framework for comparing requirements. Development of such a framework is the principal goal of this article.

To create the framework, we begin with a top-down approach in which we analyse the nature of interaction-oriented software and its development, in order to arrive at an initial understanding of the relationship between interactive software and programming languages. We use this understanding to propose a requirement analysis framework that, hopefully, will help us understand where requirements meet or conflict. We then survey more than 50 research works on tools for interactive software, using a bottom-up approach to refine the framework and classify these works according to the usability requirements they appear to address. In the future we plan to analyse and classify programming languages in the same framework, and to use convergences and conflicts to inform the design of future programming languages and tools for interactive systems.

2 The nature of interaction-oriented programming

2.1 A subset of general-purpose programming?

Is programming interaction-oriented software a subset of general-purpose programming? Answering this question is important: if the answer is "yes", then the requirements of interactive software can be met through general-purpose programming language design. However, if the requirements of interactive software in some sense exceed those of general-purpose software, either as an intersecting set or a superset, then a different approach is required. (Spoiler: Wegner has argued convincingly that interaction is larger than computation [93]!)

The designers of programming languages more or less explicitly take this stance: each language is designed from a list of concepts that are intended to match all core requirements of programmers, and software libraries based on the language are supposed to address all specialisations of the programming activity. This has been a tremendously successful approach, even though the existence of a superset of all programming activities is only supported by the formal argument of Turing completeness.

With the notable exception of Smalltalk [48], most work on interactive systems has relied on the implicit hypothesis that there is such a thing as a general programming activity, supported by programming languages, and that programming interactive systems is a specialisation of it.

However, given the difficulties observed in designing toolkits and frameworks for interactive systems, this implicit hypothesis needs to be reassessed.

2.2 Commonalities

All programming activities, including that of interactive software, share a number of features. They are intellectual activities aimed at manipulating abstract objects, and as such are similar to mathematics, physics and similar domains. This induces requirements such as supporting the limits of human cognition – for instance in terms of working memory and visual information processing. More specifically, programming activities are creative activities aimed at building complex, dynamic, structured objects: sequences of nested actions, conditions and reactions. This induces specific requirements, some related to the links between our cognitive mechanisms and human languages, some related to our perception-action loop. And, of course, most programming activities sooner or later become collective activities with issues such as reuse and traceability; this invokes complex processes, and the corresponding requirements gave birth to software engineering. It seems evident that many software engineering issues are shared by interactive software.

These common features explain why user interface programmers find it legitimate to use generic programming language and tools, even though their experience degrades in some situations. The existence of common features also explains why tool designers, aiming at economy of design and at reaching the largest possible user base, have generally designed their tools on top of existing languages. These commonalities even explain why some graphical designers find it reasonable go beyond their traditional tasks and write interactive programs or components [61].

2.3 Discrepancies

So, in some ways, interactive software looks similar to other software. But in our experience with interactive software toolkits we often find ourselves asking "why?" Why do toolkits for interaction offer not only specialised objects such as graphics or interactors but also unique component structuring mechanisms [58] and distinct control flow mechanisms like events or data-flow instead of function calls [18]? In these ways they act on languages more as "corrective patches" than as specialisations, and in some cases we suspect the inconsistencies arising from these patches are actively harmful.

The discrepancies we see between interactive and non-interactive software are in the structure of code itself and in the nature of development processes [18]:

Contravariance in reuse and control. The control flow in interactive systems often goes from the outside to the main program. The code that transfers control (input drivers, interactors) predates the code that receives control (the application). This is the opposite of the situation that function calls were made for, thus requiring events or data-flow.

Locality of state. In interactive systems the complexity is in behaviours, that is in the change of state of objects, and not in computations. Solutions that focus on making computations as local as possible in the code tend to be counter-productive, hence the use of state machines.

Concurrency. More and more, interactive systems involve concurrent processes such as animation. Even when there is no such concurrency, applying software engineering techniques to interactive systems and splitting their code into components turns programmers into assemblers of concurrent processes: interactive components run concurrently. User interface frameworks provide this concurrency in a more or less disguised way.

Different reuse patterns. Very diverse stakeholders are involved in producing interactive applications, from programmers to graphic designers and even to end users. Many reuse scenarios for interaction are not well-supported by encapsulation mechanisms proposed by programming languages. For instance, an end user may want to modify the size of a font in a button; however, this might be considered an implementation detail to be hidden from the programmer.

3 Requirements for interaction-oriented development

The previous section provided a top-down review of the nature of interaction-oriented programming along with the large-grain commonalities and discrepancies between it and computation-oriented programming. Our ultimate intent is to achieve a fine-grained understanding of the same material. The approach we have chosen is to analyse of the requirements for each more finely, in order to understand where computation-oriented programming languages and interaction-oriented tools address similar or compatible requirements and where they address incompatible ones. For this, we enrich Weinberg's analysis framework [95] in two ways. First, we base our analysis on the following sets of activities:

- intellectual activities, where programming resides with mathematics and others;
- various sets of concept manipulation activities, because to build objects you first need to understand the nature of your building blocks and of the objects you want to build;
- construction activities, which include manipulation and evaluation; and
- collaboration activities and more generally production processes.

Second, in the following sections we consider more than fifty tools or research works and use them to refine and populate the resulting framework.

In the future, we plan to add programming language research to this analysis, in the hope that this will bring insight as to how solutions from traditional programming language design can be combined with techniques from interactive software to build better tools and languages for both.

3.1 A human intellectual activity

As an intellectual activity, building interactive systems is similar to other activities where conceptualizing, abstracting, reasoning, inducing and understanding are involved. In this sense, mathematics, logic, physics and engineering are disciplines that share common aspects with programming. Because human intellectual capacity is inherently limited, intellectual usability normally requires minimising complexity. In the case of interactive systems, targets for minimization include information complexity, access complexity and unpredictability.

Minimising information complexity. Some HCI toolkit designers advocate reducing the diversity of manipulated objects (graphical objects, input devices, behaviours, etc.) to the fewest general "atoms" that can be used to describe a whole system.

Few constructs. Occam's razor – entities should not be multiplied beyond necessity – is often cited as general rule for design. Smalltalk, which is arguably an interaction-oriented programming language, has only a few constructs such as classes, instances, and messages (Kay claims inspiration from mathematics and biology [48]). In interaction toolkits, Ubit builds a "molecular architecture" [52] from small-size components called brickgets enable to definition of any kind of user interface element.

Homogeneity. Homogeneity, sometimes referred to as "consistency" or "uniformity," is another source of simplicity. According to Weinberg [95], non-homogeneous environments discourage exploration.

Homogeneity is found in at least two dimensions, which we call horizontal and vertical. Horizontal homogeneity occurs where the same construct applies across a range of situations. For example in Sassafras the same construct is used for input/output, inter-process communication, and function calls [42]. In Flapjax, user input events and network events are described by a single abstraction, the event stream [57].

In vertical homogeneity (also known as "fractality" or "layering") the same structure recursively applies at different levels of composition. For example, Kay describes objects in Smalltalk as recursively incorporating the structure of the entire computer [48]. Brickgets in Ubit can be composed to form more complex brickgets in a hierarchical scene-graph [52]. In DIWA, the recursive decomposition of "user interface objects" provides a frame for locality and isolation [85].

Minimising access complexity. Because software is multi-dimensional, the representation of certain concerns can sometimes be spread across the system description, making them difficult to understand. This can be addressed by using structures that gather and bundle together related representations (increasing locality) or by removing extraneous representations (increasing legibility and conciseness).

Increasing locality. Locality means that the user can find elements of interest in a single place and the need for locality explains certain design choices in HCI toolkits. For instance, according to Myers

> OOP [object-oriented programming] is especially natural for user interface programming since the components of user interfaces (buttons, sliders, etc) are manifested as visible objects with their own state (which corresponds to instance variables) and their own operations (which correspond to methods). [60]

This observation forms part of the original design rationale for Smalltalk [48]. It also helps explain why object-oriented approaches seem more popular for interaction toolkits than functional approaches, in which single behaviours are typically spread across many functions. With non-local code, the user must replace visualization with interaction and memory: switching between files, searching for the related code, and mentally assembling multiple chunk of codes in order to understand it.

Other programming approaches popular in interaction toolkits also provide support for locality. Finite State Machines (e.g., SwingStates [2]) allow control aspects of interactive systems to be localized, contrasting with the "spaghetti-code" [62] inherent in the function-oriented programming paradigm [18]. Reactive programming (Esterel [20]), process-based user interfaces languages (Pike's window system [79]), data-flow (Ituikit [16] and Icon [31]), and functional reactive programming (Arrowized FRP [69]) not only avoid the need to cope with a main loop, but also enable the user to better visualize and understand the sequence of transforms from input to display.

Other examples of strong locality support include MDPC [21] in which picking (identification of the graphical object pointed to in the interface) is insulated in a single place, and Boxer [28], where each interactive feature is isolated in its own "box".

Increasing legibility and conciseness. In our context, legibility refers to the degree to which a developer is able to read and understand code in a reasonable time. Lecolinet suggests that

> GUI source code tends to be verbose and hard to read. Informative data is often hidden in a large amount of "syntactic sugar" that conveys little information but is necessary for proper compilation. This lack of conciseness tends to make programs harder to understand and to maintain. [52]

As used by Lecolinet, "conciseness" is a property that refers to the length and number of constructs needed to express the semantics implemented by a chunk of code. It is (inversely) related to the property of diffuseness in the Cognitive Dimensions Framework [37]. Cordy argues that conciseness is an important feature for interaction languages based on his experience designing Turing [22] and Kay makes a similar case in the context of Smalltalk [48].

The legibility of semantically-necessary code can be hindered by the presence of other code required for syntactic compliance. Removing this code makes the result more concise and improves legibility. This is the approach taken in Ubit, whose syntax resembles a formal specification more than a classic programming language [52]. Similarly, the Event Response Language in Sassafras was designed specifically to be more compact than an equivalent recursive transition network-based language [42].

Minimising unpredictability. Unpredictability sometimes arises when automatic algorithms are used to implement system decisions, and where either the rules by which these algorithms operate are complex or opaque, or the number of interdependent entities managed by the algorithms become large.

Many authors, including Winograd, argue that programming languages and toolkits should be primarily declarative [97]. In interaction-oriented systems an argument in favour of such algorithms is that they allow for concise declaration and automatic management of dependencies, which reduces the information and access complexity [52]. Toolkits that provide constructs for defining behaviours, graphical objects, or transformations without any control construct are sometimes described as declarative, at last in part. Many interaction-oriented systems are more or less declarative in this sense, including Garnet [58], Amulet [65], PAC-Amodeus [68], Ubit [52], and MDPC [21].

Myers notes that constraint-based systems, various flavours of model-based systems, and even simple layout algorithms are examples where the declarative aspect of the program should relieve the programmer from implementation details. However, in practice programmers have difficulty understanding declarative mechanisms well enough to to align their needs to the features that are provided [60]. Kay makes a similar observation regarding difficulties observed in skilled programmers attempting to declaratively specify an algorithm for sorting numbers into odd and even sets [48].

3.2 A concept manipulation activity

Like many activities, programming involves the use of concepts for understanding situations and possible actions as well as for defining desired results. A classical usability requirement is that tools present a consistent view of these concepts. Each domain has its own concepts, and this is one place where it appears that computing-oriented and interaction-oriented programming show interesting differences.

Graphics. Graphics has always been a major area within user interface programming, and many requirements for tools stem from the graphical nature of most user interfaces.

Historically, managing graphics has been focused on hardware and rendering processes. This started for instance with tracers: picking a pen, moving it, putting it on paper, and so on. These concepts were ubiquitous in early standards such as GKS [10]. The importance of the algorithmic approach was enforced by the direct rendering mode used in early raster displays, then by the picking algorithms used to determine which graphical objects are designated. Even today, graphics algorithms play an important role in interaction-oriented programming because developers are still required to deal with hardware limitations and to optimize their code.

A different point of view holds that managing graphics is mainly the manipulation of graphical entities: shapes, layers, visual attributes, etc. The approach was probably first taken in graphics editors and 3D programming tools with the introduction of retained rendering; later, the same concept came to user interface tools. This is an appealing perspective for interaction-based systems since it accords well with the fact that graphical designers, and not just programmers, are involved in the production of user interfaces. Tool designers have then come up with various solutions for structuring and manipulating graphical scenes, which are intended to simplify the development of structured graphics. These range from tree data structures (e.g., Piccolo [7]), to directed acyclic graphs (SVG [9, 17]), to tag-based structures (Tk [2, 74]) Other toolkits are specialized for a particular type of interactive software such as Prefuse [41], or the InfoVis toolkit [46]. In these tools simplicity is balanced with efficiency, since graphical scenes can contain large numbers of elements.

While most toolkits propose a graphics API on top of a programming language, there are some languages developed specifically to describe graphics. Baudel proposes a language that is able to describe a large class of linear visualizations such as scatterplots and treemaps [5]. The language is said to be “compact” and “canonical”, in the sense that all text in a program is dedicated to graphics description and the textual description cannot be reduced further. The language uses a data-flow programming style, similar to Processing [33]. Wilkinson designed a grammar and a textual language to describe graphics, together with nVizn, a toolkit based on this language [96]. Protovis is another language dedicated to graphics description which, according to the authors, lowers the entry barriers to creating new visualizations [12].

Runtime adaptation. The execution environment of interactive programs can vary in terms of input and output devices available: mouse, keyboard, trackball, touchscreen, speech input, small or large display, etc. Programmers therefore need ways of describing what devices they wish to use and how. This was first recognized for input devices [42, 60, 84], probably because they have varied more in the past and because their structure cannot always be abstracted: a mouse has buttons, for instance. This need for describing devices includes numerous low-level pragmatic aspects, so the programmer of an interactive system must sometimes turn into a device driver programmer [18].

With these considerations comes the question of allowing programmers to choose between device-dependent and device-independent architectures. The Seeheim architecture of the mid-1980s aims to separate interaction from program logic, which at the time allowed the adaptation from text to graphical interfaces [77]. PAC-Amodeus also offers concepts to adapt to various interaction devices [68].

However, while these systems enable the programmer to abstract input and address the moving target issue as described by Myers [60], some tools may deprive programmers from the benefits of new technologies [42]. Nonetheless, the need to separate the description of the application from the description of the environment led to a series of works on adaptability or plasticity. This converged with the introduction of model-driven user interface development, where the user task and the environment are described with different language [90, 91].

Interaction modalities. Programmers of interactive software need to produce code that allows humans to interact with the machine. This potentially covers a huge range of interaction modalities and languages, from low-level perception-action loops to natural spoken language. In the absence of a grand unified theory of interaction this means many different and potentially mutually incompatible sets of concepts, from high level logic to physical models. This explains why various proposals, more or less distant from the concepts of the Turing machine, have been made for describing different modalities or concerns.

State representation. Various interaction modalities resemble network protocols in that the state of each party and its variations capture the nature of the interaction. This is true of command line dialogue, for which finite state machines were proposed early [67]. This also holds for individual widgets (buttons, menus, etc) for which state machines were reused, then improved with Statecharts [39]. State machines have also been proposed for direct manipulation and multimodal interfaces [17, 58]. More recently, they became so ubiquitous that they became a central part of UML diagrams, and adapted for mainstream languages [2].

Connections between properties. Some parts of direct manipulation, and more generally low level perception-action coupling, are best described as connections between properties of objects. This has led researchers to propose data-flow as a control mechanism [16, 19, 31, 58]. Some also saw a link with data iteration and proposed to combine this with functional programming.

Time. Animation and some types of time-sensitive input, including multimodal interaction [68], often happen in parallel and require a good representation of time [84, 61]. Specific solutions have been proposed for this, including the use of temporal logic.

Algorithms. Some specific interaction modalities, singularly gesture recognition, rely on computation and algorithms. These fit well in the functional paradigm, less in the more reactive ones. Some have proposed that incertitude in recognition be treated with mechanisms similar to errors and exceptions in imperative languages.

Things become more complex with modern user interfaces that combine all of the above modalities: the requirements add up but none of the proposed solutions satisfy all requirements. Programmers have to choose one solution and deal with the resulting complexity. Several approaches have been proposed to address this issue. One is the introduction of formalisms meant to cover the whole range of possible interactions. See Harrison and Duke [40] and Brun and Beaudouin-Lafon [14] for reviews of this approach.

Generally, there is a growing understanding that reactive programming [42, 47] and more generally parallel programming are more suited to interaction-oriented programming than the traditional sequential programming used for computations. This realization started with the debate on internal and external control [23] and reached the state where a growing number of researchers agree with Wegner that "interaction is more powerful than algorithms" [93], which means that computing is a special case of interaction and not the opposite.

Distribution. Modern interactive systems frequently include distributed users [3] and external and potentially non-anticipated interaction devices [4]. This normally necessitates a distributed system implementation, which places additional burdens on developers and the effects of which cannot be completely hidden from end users.

Important requirements for interaction-oriented distribution include entity naming [56] and concurrency control and consistency maintenance for shared and replicated artifacts [78]. Entity and device discovery and subscription features to allow dynamic adaptation to changing external environments and user needs are also required [4].

3.3 A constructive activity

The activity of constructing an interactive system can be described using Norman's model of action [71]. This provides us with a frame for addressing gulfs and discontinuities in programming tasks: how do the tools make it easier for the programmer to tell what actions are possible, to determine mappings between intention and action, to perform the action (e.g., write the code), to interactively specify the graphical user interface, and to determine the mapping from system state to interpretation? The usability requirements follow two main lines: supporting code production, thus reducing the gulf of execution, and how to helping programmers match code and execution, in order to minimize the gulf of evaluation. In this section we focus on programming as an individual activity; we address collaborative development in section 3.4.

Supporting code production. Code production can be supported by allowing easy imitation of prior systems, by supporting exploration of the potential solution space, and by automating the generation of code where appropriate.

Imitation. Programmers seldom start new programs from an empty page and without prior domain knowledge. Rather, they will often take some code they have already programmed for a similar task and modify it for the new one, or they will search for similar programs or useful program fragments on the Internet and imitate them [70]. Brandt describes this as "opportunistic programming" where programmers perform web searches for "just-in-time learning by doing" or "web auto-completions" when they do not recall the name of an API function [13]. A study of professional developers working with Alice concluded that interaction-oriented programmers need graphical copy-and-paste and histories for each graphical object to enable this type of imitation [50].

Programmers also use prior knowledge at the level of architectures, rules, design patterns and programming plans, based on existing documentation, their own experience as a programmer [26], and on discussions with colleagues [33]. High-level user interface patterns such as those documented by Schummer [83] and Borchers [11] can be used, as can the more technical patterns from the famous "Gang of Four" book [34], which contains numerous examples drawn from graphical toolkits like ET++ [94]: composite for scene graphs, abstract factory for supporting various graphical standards, command for undo facilities, etc. Imitation is not precisely reuse (see section 3.4): software reuse means referencing other elements, not copying them.

Exploration. Translating intentions into actions also requires support for exploratory manipulation of possible solutions. Weinberg argues that exploration of both problem and solution often occur at the same time [95]. This applies fully to interactive systems, where the solution is normally difficult to define without several iterations [59]. Exploration is fostered by homogeneous environments (see section 3.1) and by being able to combine elements, which is the case in Ubit [52] where "brickgets" and behaviors can be combined to construct new interaction techniques (multiple pointers, multiple remote displays, semantic zooming, multi-scale, transparency, and control-menus). The possibility of exploring also assumes progressiveness: for example, Stylos et al. the authors demonstrate that API providing constructors without parameters are easier to explore [89].

Prototype-based languages are often advocated for a more exploratory development of the user interface since changes in slots (such as a graphical aspect) can be dynamically propagated to the whole instances at run-times [63]. Noble compares using prototype-based languages to using a document editor, where copying an existing document is more direct and easy to grasp than using a template [70]. The developers of Apple's Newton also advocate for using prototype-based languages:

> The needs of the user-interface side of the program are different. In contrast to the model, the user interface usually consists of relatively few objects, most of which appear only once in a given context, and most of which are unique in small but significant ways. [. . .] It is easier to reason about and control the interactions of individual objects—the usual requirement for UI programming—when the objects themselves are being programmed directly. [86]

Exploration also benefits from introspection mechanisms that help the programmer understand or access the underlying structure or manipulate the source code. ET++ provide meta-classes for building inspectors [94], InfoVis enables a deep access in the toolkit [46], and HGraph provides mechanisms for the programmer to make sense of the running system [35]. Hudson advocates for method interposition and representation of the code suited for manipulation by programs (e.g., as a tree), so the syntax is only an environmental matter [44].

Automatic programming. Some tools relieve the programmer from coding parts of the interactive system. These include interface builders like Garnet [58], or more recently tools such as Apple's Interface Builder, Qt Designer, Eclipse, Tk Komodo, and Expression Blend from Microsoft; as well as model-based interface generators such as Mickey [72] and HUMANOID [90]. In the same vein, programming-by-demonstration techniques enable the user as a programmer, to program by showing the sequence of actions to the system [24, 54]. This of course raises the issue of how to integrate automatic code generation and more conventional textual coding into the same environment.

Matching code and execution. Once code has been developed, it must be verified against the developers' original intent, validated against the users' needs, and modified as necessary.

Evaluation. Myers notes the low testability of interaction-oriented software; classical regression testing is adapted to computation-oriented programs [59]. In interactive software, it is not clear whether the evaluation steps (perception, interpretation, and comparison to the goal) should be performed on the program as a specification, by verification tools, or on the program as a dynamic artifact. Another difficulty is that the "result" of an interactive systems is not clear-cut. Interaction can lead to several potential solutions that need to be compared, but Myers also observes, while studying how designers specify interactive systems, that it is difficult to compare explored solutions, particularly for descriptions of behaviour [61]. One school of thought considers Model Driven Engineering as a means of avoiding such concerns by performing tests on models and relying on model refinement to working code [76]; however, the effectiveness of this approach is not yet proven.

Debugging. Authors observe how difficult it is to debug interactive software. In some cases, the tools that enable the perception of the program disturb its flow of control and thus its correct behaviour [45]. In others, the perceptible result of the program is exceptionally difficult to relate to the code that produces it [55].

Tools for visualizing program execution have been proposed to address some aspects of this issue. Debugging lenses provide access to the state of the system by enabling the programmer to see information about the attributes of interface elements using movable floating windows [45]. ZStep offers mechanisms for understanding the behaviour of the program by stepping through graphical changes in the user interface, as opposed to stepping through lines of code [92]. The Whyline allows developers to perceive and interpret the state of the system in terms of the actions that produced it: which statement has set which attribute, why does this window encounter this change, and so on [45]. SwingStates provides a visual depiction of its finite state machines' dynamics, allowing them to be related to interface changes [2].

Interactive coding. Of course, constructing a program does not involve sequential execution and evaluation steps, as might be implied by the previous two sections [71]. During the problem-solving process, the programmer plays with potential solutions, conducting a “conversation with the medium” as described by DiSessa [28, 29]. Progression of the work can be helped by giving an appropriate continuous feedback [71]. The gap between the representation of actions and the representations of results can be reduced by tools that enable a continuous flow through the successive actions [33]. Ideally, programming and debugging should be tightly integrated, either by designing development environments with the dual perspective in mind [8], or by having the two occur within the same representation [92, 45]. Being able to perform execution and action within the same medium enables a progressive evaluation of the program being constructed [37]. This is the general idea behind programming-in-the-user-interface (PITUI), in which the user can switch to a programming mode by using affordances in the tool to modify its behaviour [27, 29, 30, 32, 35].

3.4 A process-based activity

The requirements described so far address aspects of the objective expressivity of tools and languages with respect to to the concepts to be described, and the subjective expressivity of the construction of software pieces. Building interactive software requires support for processes where software is not only produced, but also managed over time, either individually or as collective processes including sharing, reuse and communication.

Managing the life cycle. Specific requirements of interactive systems design led to the advent of participative (user in-the-loop) iterative development processes [59]. A number of tools aim to shorten the iteration cycle, particularly those dedicated to prototyping, beginning with SILK [51] and culminating in Microsoft Expression Sketchflow.

Implementing an interactive system requires a choice of languages, toolkits, and design environment. This choice is guided by the needs discussed throughout this article, some of them closely related to life cycle. For instance, developing on top of a portable language or platform and extending it through a library is a means to reduce set-up cost of development tools. This can be implicitly targeted by tool designers, as in SwingStates’ addition of state machines to Java [2]. It can also be addressed explicitly, for example in web browser-based tools like Balsamiq [88] and the Processing IDE HasCanvas [73].

Managing reuse and knowledge capitalization. Reuse is a major concern in software engineering. Its claimed benefits include improved productivity, correctness, efficiency and even safety, since a widely reused solution is likely to benefit from broad testing and maintenance. Reuse can be can be opportunistic or planned and can be seen as a concrete subset of knowledge capitalization.

In interaction-oriented system design, knowledge capitalization can be applied at different levels. At the code and architecture level, some pre-cut lines have been identified. The best known are splitting interaction from application code [23] and abstracting from input devices. This latter is a mainstay of interaction-oriented toolkits, which offer widgets to attach to application code without the requirement to address low-level input device management. In addition, these libraries contribute to the homogeneity of the final interface. However, as argued by Chatty, offering pre-defined “Lego bricks” to interface developers is not enough [18]. As innovation plays an important role in interactive system design, developers will want to build their own bricks or bypass abstraction layers to allow for, e.g., working at the driver level to support a novel input device. Tools can assist developers to address these requirements in two ways: elementary bricks can be assembled into larger ones with the same properties; and homogeneous component APIs help designers to build their own. For reuse to be effective, it is also necessary that development environments support and encourage the creation of reusable artifacts.

In addition to code and architecture concerns, the interactive system domain is rich in models and code design patterns that provide elements of solutions for designers [2, 9, 31, 39, 6]. For example, MVC is a design pattern for code that aims to improve modularity by supporting separation of concerns between input, output, and represented state [80]. MDPC improves modularity by distinguishing the display view from the picking view [21]. Vigo is a pattern that helps implementing interactive systems based on instrumental interaction [49, 6]. Used this way, models and code design patterns allow knowledge capitalization and support best practices. Additionally, models provide formalism and semantics, and can be instrumented, or turned into a libraries.

A higher level of reuse involves creating guidelines or interaction patterns by matching common problems to known solutions [11, 83]. Codification of patterns and architectures also answers a need for shared knowledge to permit communication between team members.

Managing collective development The multiplication of roles in interaction design and development brings new problems in managing and syncing teams working on different tools and objects of interest.

Text-based revision control systems pioneered in early 1970s are widely used and have proven efficient for synchronizing the work of large development teams [81]. Unfortunately, to the best of our knowledge there are no tools of the same maturity for dealing with graphics. Some graphics, particularly vector-based graphics with textual representations like SVG, can be handled by code revision systems; however, interpreting change history for such files is error-prone and tedious.

Iterative processes also reinforce the need for tools such as sandboxes and branching, to free creativity and safely extend the design space, along with a convenient way for combining such heterogeneous artifacts as code, mock-up drawings and gestural grammars. Some propose a glue in the form of a middleware [15]; others answer this need by providing a common framework for multidisciplinary teams like IntuiKit [17] and Microsoft WPF/XAML. Ideally, each member of the development team would be presented with a role-appropriate representation of each development artifact [18].

4 Related work

There have been a number of studies of programming interfaces in areas where the difficulty of programming enforces the need for a careful design. Approaches vary by the targeted users (professional programmers, end-user programmers, novices), the types of tools studied (programming languages or APIs, generic or interaction-oriented tools), and the viewpoint taken (usability study, language or API design, survey).

Novice and End-User Programming. Some of the most prominent early studies of human aspects of programming are by Weinberg [95], Hoc et al. [43] and Soloway and Spohrer [87]. These books primarily address the psychology of novice programmers, focusing on the cognitive difficulties and educational aspects. Weinberg also describes the psycho-sociological aspects of programming [95]. The field of end-user programming is very concerned with usability issues and the design of tools and languages for unsophisticated users. Significant work in this area includes Cypher [24] and Lieberman [53, 54]. Our study targets multi-disciplinary teams which may include both professional programmers and end-user programmers such as designers.

Usability studies. Several studies have addressed the usability of languages and toolkits, including those of Myers [64] and Agarwal [1], the latter reporting on the usability of object-oriented representations. Myers' group has an ongoing project, under the name Natural Programming (`www.cs.cmu.edu/~NatProg/`) which investigates how to design more usable programming languages and systems [66, 75]. There is a SIGCHI group [25] and related web site (`apiusability.org`) devoted to API usability.

Green's cognitive dimensions framework [37] has influenced much research on the usability of computing artifacts, including programming languages. Clarke has applied cognitive dimensions to a class library [82] and Green has investigated the cognitive dimensions of visual languages [38].

Most usability studies we are aware of target general purpose languages or APIs rather than tools for building interactive systems. Exceptions include Ko's study of Alice programmers [50] and Myers' study of the programming practices of graphical designers [61].

Other surveys. This paper is conceptually related to several reviews that have been conducted in to help state directions for future research. Brun and Beaudouin-Lafon's taxonomy attempts to compare formalisms for describing user interfaces according to their expressive power, their generative capabilities, and their extensibility and usability [14]. Our classification partly matches this work, but the proposed taxonomy does not address programming tools.

In [59], Myers addresses the challenges of programming user interfaces: this approach, by focusing on why interactive systems are difficult to build, is similar to ours, but we wanted to identify all the needs, not only the ones related to failures. In a later study, Myers et al. describe and evaluate software tools according to five themes: parts of the user interface addressed, threshold and ceiling, path of least resistance, predictability, and moving targets. Their study identifies successful and less successful approaches, with the aim to draw lessons for the design of future tools [60]. Where Myers classifies the tools themselves, we attempt to capture and classify the underlying requirements for tools.

In the book "Languages for developing user interfaces" several chapters address interaction-oriented toolkits or languages. Hudson identifies how programming languages might better support user interface tools [44]: this approach highlights technical needs, but leaves cognitive and collective aspects aside. Our approach is quite similar to that of Cordy, who looks at the design behind the Turing general-purpose programming language to discover ideas that might be applied to tools for interactive systems [22]. Singh identifies three main requirements for a user interface language: object-orientation, time as a first-class object and interactive programming, but his aim is to identify a unique best language [84]. Finally, Graham provides a summary of the book and related workshop, focussing his discussion on whether interactive system tools development needs one language, several languages or an API [36].

5 Conclusion

In this article we began with an analysis of the differences between computing-oriented programming and interaction-oriented programming in an attempt to understand why the latter appears unreasonably difficult with current tools. We then proposed a requirement analysis framework, the ultimate aim of which is to clarify where these two classes of programming have common requirements and where they diverge, and populated this framework through a broad survey of research in interaction-oriented programming.

In the future we plan to populate the framework with a survey of the requirements underlying computation-oriented programming languages. The fully-populated framework will provide a basis for analysis of the commonalities and discrepancies between requirements for interaction- and computation-oriented programming. Our ultimate aim is to identify mechanisms that will allow us to seamlessly address the requirements of both approaches, in hopes that better development tools can then be designed.

Acknowledgements

This work was supported by ANR (project Istar), Aerospace Valley and FUI (project ShareIT). Comments from the anonymous reviewers provoked significant presentational improvements.

References

1. Ritu Agarwal, Prabuddha De, Atish P. Sinha, and Mohan Tanniru. On the usability of OO representations. *CACM*, 43(10):83–89, October 2000.
2. Caroline Appert and Michel Beaudouin-Lafon. SwingStates: adding state machines to the Swing toolkit. In *Proceedings of ACM UIST'06*, pages 319–322, Montreux, Switzerland, 2006. ACM.
3. R.M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration.* 1993.
4. Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: a physical user interface toolkit for ubiquitous computing environments. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 537–544, New York, NY, USA, 2003. ACM.
5. Thomas Baudel. Visualisations compactes: une approche déclarative pour la visualisation d'information. In *Actes de la conférence IHM'02*, pages 161–168. ACM, 2002.
6. Michel Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of ACM CHI'00*, pages 446–453. ACM, 2000.
7. Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Softw. Eng.*, 30(8):535–546, 2004.
8. O. W. Bertelsen and S. Bødker. Studying programming environments in use: between principles and praxis. In *Proceedings of NWPER'98, the Eighth Nordic Workshop on Programming Environment Research*, 1998.
9. R. Blanch, M. Beaudouin-Lafon, S. Conversy, Y. Jestin, T. Baudel, and Y.P. Zhao. Concevoir des applications graphiques interactives distribuées avec INDIGO. *Revue d'Interaction Homme-Machine*, 7(2):113–140, 2006.
10. P Bono, J. Encarnação, R. Hopgood, and P. ten Hagen. GKS – the first graphics standard. *IEEE Computer Graphics and Applications*, 1982.
11. Jan Borchers. *A Pattern Approach to Interaction Design.* John Wiley & Sons, 2001.
12. Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:1121–1128, 2009.
13. J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. of CHI'09*, pages 1589–1598. ACM, 2009.
14. Philippe Brun and Michel Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of interactive systems. In *Proceedings of HCI'95*, pages 197–212. Cambridge University Press, aug 1995.
15. M. Buisson, A. Bustico, S. Chatty, F-R. Colin, Y. Jestin, S. Maury, C. Mertz, and P. Truillet. Ivy: un bus logiciel au service du développement de prototypes de systèmes interactifs. In *Proc. of IHM'02*, pages 223–226, Poitiers, France, 2002. ACM.
16. S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proc. of UIST'94*, pages 195–204.
17. S. Chatty, S. Sire, J.L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of UIST'04*, pages 267–276. ACM, 2004.
18. Stéphane Chatty. Programs = data + algorithms + architecture, and consequences for interactive software. In *Proceedings of IFIP EIS 2007*, 2007. LNCS, Springer Verlag.
19. Stéphane Chatty, Alexandre Lemort, and Stéphane Valès. Multiple input support in a model-based interaction framework. In *Tabletop*, pages 179–186, 2007.
20. Dominique Clément and Janet Incerpi. Specifying the behavior of graphical objects using Esterel. In *TAP-SOFT, Vol.2*, pages 111–125, 1989.
21. Stéphane Conversy, Eric Barboni, David Navarre, and Philippe Palanque. Improving modularity of interactive software with the MDPC architecture. In *Proceedings of EIS 2007*, pages 321–338, 2008. LNCS, Springer.
22. James Cordy. *Languages for developing user interfaces*, chapter Hints on the design of user interface language features: lessons from the design of Turing, pages 329–340. A. K. Peters, Ltd., 1992.
23. Joëlle Coutaz and L. Bass. Requirements on UIMS's. In *Proceedings of the Workshop on UIMS and Environments, Lisbon*, 1990.
24. Allen Cypher. *Watch What I Do. Programming by Demonstration.* MIT Press, 1993. 652 pages.
25. John M. Daughtry, Umer Farooq, Brad A. Myers, and Jeffrey Stylos. API usability: report on special interest group at CHI. *SIGSOFT Softw. Eng. Notes*, 34(4):27–29, 2009.
26. S. Davies. The nature and development of programming plans. *Int. J. of Man-Machine Studies*, 32(4):461–481, 1990.
27. Chris DiGiano and Michael Eisenberg. Self-disclosing design tools: a gentle introduction to end-user programming. In G. Olson and S. Schuon, editors, *In Proc. DIS'95*, pages 189–197. ACM Press, 1995.
28. Andy DiSessa. *Changing Minds: Computers, Learning, and Literacy.* MIT Press, 1999.
29. Andy DiSessa and H. Abelson. Boxer: a reconstructible computational medium. In *Studying the Novice Programmer*, pages 467–481. Lawrence Erlbaum Associates, 1989.

30. Paul Dourish. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM Transactions on Computer-Human Interaction*, 5(2):109–155, June 1998.
31. Pierre Dragicevic and Jean-Daniel Fekete. Support for input adaptability in the ICON toolkit. In *Proceedings of ICMI'04*, pages 212–219. ACM, 2004.
32. Michael Eisenberg. Programmable applications: Interpreter meets interface. *ACM SIGCHI Bulletin*, 27(2):68–93, April 1995.
33. Benjamin Jotham Fry. *Computational information design.* PhD thesis, MIT, 2004.
34. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
35. Tony Gjerlufsen, Mads Ingstrup, Jesper Wolff, and Olsen Olsen. Mirrors of meaning: Supporting inspectable runtime models. *Computer*, 42:61–68, 2009.
36. T. C. Nicholas Graham. *Languages for developing user interfaces*, chapter Future research issues in languages for developing user interfaces, pages 401–418. A. K. Peters, Ltd., 1992.
37. T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of HCI'89*, pages 443–460. Cambridge University Press, 1989.
38. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing, 7*, pages 131–174, 1997.
39. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
40. Michael D. Harrison and David J. Duke. A review of formalisms for describing interactive behaviour. In *ICSE Workshop on SE-HCI*, pages 49–75, 1994.
41. Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of ACM CHI'05*, pages 421–430. ACM, 2005.
42. Ralph Hill. Supporting concurrency, communication, and synchronization in human-computer interaction—the Sassafras UIMS. *ACM Trans. Graph.*, 5(3):179–210, 1986.
43. J-M. Hoc, T. R. G. Green, D.J. Gilmore, and R. Samurçay, editors. *The Psychology of Programming.* Academic Press, 1991.
44. Scott Hudson. *Languages for developing user interfaces*, chapter How programming languages might better support user interface tools, pages 105–113. A. K. Peters, Ltd., 1992.
45. Scott E. Hudson, Roy Rodenstein, and Ian Smith. Debugging lenses: a new class of transparent tools for user interface debugging. In *Proceedings of ACM UIST '97*, pages 179–187. ACM, 1997.
46. Jean-Daniel Fekete. The InfoVis Toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis 04)*, pages 167–174, Austin, TX, October 2004. IEEE Press.
47. Jean-Daniel Fekete and Martin Richard and Pierre Dragicevic. Specification and verification of interactors: A tour of Esterel. In *Proceedings of FAHCI'98*, September 1998.
48. Alan C. Kay. The early history of Smalltalk. In *HOPL Preprints*, pages 69–95, 1993.
49. Clemens Nylandsted Klokmose and Michel Beaudouin-Lafon. Vigo: instrumental interaction in multi-surface environments. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 869–878, New York, NY, USA, 2009. ACM.
50. Andrew Jensen Ko. A contextual inquiry of expert programmers in an event-based programming environment. In *CHI'03 extended abstracts*, pages 1036–1037. ACM, 2003.
51. James A. Landay and Brad Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of CHI'95*, pages 43–50. ACM Press, 1995.
52. E. Lecolinet. A molecular architecture for creating advanced GUIs. In *Proc. of UIST '03*, pages 135–144.
53. H. Lieberman, F. Paterno, and V. Wulf, editors. *End-User Development.* Kluwer/Springer, 2005.
54. Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software.* Morgan-Kaufmann, 2000.
55. Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *ACM CHI'95 Summaries and demonstrations*, pages 480–486. ACM Press, 1995.
56. Blair MacIntyre and Steven Feiner. A distributed 3d graphics library. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370, New York, NY, USA, 1998. ACM.
57. L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of OOPSLA '09.* ACM, 2009.
58. B. Myers, D. Giuse, A. Mickish, B. Vander Zanden, D. Kosbie, R. McDaniel, J. Landay, M. Golderg, and R. Pathasarathy. The Garnet user interface development environment. In *CHI'94 Conference companion*, pages 457–458. ACM, 1994.
59. Brad Myers. Challenges of HCI design and implementation. *Interactions*, 1(1):73–83, 1994.
60. Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.
61. Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *Proceedings of IEEE VL/HCC '08)*, 2008.
62. Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of ACM UIST '91*, pages 211–220. ACM, 1991.

63. Brad A. Myers. *Languages for developing user interfaces*, chapter Ideas from Garnet for Future User Interface Programming Languages, pages 147–157. A. K. Peters, Ltd., 1992.
64. Brad A. Myers. Usability issues in programming languages. Technical report, School of Computer Science, Carnegie Mellon University, 2000. Part of the Natural Programming Project.
65. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Trans. Softw. Eng.*, 23(6):347–365, 1997.
66. Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *CACM*, 47(9):47–52, September 2004.
67. William M. Newman. A system for interactive graphical programming. In *Proceedings of the AFIPS '68 Spring joint computer conference*, pages 47–54, Atlantic City, New Jersey, 1968. ACM.
68. Laurence Nigay and Joëlle Coutaz. A generic platform for addressing the multimodal challenge. In *Proceedings of CHI'95*, pages 98–105. ACM Press/Addison-Wesley Publishing Co., 1995.
69. Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the ACM SIGPLAN Haskell'02 workshop*, pages 51–64. ACM, 2002.
70. James Noble. Prototype based user interfaces. In *Proceedings of the ECOOP'97 Workshop on Prototype Based Object Oriented Programming*, 1997.
71. Donald A. Norman. *The Psychology of Everyday Things.* Perseus Books, 1988.
72. D. Olsen. A programming language basis for user interface. In *Proc. of CHI '89*, pages 171–176. ACM, 1989.
73. Robert O'Rourke. HasCanvas. http://www.hascanvas.com/, April 2009.
74. John K. Ousterhout. *Tcl and the Tk Toolkit.* Flatbrain Com, 1996.
75. J. Pane, C. Ratanamahatana, and B. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. of Human-Computer Studies*, 54(2):237–264, February 2001.
76. J-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front. A Survey of Model Driven Engineering tools for User Interface design. In *Task Models and Diagrams for UI Design*, pages 84–97. Springer Berlin, 2007.
77. Günther E. Pfaff, editor. *User Interface Management Systems.* Eurographics Seminars. Springer, 1985.
78. W. Greg Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Queen's University, Kingston, Ontario, Canada, May 1999. Available from `www.cs.queensu.ca`.
79. Rob Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 1989.
80. Trygve Reenskaug. Models - views - controllers. Technical report, Xerox PARC, December 1979.
81. M. Rochkind. The source code control system. *IEEE Trans on Soft. Engineering*, 1(4):364–370, Dec. 1975.
82. Clarke S. and Becker C. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*, 2003.
83. T. Schummer and S. Lukosch. *Patterns for Computer-Mediated Interaction.* John Wiley & Sons, 2007.
84. Gurminder Singh. *Languages for developing user interfaces*, chapter Requirements for user interface programming languages, pages 115–123. A. K. Peters, Ltd., 1992.
85. J. Six H.-W., Voss. A software engineering perspective to the design of a user interface framework. In *CompSAC'92: Proceedings of the Computer Software and Applications Conference, Chicago 1992*, 1992.
86. Walter R. Smith. Using a prototype-based language for user interface: The Newton project's experience. In *OOPSLA '95*, pages 61–72, 1995.
87. E. Soloway and J. Spohrer, editors. *Studying the Novice Programmer.* Lawrence Erlbaum Associates, 1989.
88. Balsamiq SRL. Balsamiq Mockups. http://www.balsamiq.com/products/mockups/.
89. Jeffrey Stylos, Steven Clarke, and Brad Myers. Comparing API design choices with usability studies: A case study and future directions. In *Proceedings of the 18th PPIG Workshop*, 2006.
90. Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *Proceedings of CHI '92*, pages 507–515. ACM, 1992.
91. David Thevenin and Joëlle Coutaz. Plasticity of user interfaces: Framework and research agenda. In *Proc. of Interact'99*, pages 110–117. IFIP IOS Press, 1999.
92. D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *CACM*, 40(4):38–43, April 1997.
93. Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5), May 1997.
94. Andre Weinand, Erich Gamma, and Rudolph Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.
95. Gerald M. Weinberg. *The Psychology of computer programming.* Dorset House Publishing, 1979.
96. Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing).* Springer., 2005.
97. Terry Winograd. Beyond programming languages. *Commun. ACM*, 22(7):391–401, 1979.

A Cognitive Neuroscience Perspective on Memory for Programming Tasks

Chris Parnin[1]

College of Computing
Georgia Tech Institute of Technology
`chris.parnin@gatech.edu`

Abstract. When faced with frequent interruptions and task-switching, programmers have difficulty keeping relevant task knowledge in their mind. An understanding of how programmers actively manage this knowledge provides a foundation for evaluating cognitive theories and building better tools. Recently, advances in cognitive neuroscience and brain imaging technology has provided new insight into the inner workings of the mind; unfortunately, theories such as program understanding have not been accordingly advanced. In this paper, we review recent findings in cognitive neuroscience and examine the impacts on our theories of how programmers work and the design of programming environments.

1 Introduction

Researchers have long been perplexed in understanding how programmers can make sense of millions of lines of source code text, extract meaningful representations, and then perform complex programming tasks, all within the limited means of human memory and cognition. To perform a programming task, a programmer must have the ability to read code, investigate connections, formulate goals and hypotheses, and finally distill relevant information into transient representations that are maintained long enough to execute the task. Amazingly, programmers routinely perform these mental feats across several active programming projects and tasks in fragmented work sessions fraught with interruptions and external workplace demands.

In coping with these demands and limitations, the programmer must have mental capacity for dealing with large workloads for short periods of time and cognitive mechanisms for maintaining and coordinating transient representations. As of yet, we have no cognitive model that adequately explains how programmers perform difficult programming tasks in the face of constant interruption. As a consequence, we have a limited basis for predicting the effects of interruption or evaluating different tools that may support task-switching for programmers.

Perhaps, new perspectives on memory and programmers are needed. Early models of memory, which we review below, have identified several key processes and provided many fruitful predictions. However, when pressed with more strenuous tasks, such as dealing with an interruption, these models have difficulty accounting for sustained performance [20]. Further, new results continue to emerge from studies of patients with novel brain lesions (injuries to specific brain regions after a stroke or accident) who display behaviors that undermine many of the assumptions of early memory models [55]. Likewise, early perspectives on programmers now seem dated. Shneiderman, who has published several influential articles on programmer memory and comprehension, once likened the ability of musicians to memorize every note of thousands of songs or long symphonies to that of programmers and suggested programmers would obtain the same ability to commit entire programs to memory in exact detail [50]. Rather the opposite has seemed to occur: Programs are not untouched sacred tomes, but organic and social documents that are understood and navigated with the assistance of abstract memory cues such as search keywords and spatial memory within a tree view of documents or scrollbars [27].

The methods available to researchers have expanded greatly. For example, it is now possible to administer drugs that interfere with memory formation or genetically engineer rats, whose basic brain structure for memory is remarkably similar, without the genes for neurotransmitters necessary for consolidating short-term memories into long-term memories. Additionally, fMRI machines provide the ability to measure changes in blood oxygenation levels associated with increased brain activity within 1-2 seconds to regions of brain with 1-3 mm^3 precision [62]. These methods have not been previously available have lead to the founding of a new inter-disciplinary field: *Cognitive neuroscience*, coined by George Miller and Michael Gazzaniga, is "understanding how the functions of the physical brain can yield the thoughts and ideas of an intangible mind" [24]. For researchers studying the cognitive aspects of programmers, never have more opportunities been available to expand our understanding of the inner workings of the programmer's mind.

In this paper, we review perspectives on memory from the cognitive neuroscience literature to gain insight into how a programmer maintains and remembers knowledge used during a programming task. The perspectives on memory offered by classical psychology have difficulty accounting for programmers in practice and have followed us in our formation of theories of program comprehension. Following our review, we discuss implications for the design of programming environments and comprehension theories as well as remaining issues.

2 Memory and Theories of Program Comprehension

2.1 Psychological Studies of Memory

Memory research has had a long and rich history in the psychology community. Here, we briefly attempt to cover some of the key findings.

One of the earliest contributions to memory was Miller's work in 1956 on limitations on information processing. Regardless of what item a participant was being asked to memorize, Miller observed that the capacity for short-term memory appeared to be 5-9 items [35]. Recent research has suggested the actual limit is closer to 4 items [16].

In 1968, Atkinson and Shiffrin presented an influential model of memory called the *modal model of memory* [7]. In the modal model, information is first stored in sensory memory. Attentional processes select items from sensory memory and hold them in short-term storage. With rehearsal, the items can then be moved into long-term storage. The model characterizes the process of obtaining long-term memory as a serial and intentional process with many opportunities to lose information along the way via decay or interference from newly formed memories.

Attempting to refine the modal model's account of short-term memory, in 1974 Baddeley and Hitch introduced the idea of working memory [8] to help explain how items could be manipulated and processed in separate modalities (*e.g.*, visual versus verbal). The original model included separate storage of verbal (phonological loop) and visual-spatial memory with a central executive process that guided attention and retrieval from the stores. In 2000, Baddeley added an episodic buffer which allowed temporary binding of items.

Chase and Simon proposed that experts such as chess players can manage larger mental workloads by learning how to effectively *chunk* information after extensive practice and study [14]. The chunking theory proposes that it takes about 8 seconds to learn a new chunk, and that only about seven chunks can be held in short-term memory. For example, a chess master can outmaneuver an expert player because they can store and recall larger amounts of plausible moves and better assess positions of the chess board.

Several researchers have raised concerns about limitations with the chunking theory. First, information for tasks such as playing chess did not appear to be stored in short-term or working memory (or at least was transfered to long-term memory faster than predicted by chunking theory). Charness found when chess players interpolated playing chess with other tasks long enough to eliminate short-term memory, no or minimal effect on recall was found [12]. Second, chunking theory has a hard time explaining how people performing everyday tasks [19] or experts [20] could handle unpacking and shifting between multiple chunks with such a limited store.

An important alternative to the chunking theory was articulated over a series of papers by Chase, Ericsson and Staszewski [13, 21], who observed mental strategies used by mnemonists and experts. The resulting skilled memory theory identifies two key strategies experts use to achieve their remarkable memory and problem-solving ability: (a) Information is encoded with numerous and elaborated cues related to prior knowledge (similar to Tulving's encoding specificity principle [61]; and (b) experts develop a retrieval structure for indexing information in long-term memory (for example, experts might associate locations within a room with material to memorize – by mentally visiting locations within the room, the expert could retrieve associated items from those locations).

Recently, the skilled memory theory has been extended into the long-term working memory theory, which claims many of the problems with previous theories can be explained if working memory actually involves immediate storage and activation of long-term memories [20].

2.2 Cognitive Theories in the Psychology of Programmers

In studying the psychology of programmers, many researchers devised theories based on notions of memory that were available at the time. For example, many theories use the concept of *chunking* to build cognitive models of programming. Despite the problems noted by other psychologists, many of these notions still persist today. Here, we briefly review current theories of programmer cognition and comprehension.

In top-down comprehension [11] the programmers formulate a hypothesis about the program that is refined by expanding the code hierarchy. The programmers are guided by using cues called *beacons* that are similar to *information scent* in information foraging theory [43]. In bottom-up comprehension [49, 42], the programmer gradually understands code by *chunking* the source code into syntactic and semantic knowledge units. In opportunistic and systematic strategies [28], programmers either systematically examine the program behavior or seek boundaries to limit their scope of comprehension on an as-needed basis. Von Mayrhauser and Vans offered an integrated metamodel [63] to situate the different comprehension strategies in one model.

3 Memory in Cognitive Neuroscience

3.1 Building Blocks of Memory: Long-term Potentiation (LTP)

Like physicists who seek to understand the building blocks of atoms to understand the world, we seek to understand the building blocks of the brain, especially those that contribute to memory. Nearly a century after scientists recognized the atom as a fundamental unit of matter, neuroscientists followed by recognizing that the neurons play a similar role. Certainly, when examining the neuron in depth today, the picture has much changed from the simple view of passive integration of incoming signals, into the view of a complex interplay of voltage-gated ion channels with local synaptic regulation. Here, we focus on the fundamental aspects of a neuron that explains how a brief stimulus from the world can have long-lasting effects on the brain.

The neurological basis for memory is widely believed to be the long-term potentiation (LTP) of neuron synapses. After a synapse undergoes LTP, subsequent stimulus of the synapse will display a stronger response than prior to undergoing LTP. In 1973, Bliss and Lomo [10] first observed LTP after repeatedly stimulating rabbit brain cells and found responses to increase 2-3 times and persist for several hours. Some researchers consider LTP to be a neurobiological codification of the Hebbian learning process: Neurons that fire together, wire together [26].

An interesting aspect of LTP is its various forms of persistence and its connection with memory consolidation. It is now understood that LTP occurs in at least two stages: *early LTP* and *late LTP*. In early LTP, increased response is achieved for a few hours by temporarily increasing the sensitivity and number of receptors at a given synapse occurring within 1-2 seconds [29].

In late LTP, more long-lasting changes involve production of proteins to signal changes to the synapse's surface area and additional dendritic spines associated with stimulation [29].

But how long are these long-lasting changes? In general, synapses undergoing early LTP will return to baseline within three hours. Late LTP, however, has a much longer duration: lasting from several hours or days to months or over years (See Abraham's review on LTP duration [2] for a more in-depth coverage). LTP in the rat hippocampus lasting months and in one instance, one year, has been observed in the laboratory simply after applying four instances of high frequency stimulation spaced by five minutes [3]. In the human brain, newly formed memories are only expected to persist in the hippocampus for a few months or years until system memory consolidation into the neocortex is complete. This is consistent with amnesia patients who have difficulty recalling long-term memories a few months or years prior to their accident [55].

Further neurological processes of memory are of interest such as long-term depression (LTD) and neurogenesis. Whereas LTP increases the efficacy of synaptic transmission, LTD unravels those improvements to make it more difficult for two neurons two fire. The interactions between LTD and LTP are not yet entirely understood; however, it is known that during initial phases of LTP, reversal is more easily accomplished but becomes less so as time passes. If LTP is a mechanism for rapid memorization, are there other possible mechanisms for changes in the brain to occur? In short, yes, with neurogenesis it is possible to grow new neurons and form new growths of white matter. Brain cells were once considered to be like teeth, once lost we could not regrow new brain cells. It has been demonstrated that brain cells routinely die and new ones grow throughout our lives [51]. One of the most striking examples is a study of taxi drivers in London (who need to know very detailed spatial and contextual representations such as street intersections, routes, and traffic conditions of the city) that found when comparing the size of the hippocampus (an area of the brain responsible for remembering associations and spatial memory) of taxi drivers with that of the general population, a significant increase in size was observed and was correlated with time on the job [30].

3.2 Role of Hippocampus in Rapid Memorization

Few medical cases both arrest the imagination and have made a profound impact on memory research as has the story of H.M. [48]. H.M. was a man suffering from severe seizures who elected to have most of his medial temporal lobe bilaterally removed in an attempt to reduce the occurrence of the seizures. Although the surgery was successfully in reducing the seizures, an unforeseen consequence was that H.M. now suffered from anterograde amnesia, a condition where a patient cannot recall or form new memories but can otherwise recall past life events and facts and operate normally. H.M., with very few exceptions, could not learn new semantic facts such as new words or remember recent events such as meeting a person or having a meal. For H.M., retention of new memories generally only lasted a few minutes. If H.M. was having a conversation with a person for the first time, who then left the room and then reentered after a few minutes, afterward H.M. would not have recollection of having met the person or even having a conversation. A detailed analysis of the surgery performed on H.M. indicates that virtually all of the entorhinal cortex and perhinal cortex were removed, about half of hippocampal cortex remained although severely atrophied, and a largely intact parahippocampal cortex [15]. Since H.M., numerous cases have emerged demonstrating how different lesions result in different loss memory abilities; however, the case of H.M. illustrates the essential role of the hippocampus in forming long-lasting memories.

Morris and Frey postulate that the hippocampus provides the ability for an "automatic recording of attended experience" [38]. They argue that many important events cannot be anticipated nor may not occur again, and therefore traces and features of experiences must be recorded in real-time as they happen. Further, Morris makes the argument based on neuroanatomical studies that the hippocampus does not store sensory stimuli directly, but rather associates indices into other cortical regions [39]. For example, the memory of eating a new food at a restaurant is associated with various stimuli (the visual appearance, aroma, taste), contextual details such as the scuffle and movements of other patrons, and semantic details such as the name of the restaurant. The hippocampus is perfectly situated and equipped for this role of automatic association: with very plastic neurons able to undergo LTP and with connections from numerous regions such as visual and auditory pathways having already performed bottom-up processing, and connections with the prefrontal cortex for top-down processing.

Although studies of amnesia patients provide insight into loss of ability, they cannot account for how these systems operate for healthy people. Imaging studies of people performing memorization tasks have provided even more understanding of the hippocampus. In one study, subjects memorized a list of words, and then were asked to recall the studied words [18]. What was unique about this study was that fMRI images were taken while the subjects where studying and recalling the words. The researchers found that failure to recall a word was linked to weaker activity in the hippocampus during memorization; in contrast, success of recalled words was linked to stronger activity. From this study, one could conclude that if a stimulus failed to induce LTP in hippocampal cells at the time of the event, then no conscious memory is likely to remain. Another study has found a similar effect in the entorhinal cortex for items judged to be familiar but not recalled [37].

Research has also found evidence suggesting that specific subareas (*e.g.*, perirhinal or parahippocampal cortices) and specific lateralization (left or right) appear to be associated with different functions (*e.g.*, familiarity recognition or encoding) and different modalities (*e.g.*, spatial vs. verbal). However, it is not still not entirely clear how well we can localize function. For example, the parahippocampus was associated with encoding and recall of spatial memories [44], but activity in the parahippocampus was also found to be highly associated with recognizing objects with unique contextual associations: A hardhat invokes a specific context of dusty construction yards and therefore is associated with higher parahippocampal response; whereas a book has a less specific context and thus less activity [9]. One view put forward by Mayes, proposes that rather than operating on specific modalities of a hard-coded domain, such as verbal specific processing, the hippocampus supports different types of associations —inter-item, within-domain, and between-domain associations —and requires different computations for these associations types [31]. This domain dichotomy view explains why a process such as familiarity recognition may be associated with different regions because recognizing a familiar object would require different processing (and thus different regions) than recognizing a familiar object and location association.

3.3 Memory Organization and Architectures

Memory Types As previously mentioned, researchers distinguish between sensory, short-term, working memory, and long-term memory. For long-term memory, Squire proposed a taxonomy [56] that divides types of long-term memory hierarchically starting with a distinction between non-declarative (implicit) and declarative (explicit) memories. Non-declarative memory includes priming and muscle memory whereas declarative memory includes knowledge of facts and events. Tulving, an influential memory researcher publishing for over 50 years, describes semantic memory as knowledge of facts and episodic memory as a recollection of past events. Tulving's experience with an amnesiac patient E.P., who could learn new facts but not remember how he came to learn about them, lead Tulving to distinguish between our ability to *know* (to recall that the sky is blue) and *remember* (to relieve a past experience via mental time travel) [59].

Studies of patients with newly acquired amnesia have revealed further subtypes of memories. This includes *familiarity*, *recency*, and *source* memories. *Familiarity memory* involves the "feeling of knowing" that an object in a particular context has been encountered before without necessarily recalling the context (*e.g.*, seeing a face in the crowd that seems familiar but does not trigger a name). Familiarity memory is not to be confused with priming. First, in priming, a person previously exposed to an item is more likely to recall that item in the future; however, without a conscious recollection of having been primed. With familiarity, a person is aware that something seems familiarity. Second, priming and familiarity have doubly dissociated brain regions: Familiarity is supported in the entorhinal cortex; priming is believed to involve *modification* of the object representations within perceptual memory (for example, H.M. could be primed only for words he had learned prior to his accident) [45].

Some tasks involve recalling how long ago an event occurred, called *recency memory.* Milner studied patients who underwent surgery affecting the frontal lobes and found certain patients would have difficulty recalling how recently they have seen a word [36]. This suggests the prefrontal cortex plays a role in maintaining and binding a temporal context to memories. Further research has uncovered the importance of top-down involvement of the prefrontal cortex in episodic memory. Although many associations can be remembered in a bottom-up fashion as part of episodic memory, certain types of memories require top-down control and thus direct involvement of the prefrontal cortex.

Often when we learn facts we can associate the initial experience where we learned that fact; these types of memories are called source memory. Activation of the prefrontal cortex is necessary for forming source memories [25].

Memory Systems Since the modal model of memory was proposed in 1968, numerous findings have challenged many of basic premises of the model and, accordingly, several researchers have sought to put forth their own account. Here, we review a few of these models.

In Tulving's serial-parallel-independent (SPI) model [60], rather then providing a mechanistic model of memory, Tulving provides a few guiding principles or generalizations of memory. Simply, he believes the process of encoding a memory to be a serial process (output of one system provides the input for another), the process of storage to be distributed and in parallel (traces of a single stimulus exists in multiple regions of the brain with the potential for later access), and the retrieval of memory to be independent (different systems do not depend on others – once a memory is formed it is available within that system even if others are damaged). This view highlights the importance of viewing memory as a network of coordinating systems rather than an unified store.

In Fuster's account of memory systems [22], he uses a two-stage model derived from anatomical and neuropsychological studies of the brain. In general, raw senses are processed at progressively higher and higher levels of analysis starting from the most posterior region of the brain to the most anterior region of the brain. Fuster divides this journey into two major components: perceptual memory and executive memory. Within the perceptual region, processes and memory start from phyletic sensory memory, then integrating into polysensory, forming into episodic memory, generalizing into semantic memory, and abstracting into conceptual memory. After a brief hop over the motor system, the executive memory region involves concept, plan, program, act and phyletic motor memories. Fuster's account again highlights the specialization and localization of memory, but highlights the importance of top-down and bottom-up processes in memory, and how processing of an stimuli is collocated with its memory.

In Anderson (of ACT-R fame [6]) and colleagues' account of memory [5], a cognitive architecture is composed of several modules responsible for specialized processing of information. Modules have access to buffers which include: goal buffer, a retrieval buffer, a visual buffer, and a motor buffer. Interestingly, the model also includes a production system for learning based on the basal ganglia. The basal ganglia is mainly responsible for motor control; however, it also has been "recruited" by the prefrontal cortex for reward-based learning of rules [34]. As such, the strength the ACT-R model is in simulating learning and problem solving; however, the model is less effective in modeling memory retention (for an exception, see Altmann and Trafton's work on memory for goals [4]).

3.4 PFC: Goal Memory and Executive Processes

Humans fluidly perform and seamlessly switch among different tasks in a day. Routine activities, such as ordering a cup of coffee, can be performed without much cognitive effort. The rules are readily apparent: selecting a cup size, specifying a brew, paying the cashier – yet routine activities can be highly dynamic and even arbitrary. People have no problem adapting the rule to buy a cup of tea instead, or, given a rule never before encountered, *clap if you hear a phone ring*, most people would have no problem performing the task. However, if the rule was instead, *clap if you hear a phone ring in a coffee shop*, then people may fail to remember to apply the rule. Such forgetting would be a failure of *prospective memory*, remembering to remember. How does the brain store and manage prospective memories and other supporting memories needed for performing tasks?

The prefrontal cortex (PFC) is a region situated in the most anterior (toward forehead) portion of the frontal lobe. The PFC, a recent evolutionary addition, extends from the motor control regions of the frontal lobe to provide cognitive and executive control. E. Miller and Cohen [33] provide a compelling and influential account of the PFC. They argue the PFC provides the ability to bias a particular response from many possible choices. For example, when crossing a street, a person may be accustomed to looking left to check for oncoming traffic. However, if that person were an American tourist visiting London, then top-down control would be required to override the typical response and bias it toward a response for looking right first. In this theory, rules, plans, and representations for tasks are learned via highly plastic PFC neurons (a view also shared by Fuster [22]), but may migrate over time. One apt metaphor offered by Miller and Cohen is a railroad switch:

> “The hippocampus is responsible for laying down new tracks and the PFC is responsible for flexibly switching between them.”

The PFC also plays an important role in top-down attention: In early studies of monkey brains, when a food reward was shown to a monkey and then subsequently hidden for a delay period, persistent firing of neurons in the PFC was sustained during the delay period. Despite distracting stimuli, the monkey could recall the location of the food reward. However, monkeys with PFC lesions could not maintain attention and performed poorly at recalling the food [23]. More recent work has uncovered a possible mechanism for how the PFC can simultaneously maintain several active items in mind. When examining the firing patterns of ensembles of neurons, rhythmic oscillations can be observed. These oscillations are believed to encode attributes of an attended item. Siegel and colleagues [52] observed when multiple items need to be attended to, distinct items were maintained in distinct phase orientations of the oscillating signal. Like our ability to wave a string tied to a door knob, our limit to attend multiple objects may be simply bound to a limit of speed and space for separating items within a frequency spectrum (a problem well known in telephone and ethernet communications). An interesting benefit emerging from phase coding of items is “free” temporal order of those items. In the same experiment, when the order of items were misremembered, there was a correlation with inadequate phase separation of the encoded items: The signal still preserved enough information to represent the items, but not enough information was available to determine order. This view offers an interesting alternative to the concept of working memory. The prefrontal cortex can maintain many representations for tasks (especially if the representations refer to associations within the hippocampus), but can only attend to a few at a time.

Understanding how cognitive control occurs in the prefrontal cortex is still an ongoing research question. However, researchers have been successful in understanding how the prefrontal cortex supports one type of process that is important for suspension of tasks —prospective memory. *Prospective memory* is remembering to remember to perform an action in the future under a specific context (*e.g.*, setting up a mental reminder to buy milk on the way home from work) [64]. Often intentions appear to spontaneously pop into mind prior to an important event or sometimes unfortunately later than intended. Researchers have sought to understand the underlying mechanisms for prospective memory. Essentially, some researchers believe prospective memory requires some form of attentional resources [54]; whereas other researchers believe if reminder cues are readily available then the process could be automatic [32]. A recent fMRI study has found that depending on the nature of the intention, prospective memory could involve both strategic monitoring and automatic retrieval from cues [46].

4 Task Memory Model

Understanding both brain structures and the associated jargon (*e.g.*, *dorsolateral prefrontal cortex*) can be a daunting endeavor for anyone. Here, we present the task memory model in part to summarize insights from cognitive neuroscience literature on memory but also to abstract from the intricacies of the brain and nomenclature. Our model shares similar goals with the *stores model* [17], where Douce argues multiple modalities, such as spatial memory, play a crucial role in code cognition. Our model intends to go further, by first accounting for the underlying constraints of different memory, and then reconnecting these constraints to memory requirements in programming tasks and design of programming environments.

For the purpose of the forthcoming discussion, we introduce the term *task memory*, which is the set of constructs (such as goals) and processes (such as suspension) needed to perform tasks. Our goal is to explain how people such as programmers can maintain representations for complex and long-running tasks (over the course of many hours or several days) despite interruptions or task-switches. In defining task memory and its corresponding model, not only do we want to avoid the ambiguity of a term such as working memory, we also want to go further by specifying task related concepts such as suspension or goals and relate them to specific processes and localized function within the brain.

4.1 Memory Pathways

As we perceive sensations from the world, those sensations flow along pathways that actively process and interpret perceptions of our world. The impression of these perceptions is what we understand as memory. Even purely internal events, such as our inner thoughts, will activate the same motor speech areas and auditory comprehension pathways (*i.e.*, subvocalization) as listening to ourselves talking. Therefore, to speak strictly in terms of storage, would be to misunderstand memory —the storage of memory is interleaved with the same pathways that process and and later recognize past sensations.

We divide the storage pathways of task constructs into three regions: frontal region, associative region, and perceptual region (see Figure 1).

Perceptual Region The perceptual region contains both primitive and salient representations of stimuli. This region is segmented into visual, spatial, and semantic (including auditory) areas. Each area is responsible for interpreting and storing representations of stimuli. These representations are linked so that spreading activation is possible for learned concepts.

There are short-term effects of perceiving a stimulus. Short-term retention occurs locally, allowing for example same/different comparisons to be made. In addition, a stimulus will prime representations, but only at the level of which a person has previous experience (*e.g.*, a person can be not be primed for the semantic meaning of a word they have never learned, but under the right conditions they can be primed for the visual perception of the word).

Associative Region The associative region receives inputs upstream from each area of the perceptual region. The associative region has several interesting capabilities. The associative region is capable of receiving several distinguishing features (such as visual and semantic feature) and can create a resulting association. The formation of the association is fast and automatic —an autoassociative encoding of perceptual features —but the features are not stored, but rather indices into representation sites in the perceptual region. Associations are not formed for every stimuli but instead are selectively formed based on stimuli strength (attention and and novelty detection play a large role).

These associations are formed in such a way that activation of any one of the features will activate the indices of other associated features, which will in turn activate the representations within the perceptual region. The duration of an association can last several hours, but if strengthened can last days and in some cases years. However, associations can be overturned, or may not form in the first place if similar associations already exist.

Finally, the associative region is capable of encoding familiarity. Encoding familiarity allows stimuli to be identified more readily without requiring representations of the stimuli to be well formed. This will allow a feature to be recognizable (*e.g.*, a face) but not associated with other features (*e.g.*, a name).

Frontal Region The frontal region contains important pathways for interpreting perceptions, selecting responses, forming and attending to representations and goals, and directing learning and memory. Pathways in the frontal region are well connected the perceptual and associative regions, allowing multiple pathways to accessing representations and imposing top-down influence. Within the frontal region, important pathways exist for managing tasks such as monitoring and switching tasks.

Memory supported by the frontal region includes prospective, source, recency memory. The frontal region provides the primary infrastructure for holding a task's plans, goals, and task-relevant bindings. The duration of these task elements do not fade like short-term memory, but persist for hours or days. Task-relevant bindings do not store items directly, but rather refer to long-term memory or stores within the associative and perceptual regions. Finally, the frontal region provides infrastructure for reminders to "pop into the mind" in the presence of appropriate cues.

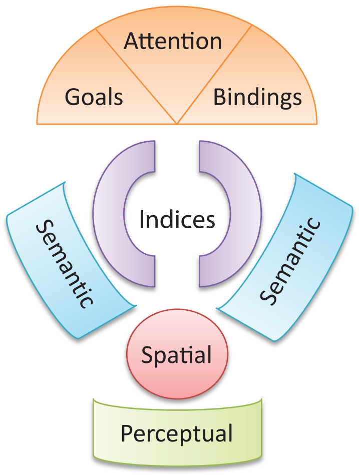


Fig. 1. Task memory model.

5 Considerations

We have presented several possible mechanisms underlying memory and associated cognitive control processes. In this section, we consider possible ramifications and speculate on impact on programming environments and theories of program comprehension. Design elements for programming environments has been discussed before [57]; here, we describe elements not previously covered in light of new findings in memory and considerations such as interruptions and multitasking. Remember, these ideas are not claims but a line of inquiry.

5.1 Programming Environment Support

Development tasks typically require coordinating software changes across multiple locations in a programs source code. Programming environments have provided limited support for managing the active artifacts relevant to the programming task. How a programmer represents these items in memory can inform how to better design programming environments.

Names and Notes Programs are comprised almost entirely of names. People forget or have difficulty recalling names on a daily basis. Some names in programs are for common operations such as iteration or familiar concepts such as sorting. But for many program elements, we may know the face, but not the name.

To find or write code, programming environments require either knowing the name precisely or partially, in the case of name completion system or class names. But unlike everyday objects, we do not have other aspects to assist in recall. We cannot call upon temporal or contextual clues, "what was that object I saw yesterday when I was debugging?". We cannot easily ask spatially "what was the object near the parsing code", nor semantically "what object was for checking security". Programmers still try asking these questions, they just have to find creative (but costly) ways of answering them.

For written notes, we often invent our own names to represent a current thought. We might write down, "labels" or "security". Presumingly, we want to record a prospective reminder to perform some action for a programming task. Amazingly, when looking at an old note, we can often recall the purpose of the note and the circumstance in which we wrote it. However, other times we do not even remember writing down the note or the note only triggers a vague recollection. For programmers, although notes have benefits in low overhead and conciseness, they are deficient when capturing detailed and delocalized knowledge. When notes fail to capture appropriate detail, programmers have to resort to costly information-seeking activities such as navigating source code or viewing source code history to rebuild their working context. Ultimately, neither notes nor environmental cues fully utilize program structure or state within programming environments, and more importantly, neither notes nor environmental cues digitally link together. Support for easily attaching notes to cues could create quick and powerful reminders: For example, pinning down a virtual sticky note on a code document or on a file within the document treeview.

Levels of Support: Memory and Time In Table 1, we consider different levels of support for interruption recovery based on decay of task memory. When suspending a task for a few minutes, what is at most risk is the loss of an ensemble of well-crafted thought. Humans are limited by the ability to simultaneously maintain attention to mental thoughts. Thus, a short-term interruption may not necessarily erase the memory of those thoughts, but we may never again find that insightful combination of those thoughts attended simultaneously with the same active top-down representations.

When suspending a task for a few hours, many newly formed associations and representations may still be intact. Upon return to the task, the programmer may need a brief reminder to reactivate suspended task goals and representations; it is not likely they would have forgotten these yet. In support of this process, programmers may need a quick refresh of the artifacts to help restore the details of the representations. During the task suspension, weak associations may have faded. Programmers may forget a relationship they discovered between code items or not recall where items are located.

Programmers returning to a task after several days require a different level of support. After such a delay, details such as new names of identifiers may have faded, and many representations used for the task may no longer be active. Traces of memories will guide the programmer in returning to work: Some code sections will feel more familiar than others. Further, external cues, such as jotted down goals, will help guide navigation and jump-start work. Finally, episodic recall of activity will help restore plans and potentially identify what actions to perform next.

Interval	Support
minutes	Support for managing attention.
hours	Brief reminder to restore top-level goals. Support for restoring artifacts. Simple associative cues such as words from familiar code symbols effective.
days	Support for restoring representations. Semantic-based interfaces less effective, use episodic-based interfaces.
weeks	Most representations have faded. Focus on restoring goals and plans.

Table 1. Different length intervals of task suspension require different types of support from the programming environment when resuming.

Environmental Cues and Beyond Observations of developers suggest they frequently rely on cues for maintaining context during programming. For example, Ko *et. al* [27] observed programmers using open document tabs and scrollbars as aids for maintaining context during their programming tasks. However environmental cues often do not provide sufficient context to trigger memories: In studies of developer navigation histories, a common finding is that developers frequently visit many locations in rapid succession in a phenomenon known as *navigation jitter* [53]. Navigation jitter has been commonly attributed to developers flipping through open tabs and file lists when trying to recall a location [53, 41]. Environmental cues such as open tabs may be insufficient because *what* a developer remembers may be spatial and textual cues within the code document and not the semantic or structural location of the code element when automatically encoding working state [58].

By enriching environmental cues to take more advantage of the temporal, spatial, and contextual aspects we have previously discussed we would expect improvements to programmer productively. Research comparing development interfaces using names or *content* that a name refers to has shown that names are slower and less accurate than content [47], and content is strongly preferred over names when presented temporally [40]. Cues should enhance both an item's recency and familiarity. Temporal order of visiting an item should be easily discoverable. The context of visiting an element should also be clear: Tabs or files can be more understandable if it was made clear how a programmer visited the item (*e.g.*, indicate if a file was edited, visited from stepping through a debugging session, or found from a search result [with search keyword used to find it]). Other artifacts can be important cues for a programming task: events on calendars, meeting notes, checkins from source control, and emails from colleagues. Exploring how to collect, integrate and present these various cues offers an exciting research challenge.

5.2 Theories

Here, we consider some implications to current programming theories of comprehension and provide some concepts for developing richer theories of program comprehension.

Visual Chunks In Shneiderman and Mayer's syntax/semantic model [49], programmers do not retain memories of syntax, but only their meanings. This conclusion was reached based on the programmer's ability to *exactly* recreate a program statement: *i.e.*, even changing a symbol from i to x would invalidate that statement. By these measures, programmers tended to perform poorly when exactly reproducing the syntax of statements recently read, but instead retained their meanings.

When the syntax/semantic model was conceived, it was based on a variation of the modal model of memory (items move from sensory memory, short-term memory, and then long-term memory through active rehearsal). For anyone that has read paragraphs of text or lines of code, such a model may seem counter-intuitive. Unlike attempting to rehearse a phone number, when we read text, we do not frequently stop to remember the words or meaning, neither do we pause when engaged in casual conversations.

We propose that semantic meanings of read program statements are retained without intentional rehearsal, but instead with autoassociative support from the hippocampal formation. In contrast to the syntax/semantic model, we suggest memory of syntax is retained —not in an exact memorization of characters of text —but via abstracted perceptual patterns or visual sketches. For example, a certain region of code containing many distinct patterns of for loops and operations with character strings produces an unique signature of text indention and syntax highlighting that would be recognizable when quickly scanning source code. Such an ability would be advantageous to programmers who need to quickly and frequently switch documents and skim through code without having to deeply process the text in order to recognize relevant bits.

We introduce the concept of *visual chunks*, regions of code which may not yet have any strong semantic association, but which have perceptual features that are familiar and recognizable by a programmer. Visual chunks can be associated with temporal and contextual details such as a search term or hypothesis used in finding the visual chunk. Visual chunks can also be associated spatially within each other (*e.g.*, above or below another visual chunk). Finally, visual chunks can associated with subvocalized inner thought, giving it an internal nickname.

Iterative Comprehension The syntax/semantic model suggests that programmers use previously learned schemas (programming plans) to interpret text into semantic chunks in a hierarchal process. Alternatively, top-down theories explain that programmers parse code based on their current level of understanding and goals. Neither theory details the structures or mechanisms necessary for partial understanding of code or explain how a programmer can maintain these intermediate representations of unfamiliar code when switching between multiple tasks as observed in our recent experiment [40]. Opportunistic theories do not fare better: As programmers do not necessarily – upon reaching a new understanding – revisit every previously encountered item to update its understanding, but rather a programmer must have some form of intermediate representation in mind. Finally, a failing of all of these theories is their inability to identify exactly when learning occurs.

We also introduce the concept of *iterative comprehension*. With iterative comprehension, a programmer uses autoassociative memory of processed perceptual events to rapidly record many traces and facts about a program, even without having seen the code before. The programmer can draw upon numerous resources —familiarity, spatial, visual, auditory, autoassociative, and prospective memory, each involving distinct parts of the brain —that collectively allow the programmer to maintain partial representations when solving a problem.

For a new program, a programmer initially gathers numerous visual chunks when exploring the program. As the programmer learns more about the program, she iteratively updates previous visual chunks with knowledge of new events or relates with top down concepts and goals. The programmer can take advantage of previously learned schemas to provide strong associations with events and rapidly consolidate new facts. This explains how a programmer can retain memory of semantic properties of code while also associating other visual and spatial properties.

Here, we have only provided a sketch of what iterative comprehension entails. However, we believe iterative comprehension may provide a more compelling account of how programmers manage programming knowledge and can explain how programmers are able to explore and keep track of many items beyond traditional accounts of memory while only having partial knowledge of the code.

6 Remaining Issues and Future Questions

Several directions can be taken to move ideas presented in this paper forward. For the most part in our discussions on structures within the brain we have omitted detail on how structures differ based on location within the left or right hemispheres of the brain, also called *lateralization* of the brain. Extending models to include lateralization is both necessary for brain imaging studies and understanding the dual but separate roles that a structure plays (such as differences in encoding and retrieval in the left and right hippocampus).

Computational architectures for cognitive models such as ACT-R [6] or SOAR [1] are steadily improving. Still, these models are dealing with relatively simple tasks. Recently, Altmann and Trafton's work on memory for goals [4] have made modifications of the ACT-R architecture to include the ability to model the effect of interruptions on goal memory. Situating our work within these models would provide a mutual benefit of validating these ideas while suggesting modifications to the computational architectures.

Finally, despite new advances in memory research, there remains numerous unresolved issues. The biological mechanisms for forming memory are still not fully understood: One striking observation has been that spatial memory appears to use distinct processes when compared to those used in the normal associative processing occurring in the hippocampus. We do not yet understand the impact this has on encoding and consolidation of spatial memories. A strength of memory research has been the various lines of evidence used to investigate memory. But this is also a weakness: Some findings have only been established in animal studies, which may not hold in the same manner for humans. Further, these approaches have been effective at finding dissociations between brain regions and memory types but not in understanding how these regions coordinate and what information they carry. Finally, much care must be taken when using the results of fMRI studies; if not carefully guarded, poor statistical designs can allow over broad interpretations.

7 Conclusion

Nearly 40 years have passed since some of the earliest cognitive models of programmers have been proposed. Both the programming landscape and our understanding of the human brain have dramatically changed. Unfortunately, in the time since, the impact on practicing programmers has been negligible; the predictive power nearly non-existent; and, our understanding of the mind furthered little beyond common sense.

In this paper, we have outlined the background, tools, concepts and vocabulary for a challenging but hopefully rewarding trek forward. By understanding how a programmer manages task memory, especially in the context of multi-tasking and interruptions, we can begin to unravel this mystery.

References

1. Newell A. *Unified Theories of Cognition.* Harvard University Press, Cambridge, MA, 1990.
2. Wickliffe C. Abraham. How long will long-term potentiation last? *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 358(1432):735–744, April 2003.
3. Wickliffe C. Abraham, Barbara Logan, Jeffrey M. Greenwood, and Michael Dragunow. Induction and Experience-Dependent Consolidation of Stable Long-Term Potentiation Lasting Months in the Hippocampus. *J. Neurosci.*, 22(21):9626–9634, 2002.
4. E. M. Altmann and J. G. Trafton. Memory for goals: An activation-based model. *Cognitive Science*, 26:39–83, 2002.
5. J. R. Anderson, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1050, 2004.
6. John R. Anderson and Christian Lebiere. *The Atomic Components of Thought.* Erlbaum, Mahwah, NJ, June 1998.

7. R. C. Atkinson and R. M. Shiffrin. *The psychology of learning and motivation (Volume 2)*, chapter Human memory: A proposed system and its control processes, pages 89–195. Academic Press, 1968.
8. A.D. Baddeley and G. Hitch. *The psychology of learning and motivation: Advances in research and theory*, chapter Working memory, pages 47–89. Academic Press, New York, 1974.
9. Moshe Bar and Elissa Aminoff. Cortical analysis of visual context. *Neuron*, 38(2):347–358, April 2003.
10. T. V. Bliss and T. Lomo. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of physiology*, 232(2):331–356, July 1973.
11. Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
12. N. Charness. Memory for chess positions: resistance to interference. *Journal of Experimental Psychology: Human Learning and Memory*, 2:641–653, 1976.
13. W. G. Chase and K. A. Ericsson. *The psychology of learning and motivation*, volume 16, chapter Skill and working memory, pages 1–58. Academic Press, New York, 1982.
14. W.G. Chase and H.A. Simon. Perception in chess. *Cognitive Psychology*, 4:55–81, 1973.
15. S. Corkin, D. G. Amaral, R. G. González, K. A. Johnson, and B. T. Hyman. H. m.'s medial temporal lobe lesion: findings from magnetic resonance imaging. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 17(10):3964–3979, May 1997.
16. N. Cowan. The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *The Behavioral and brain sciences*, 24(1), February 2001.
17. Christopher Douce. The stores model of code cognition. In *Programmer Psychology Interest Group*, 2008.
18. L. L. Eldridge, B. J. Knowlton, C. S. Furmanski, S. Y. Bookheimer, and S. A. Engel. Remembering episodes: a selective role for the hippocampus during retrieval. *Nature neuroscience*, 3(11):1149–1152, November 2000.
19. K. A. Ericsson and P. F. Delaney. *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*, chapter Long-term working memory as an alternative to capacity models of working memory in everyday skilled performance, pages 257–297. Cambridge University Press, Cambridge, UK, 1999.
20. K. A. Ericsson and W. Kintsch. Long-term working memory. *Psychological Review*, 102(2):211–245, 1995.
21. K. A. Ericsson and J. J. Staszewski. *Complex information processing: The impact of Herbert A. Simon*, chapter Skilled memory and expertise: Mechanisms of exceptional performance, pages 235–267. Lawrence Erlbaum, Hillsdale, NJ, 1989.
22. J. M. Fuster. The prefrontal cortex–an update: time is of the essence. *Neuron*, 30(2):319–333, May 2001.
23. J. M. Fuster and G. E. Alexander. Neuron activity related to short-term memory. *Science (New York, N.Y.)*, 173(997):652–654, August 1971.
24. Michael S. Gazzaniga, Richard B. Ivry, and George R. Mangun. *Cognitive neuroscience: the biology of the mind*. Norton, 3rd edition, 2009.
25. E. L. Glisky, M. R. Polster, and B. C. Routhieaux. Double dissociation between item and source memory. *Neuropsychology*, 9:229–235, 1995.
26. D.O. Hebb. *The organization of behavior*. Wiley, New York, 1949.
27. Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, 2006.
28. David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
29. M. A. Lynch. Long-term potentiation and memory. *Physiological reviews*, 84(1):87–136, January 2004.
30. E. A. Maguire, D. G. Gadian, I. S. Johnsrude, C. D. Good, J. Ashburner, R. S. Frackowiak, and C. D. Frith. Navigation-related structural change in the hippocampi of taxi drivers. *Proceedings of the National Academy of Sciences of the United States of America*, 97(8):4398–4403, April 2000.
31. Andrew Mayes, Daniela Montaldi, and Ellen Migo. Associative memory and the medial temporal lobes. *Trends in cognitive sciences*, 11(3):126–135, March 2007.
32. M. A. McDaniel and G. O. Einstein. Strategic and automatic processes in prospective memory retrieval: A multiprocess framework. *Applied Cognitive Psychology*, 14:127–144, 2000.
33. E. K. Miller and J. D. Cohen. An integrative theory of prefrontal cortex function. *Annual review of neuroscience*, 24(1):167–202, 2001.
34. E.K. Miller and T.J. Buschman. *The Neuroscience of Rule-Guided Behavior*, chapter Rules through recursion: How interactions between the frontal cortex and basal ganglia may build abstract, complex, rules from concrete, simple, ones, page (in press). Oxford University Press., 2007.
35. G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. 1956. *Psychological review*, 101(2):343–352, April 1994.
36. B. Milner, P. Corsi, and G. Leonard. Frontal-lobe contribution to recency judgements. *Neuropsychologia*, 29(6):601–618, 1991.
37. Daniela Montaldi, Tom J. Spencer, Neil Roberts, and Andrew R. Mayes. The neural system that mediates familiarity memory. *Hippocampus*, 16(5):504–520, 2006.

38. R. G. Morris and U. Frey. Hippocampal synaptic plasticity: role in spatial learning or the automatic recording of attended experience? *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 352(1360):1489–1503, 1997.
39. R. G. M. Morris. Elements of a neurobiological theory of hippocampal function: the role of synaptic plasticity, synaptic tagging and schemas. *European Journal of Neuroscience*, 23(11):2829–2846, 2006.
40. Chris Parnin and Robert DeLine. Evaluating cues for resuming interrupted programming tasks. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 93–102, New York, NY, USA, 2010. ACM.
41. Chris Parnin and Carsten Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, 2006.
42. Nancy Pennington. Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
43. Peter Pirolli and Stuart K. Card. Information foraging. *Psychological Review*, 106:643–675, 1999.
44. C. J. Ploner, B. M. Gaymard, S. Rivaud-Péchoux, M. Baulac, S. Clémenceau, S. Samson, and C. Pierrot-Deseilligny. Lesions affecting the parahippocampal cortex yield spatial memory deficits in humans. *Cerebral cortex (New York, N.Y. : 1991)*, 10(12):1211–1216, December 2000.
45. B. R. Postle and S. Corkin. Impaired word-stem completion priming but intact perceptual identification priming with novel words: evidence from the amnesic patient h.m. *Neuropsychologia*, 36(5):421–440, May 1998.
46. Jeremy R. Reynolds, Robert West, and Todd Braver. Distinct neural circuits support transient and sustained processes in prospective memory and working memory. *Cerebral cortex (New York, N.Y. : 1991)*, 19(5):1208–1221, May 2009.
47. Izzet Safer and Gail C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 229–243, New York, NY, USA, 2007. ACM.
48. W. B. Scoville and B. Milner. Loss of recent memory after bilateral hippocampal lesions. 1957. *The Journal of neuropsychiatry and clinical neurosciences*, 12(1):103–113, 2000.
49. B. Shneiderman and R. Mayer. Syntactic semantic interactions in programmer behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, June 1979.
50. Ben Shneiderman. *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers, 1980.
51. T. J. Shors, G. Miesegaes, A. Beylin, M. Zhao, T. Rydel, and E. Gould. Neurogenesis in the adult is involved in the formation of trace memories. *Nature*, 410(6826):372–376, March 2001.
52. Markus Siegel, Melissa R. Warden, and Earl K. Miller. Phase-dependent neuronal coding of objects in short-term memory. *Proceedings of the National Academy of Sciences of the United States of America*, 106(50):21341–21346, December 2009.
53. Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, 2005.
54. Rebekah E. Smith. The cost of remembering to remember in event-based prospective memory: investigating the capacity demands of delayed intention performance. *Journal of experimental psychology. Learning, memory, and cognition*, 29(3):347–361, May 2003.
55. H.J. Spiers, E.A. Macguire, and N. Burgess. Hippocampal amnesia. *Neurocase*, 7:352–382, 2001.
56. Larry R. Squire. Memory systems of the brain: a brief history and current perspective. *Neurobiology of learning and memory*, 82(3):171–177, November 2004.
57. M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44(3):171–185, 1999.
58. J. Gregory Trafton, Erik M. Altmann, Derek P. Brock, and Farilee E. Mintz. Preparing to resume an interrupted task: effects of prospective goal encoding and retrospective rehearsal. *International Journal of Human-Computer Studies*, 58:583–603, 2003.
59. E. Tulving. *Organization of memory*, chapter Episodic and semantic memory, pages 381–403. Academic Press, New York, 1972.
60. E. Tulving. *The Cognitive Neurosciences*, chapter Organization of memory: Quo vadis?, pages 839–847. MIT Press, Cambridge, MA, 1995.
61. E. Tulving and D. M. Thomson. Encoding specificity and retrieval processes in episodic memory. *Psychological Review*, 80:352–373, 1973.
62. A. Villringer. *Functional MRI*, chapter Physiological Changes During Brain Activation, pages 3–13. Springer, 2000.
63. A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tools capabilities. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 230–239, July 1993.
64. E. Winograd. *Practical Aspects of Memory: Current Research and Issues*, volume 2, chapter Some observations on prospective remembering, pages 348–353. Wiley, Chichester, 1988.

Perceived Self-Efficacy and APIs

John M. Daughtry III

Applied Research Laboratory
The Pennsylvania State University
daughtry@psu.edu

John M. Carroll

College of Information Sciences and Tech.
The Pennsylvania State University
jcarroll@ist.psu.edu

Keywords: POP-II. B. Program Comprehension, Scenario-Based Design; POP-IV. A. Object-Oriented Design; POP-I.A. Group Dynamics; POP-V.B. Research Methodology

Abstract

Application program interface (API) use and design is critical, non-optional, and cross-cutting in the construction of modern software systems. However, only recently has the explicit study of API design process and API designs been initiated from the perspective of usability, and little is known with respect to how various forms of information about an API aids programmers in the use of an API. In this paper, we present findings from our exploration of perceived self-efficacy (PSE) for API use. First, we describe the development of a novel PSE instrument that focuses on the task of using an API. Second, we evaluate the validity and sensitivity of the instrument with respect to changes in the information given professional programmers about an API. To accomplish this goal, we articulate and utilize two complementary forms of API documentation grounded in scenario-based design. Through this work, we demonstrate the validity of the evaluation tool and raise questions about the perceived value developers place on information about an API and its intended use.

1. Introduction

Modular programming is ubiquitous in modern software application construction. The benefits we seek of modularity are technical, psychological, and organizational in nature (Parnas 1972). Using modules affords reuse, which in turn reduces duplication. Thus, there is less code to maintain. Modularity also reduces code into small pieces that can be more easily understood by a developer given humanity's constrained capabilities for cognition. One can imagine the difficulties in trying to read and edit the source code of any modern software system were it contained in a single structure. Finally, modularity affords the breakdown of work in teams by allowing groups and individuals to focus on discrete pieces of a system.

Because of the prominence of modularity, the use of any popular modern programming language necessitates the use of APIs (Stylos and Myers 2007). The centrality of APIs within the context of programming is of growing importance (Stylos 2009). For example, at a minimum, Java programmers make use of the Java platform SDK (JDK), and C# programmers make use of the .NET Framework. Even to output "hello world" requires utilization of the APIs provided with the language. Note that we use the term API broadly, reflecting the programming vernacular (e.g., de Souza, Redmiles, Cheng, Millen, and Patterson 2004; Bloch 2005). This definition implies that every module has an API (the interface through which the module is invoked), which is in contrast to stricter definitions that define APIs in terms of interchanges between applications (e.g., Software Engineering Institute 2008). However, it more closely aligns with practice, as reflected in Bloch's argument that "… if you program, you are an API designer, whether you know it or not, because good code is modular, and those inter-modular boundaries are effectively APIs" (2005).

In the wild, API design perfection is unattainable (Bloch 2005). Indeed, interface design is about navigating a design space of trade-offs as opposed to finding the optimal design (Carroll 2000). For example, as de Souza and colleagues describe, information hiding has significant organizational communication drawbacks, despite the technical benefits (2004).

Given the centrality and rising importance of APIs in professional programming, our research agenda is to expose the psychological aspects of API design and use as a leverage point for impacting the practice of programming via empirical data that supports interaction design for APIs themselves and the tools used by API consumers.

1.1. API Usability

Extant work in API usability analysis focuses on traditional usability laboratory studies (e.g., Clarke 2004). In essence, the researcher sets out to analyze the use of a particular API to uncover flaws in that API. Another approach is to seek out general guidelines that can be applied to APIs. Stylos and Clarke (2007) used this approach to study the usability implications of constructor parameters versus setters in the general case. Most recently, researchers have sought out more effective methods for API usability analysis, since the lab study approach is extremely time consuming and resource intensive (Farooq and Zirkler 2010).

Each of these approaches holds value and has provided unique empirical understanding of API design and use. Yet, in large part, these approaches fail to deliver a richer theoretical understanding of the psychology of programming. Rather than explaining the rich social, improvisational, and psychological aspects of API use, they explain API designs, decisions, and their trade-offs.

Rather than focussing wholly on the artefact itself, the scientific exploration of API design and usability needs to reach beyond the artefact and explain the relationship between the social psychology of programming and the API, as well as the relationship between the computational sciences and the API. How does the API and information about the API shape, constrain, limit, enable, or undermine the psychological aspects of programming? How does the underlying technology (e.g., programming languages, paradigms, algorithms) shape, constrain, limit, enable, or undermine the API?

When thinking about API evaluation, one must be careful not to lose sight of the larger picture of reuse within the design process. We are not only designing an API that is easy to use, but also supporting the reuse of existing code in the larger design of a system. This is particularly challenging for a usability lab study, where programmers are studied working on small- to medium-sized singular tasks for which many API issues are set a priori. Of course, real programmers do not work alone and can often choose to reformulate their problem space, for example, rejecting one API for another. These decisions are not purely technical in nature. Cockburn argues that software development is in part an economic endeavour (2004). Indeed, one may choose to use the Google Web Toolkit (GWT) instead of the Dojo Framework not because it is better, but because it is perceived as being a more marketable experience to have on one's résumé. While we must not overly focus on such aspects of API design at the expense of usability, the social nature of software engineering is extremely salient.

1.2. Perceived Self-Efficacy

Perceived Self-Efficacy (PSE) is a construct from Social Cognitive Theory (Bandura 1997). It "is the belief in one's capabilities to organize and execute the courses of action required to manage prospective situations" (Zimmerman, 1995, p. 203). Examples of self-efficacy would be one's perceived ability to lift weights or one's perceived ability to pass a math test. PSE has value because it is found to correlate highly with actual task performance across a wide range of domains (Bandura, 1997).

The basic premise of PSE is that it is based on stable and grounded beliefs about one's capabilities. Over time, people develop and refine PSE with respect to the tasks in which they engage. For example, one may over time develop a PSE with respect to writing academic research papers. PSE is the stable and grounded construct that represents the overarching belief about one's ability to write such papers.

Bandura described a technique for developing psychological instruments exposing this underlying construct that has been utilized in many fields such as education and computer use (1997). Because PSE is highly correlated with performance, we believe that it may have value in the study of API design and usability. Unlike lab studies and peer reviews, PSE places small time requirements on the participants. In an API lab study, participants have to use the API. In a peer review, participants have to take the time to discuss an API with peers. PSE, however, is a psychological instrument that only requires the participant to reflect on an API and fill out a questionnaire containing Likert scale items. Further, PSE exposes a deep-rooted belief as opposed to focussing solely on the pragmatic outcomes. Thus, in addition to providing guidance, it also describes the impact of interventions at a psychological level.

Given that PSE is based on stable and grounded beliefs, after one is able to expose PSE for a given activity, the degree to which the construct is impacted becomes an important research question. For example, do you believe you can survive a bear attack in the woods? If you have a gun, do you believe you can survive the attack? What about if you have a bazooka, a gun, and some time to read a book on killing bears? Our perceived self-efficacy for accomplishing specific goals is moderated by the tools we have at our disposal. Certainly, in cases such as this toy example, our PSE is heavily impacted by the tools and environment. However, cases of PSE grounded in everyday activities are less clear. Does having a calculator impact PSE for passing a math test? Does having access to an IDE (vs. command-line) impact a programmer's PSE for debugging a problem in a large system? In these real-world cases, the degree to which PSE is impacted by tools is less clear. If we can expose programmers' PSE for API use, to what extent is the construct impacted by the tools and information at their disposal?

1.3. API Documentation

Given that no API can be perfect (or at least most), API documentation is critical to the success of projects (Bloch 2005, pg. 18). Indeed, more research into code documentation has been explicitly called for from within the software engineering community (e.g., Parnas 1994; 1998).

Existing work in API documentation focuses on documentation tools as opposed to information needs. Apatite and Jadeite are two recent examples of such tools. Apatite refocuses the structure of information around the notions of importance (i.e. – which parts of the API are most often used) and the relationship between API elements in the context of use (Eisenberg, Stylos, and Myers 2010). The interaction design in Apatite contributes to API documentation tools on two fronts. First, it shows the efficacy of using font size to indicate importance of API elements based on usage data. If you open the `java.io` package in Java, for example, `File` will be shown as an important class within that package. This helps programmers focus their attention on the most often used elements of the API. It also shows the utility of navigating by association. For example, if you open the `java.io` package, you find that `read` is an important method. By selecting that method, you find that `FileInputStream` is an important class with respect to that method.

Jadeite incorporates elements of Apatite, such as basing font size on usage, but adds in placeholder and object instantiation capabilities (Stylos, Faulring, Yang, and Myers 2009). Placeholders provide Java developers with a mechanism for communicating about API elements they expect to see as opposed to just what is there. It also scours existing source code to extract how classes are instantiated as objects. Thus, when a user uses Jadeite documentation for any given class, it tries to give you an example of how that class it created. The researchers found that developers performed three times faster with Jadeite than with traditional Java documentation. Calcite (Mooty, Faulring, Stylos, and Myers 2010) is an integrated development environment (IDE) plug-in related to Jadeite that pulls the object instantiation code directly into your code when you start to use an object (while Jadeite only puts that information in the documentation).

The approach used to evaluate the utility of Jadeite to show a significant performance improvement when using the tool is limited because it imposes significant artificial scaffolding for the programmer. Specifically, many API users have to begin with the selection of which API to use. For example, if you want to implement logging in Java, you can utilize Java logging, Log4J, or Commons Logging. One can also utilize Simple Logging Façade for Java (SLF4J) for dependency injection to separate the logging calls from the specific implementation. These are complex decisions that cannot simply be navigated away via documentation structure. However, these aspects of the problem-space are important notions when it comes to the formulation and development of a programmer's PSE.

1.4. Research Questions

To explore the role of PSE in API use, we focus on the following research questions. If we cannot operationalize the notion of a programmer's PSE for using an API, then the idea has little (if any) pragmatic value. Further, before we can make use of the construct, we need to develop a firm understanding of how the construct is impacted by the tools and information at hand. Note that in this paper we focus on the information at hand as opposed to the tools at hand.

RQ1: Can we identify, extract, and elucidate a professional programmer's PSE for using an API?

RQ2: To what extent does the information provided to professional programmers about an API impact their PSE for using that API?

2. Scale Development

Developing a perceived self-efficacy scale requires three steps (Bandura 1997). First, one must define the task, in particular the various aspects of the task. Having an analysis of the activity helps you identify the critical capabilities one would need in order to be able to do the activity. Second, one must construct statements about the task in the form of "you can do X". Third, one must incorporate obstacles to the task such as "even when you are tired". When presented with obstacles, questions become more salient, resulting in an accurate assessment of one's own capability. For example, if you ask someone if he can stick to a diet, even when he is tired, he will give a more accurate response than if you only ask him about his ability to stick to a diet.

To construct our scale, we iteratively built and refined an analysis of the core activities of API use, drawing upon Daughtry's work experience as a professional software engineer as well as extant literature on software design. We began with Schneiderman's programming task breakdown (1980). The tasks articulated by Schneiderman relevant to API use are learning, designing, composing (writing the code), comprehending, testing, and debugging. In order to use an API, programmers must learn how to use the API, design the system to use the API, write the code that calls the API, comprehend the code that uses the API, test the usage of the API, and debug the code that uses the API. Similarly, Fischer, Henninger, and Redmiles described reuse as being a cyclical process of location, comprehension, and creation (1991, p. 319). However, these task breakdowns fail to take into account the social side of programming activities.

The field of design rationale offers a particularly useful glimpse into the social process of design. From this perspective, design is a cyclical process of task analysis and artefact envisionment (Carroll and Rosson 2003). Namely, designers make claims about a design, reason around those claims, and then build the system. These claims may be implicit or explicit. The field of design rationale has sought to leverage explicit rationale; but with respect to our purpose, both forms of rationale are relevant. When designers go through the process of task analysis (whether it is grounded in formal task analysis methods or simply the thought process of what might work best), they are chiefly concerned with what Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel refer to as "goodness of fit" (1977). Specifically, does this design fit our problem, and how does it compare to other alternatives?

However, design is embedded in a socio-political process (Cockburn 2004). Thus, rationale must be defensible not only to rational team members, but also to irrational team members and time-constrained, budget-conscious managers. Further, one may make decisions based on self-interest as opposed to what is best for the system, other team members, or the customer.

Using these conceptualizations of design, along with reviews from professional programmers and other researchers, we developed the PSE instrument given in **Error! Reference source not found.**.

This instrument takes into account the design, implementation, testing, and debugging of API uses. At the same time, it also

No.	Scale Item
1	I can determine if this component is a good fit for use in a system design, even if the system requirements are not completely clear.
2	I can identify 5 or more well-suited applications for this component even if I were only given 1 minute to identify them.
3	I can recognize when this component has been wrongly chosen for use in a system design, even without knowledge of the entire system architecture.
4	I can identify an application where this component would fit (but be a poor fit) even if I had to identify such an application immediately.
5	I can defend my choice to use this component in my solution to other software developers, even if they are sceptical.
6	I can defend my choice to use this component in my solution to higher-level management, even if there is pressure to go with another component that already exists in the company product line.
7	I can make a strong case to a development team that the component should not be used, even if the team wants to use it because it is a hot technology (i.e., a buzzword that looks good on a résumé).
8	I can defend someone else's choice not to use this component in a solution to a technologically oriented customer, even if I had not discussed the decision with that developer.
9	I can use this component in building a system without having access to anyone who has used it before.
10	I can use this component in building a system without having access to any documentation other than the API (e.g., using no examples posted on the internet).
11	I can build a prototype system to demonstrate the power of the component to management in fewer than 30 minutes.
12	I can successfully augment an existing system I wrote to use this component instead of another mechanism even if it requires architectural refactoring in the system.
13	I can write coded tests against uses of this component that ensure the implementation will exceed performance needs of the entire system, even if I was extremely tired.
14	I can write coded tests against uses of this component that ensure that a system installation is configured correctly, even if the configuration is slightly different for each installation.
15	I can write system thread-level test procedures for a system that uses this component that validates every functional behavior of this component, even if a unit test procedure is not available.
16	I can write test plans for negatively testing behaviors of this component, even if the source code is not available.
17	Given a bug in the use of this component in a system I wrote, and knowing the effects of the bug, I could isolate the root cause of the problem even if the only things available were the source code and a running system. A debugger is not available, and there is no way to add new logging calls such as `System.out.println("test line 1: did it make it here?");`.
18	Given a known bug in the use of this component in a system, where the root cause has been isolated, I could fix the problem, even if I didn't completely understand the context of the component being used.
19	Having just used this component to build a larger system, I could identify most of the bugs caused by incorrect usage of this component before releasing the code to a testing group, even under extremely heavy scheduling constraints.
20	I could successfully fix a problem in the usage of this component within a larger system I didn't write, even if I had to make the fix on-site without access to a test group.

Table 1 - Perceived Self-Efficacy Scale for API Use

takes into account the social nature of programming. Because of space limitations, we do not include the Likert scales associated with each item. Each item has an associated Likert scale ranging from 1 to 9. Item 1 was labelled “Cannot do at all”; item 5 was labelled “Moderately certain can do”; and item 9 was labelled “Certain can do”.

Questions 1-8 cover the design activity (determining goodness of fit), taking into account the social aspects such as collaboration with team members, overcoming self-interest, and hierarchy. Items 9-12 deal with the use of the API within code. Item 9 takes into account the social component of implementation (having access to others who already know how to use the API). Item 12 takes into account the problem of real-world programming, whereby code is not always written from scratch. Items 13-16 deal with testing an API. This portion takes into account low-level unit testing through system-level testing, while at the same time taking into account the notion of propriety (source code may not be available) and the real-world hardships of testing (systems can be configured in many different ways). Items 17-20 address the debugging task, taking into account problems one is likely to encounter on real-world systems such as a limited schedule.

Each of the 20 items in our PSE scale includes obstacles, as previously discussed. When creating a PSE scale, it is important that the obstacles are realistic and challenging. Specifically, they should be something that one might actually encounter on the job and indeed pose a challenge for the programmer. After defining our scale, we conducted a sandbox pilot study where professional developers completed the scale in a think-aloud manner. The scale presented in **Error! Reference source not found.** is the final instrument, after revisions were made when issues surfaced during the sandbox pilot.

3. Code Vignettes and Claims Analysis

Having designed a scale for measuring a programmer’s PSE for using an API, we needed a mechanism for evaluating the validity of the scale (RQ1) and evaluating the impact information could have on the exposed construct (RQ2). The validation of a PSE scale involves the use of internal validity and factor analysis. We concluded that the same study could be used to address both research questions. However, we needed varying forms of API information in order to do this evaluation.

Scenarios and claims are the two fundamental components of scenario-based design (Carroll 2000). Scenarios are stories about people and their activities, while claims are an enumeration of the causal factors and relations that are left implicit in scenario narratives. With respect to programming, scenarios are reified as code examples with notes detailing the purpose of the example code. For the balance of this paper, we refer to such examples as code vignettes.

Example code and statements about the design are a natural representation when talking about code and software design. Indeed, one can find examples of this form in almost every practitioner-oriented discussion on API design (e.g., Bloch 2001; Bloch 2005; Cwalina and Abrams 2008; Tulach 2008; Pugh 2007) and other texts (e.g., Gamma, Helm, Johnson, and Vlissides 1994). Thus, we need to clarify what we mean by code vignettes and claims and distinguish it from other forms of example code and rationale. In each of the sources listed above, we find example code with lightweight design rationale expressed in the form of claims. However, these uses often differ in significant ways from what we mean by code vignettes and claims. With respect to the example code, they often show example *design* code as opposed to *uses* of that design. And, with respect to claims, they often describe the design *intention* of the design as opposed to describing the ramifications seen in a usage.

Figure 1 gives an example of design and intention information as opposed to what we mean by code vignettes and claims. First, let us consider the design against the code vignette. While the design conveys information about the interface, from which a programmer can derive the use, the code vignette is explicit in describing a use of the interface. This is beneficial because it affords a quicker translation to the activity of using the interface. Indeed, a programmer can copy-paste his way to using the interface with the code vignette. We know that programmers are opportunistic, debugging code into existence (e.g., Rosson and Carroll 1996; Brandt, Guo, Lewenstein, Dontcheva, and Klemmer 2009). Rather than using an API directly, they seek out uses of the API and adapt it to their needs. Turning to the justification for the design, the design example does not provide the salience the vignette example provides. Specifically, the analysis is loosely grounded on broad statements about the design as opposed to being lightweight analytical evidence. The design example can only make broad generalizations such as the interface being easy to code against while the vignette example has the grounding to make specific statements about why the interface is easy to code against. Within the publications mentioned above, and in other corpus (in print and online), code and rationale takes many forms that reflect vignettes and claims to varying degrees.

Design and Intention Information	**Code Vignette and Claims Information**
`public interface Collection {`	A programmer seeks to iterate over a collection.
`  …`	
`  public Iterable iterator();`	`for (Iterator i = c.iterator(); i.hasNext();)`
`  public boolean hasNext ();`	`  System.out.println(i.next());`
`  public Object next();`	
`}`	
This design is violates the bean custom of `get` and `set`. However, it makes for code that is easier to write and easier to read.	+ The programmer can express a loop on one line, even if the Collection name is a long expression.
	- The programmer cannot rely on the bean getter and setter convention when learning the API.

Figure 1 - A design and its intent vs. vignettes and claims

(Design and rationale adapted from Sun 1997)

Returning to Carroll's articulation of scenarios and claims, we find reasoning as to why object models and scenarios are particularly compatible (2000, p. 232). Known tasks are defined (at times explicitly) and translated to envisionments of use. For example, one may envision an API and its use via unit tests (if utilizing test-driven development). Alternatively, one may envision an API and its use via the Unified Modelling Language (if using model-driven development). This activity, and the associated reasoning via use (i.e., claims) helps to evoke, identify, and refine the API design.

We believe that code vignettes and claims are valuable information for programmers. As discussed above, vignettes support the opportunistic behaviour many programmers exhibit. Specific vignettes have been utilized in API documentation before. For example, some designers of API's have incorporated these into their API documentation. As discussed above, Jadeite and other tools have supported specific vignettes for object instantiation. Thus, it seems that the inclusion of vignettes should have significant value for programmers. We also believe that claims about an API use should have value to an API user by clarifying the designer's reasoning and thought process. For example, the designers of the Collections API for Java were compelled to document the rationale behind many decisions they made in the form of a design rationale document (Sun Microsystems 1997). Presumably, they went to such effort in order to fend off change requests, the questioning of their decisions by those who may not be able to infer the rationale from the design, and to aid other API designers by providing explicit reasoning about a particular API. Since all programmers are API designers and consumers, these are valid justifications for the documentation of rationale in all API designs. We do not know why the Sun designers chose to write a separate document instead of including the rationale in the API documentation itself. It makes it less accessible, locatable, and maintainable. Indeed, it was Sun that popularized the use of API documentation within source code via JavaDoc.

4. Methodology

Having contrasting forms of API information, we derive the following three hypotheses from our overarching research questions:

H1: Our new instrument effectively measures programmers' PSE with respect to API use.

H2: Inclusion of scenarios of use in an object-oriented API specification increases the user's PSE for using the API.

H3: Inclusion of claims in an object-oriented API specification can increase the user's PSE for using the API.

To test our hypotheses, we used the experimental design given in Table 2. To test H1, we needed a number of professional programmers to evaluate an API with our scale. To test H2, we needed a control group to evaluate an API with basic documentation (no vignettes or claims). We also needed a test group to evaluate an API with documentation that included vignettes. To test H3, we needed an additional test group to evaluate an API with documentation that included claims.

	Initial Data	**Assignment**	**Treatment**	**Observation**
Group 1	Reported number of years experience designing and developing software	Quasi-randomly assigned to groups based on experience	Presented API with basic documentation	Completed questionnaire (perceived self-efficacy)
Group 2			Presented API with basic documentation and the scenario	
Group 3			Presented API with basic documentation, the scenario, and claims	

Table 2 - Experimental Design

For subjects, we needed people who would know how to develop software and read an API. Therefore, we had two options for subjects: undergraduates or experienced professionals. While using undergraduates would allow us to obtain subjects more easily, we chose to use experienced professionals because we preferred ecological validity over power. Therefore, we chose a target of 20 subjects per group. To obtain experienced professional software developers as subjects, we leveraged existing professional contacts. In addition, we sought to keep the study as short as possible. With a limited and distributed sample pool from which to draw, we needed a very high participation and completion rate. The need for a short study was limiting in that we could not use multiple APIs.

Given that the potential subject pool would consist of primarily Java developers, we utilized JavaDoc as the documentation format. Figure 2 gives the top-level API documentation as given to group 3. We stripped the claims for group 2. We stripped the claims and scenario for group 1. One may assume from the depicted documentation that there was a substantial difference between the amount of documentation given to each group. Certainly, some groups were given more information than other groups. However, there was other basic information included in the API specification that is not included in the figure because of space limitations. The complete specification for each group (were it printed out from online) is 5.5 pages for the control group, 6 pages for the first test group, and 6.25 pages for the second test group.

Subjects were recruited via email and sessions were conducted over the web. They reported their experience and were assigned to a group using an algorithm that controlled for experience. Once assigned to a group, they were all given the same instructions. Specifically, they were told to imagine that they had just been placed on a development team tasked with building a large system. Their first task was to evaluate potential software components and frameworks for use within the system. They were then given the API and asked to complete the questions at the bottom of the page with respect to that API. There was no time limit imposed by the study. These questions were the PSE scale.

The Map class:

Understands a map as a collection of layers containing items. Layers can be deactivated to remove them from standard operations. However, those deactivated layers will still be used when adding two maps together. This map also understands the zenith and area of interest.

Scenario:

To create a map of the eastern United States, centered on Raleigh, NC.

```
      Map map = new Map("Raleigh, NC");
      map.setZenith(GeoLocater.getRaleighGeo());
      map.setAreaOfInterest(GeoRegionLocater.getEasternUSRegion());
      String EasternUSLayer = "Eastern US Layer";
      map.add(EasternUSLayer);
      map.add(JMapData.getEasternUSMapData(), EasternUSLayer);
      map.activateLayer(EasternUSLayer);
```

Claims:

+ Provides a single point-of-entry to handle maps.
+ Uses JGeolocation classes to simplify integration efforts.
+ Uses JMap classes to simplify integration efforts.
+ Uses map metaphors such as "layer" and "Zenith" because they have been extensibly used in other map software, both in API's and at the user interface level, so they should be familiar.
+ Allows for any persistence mechanism desired; does not specify persistence interface or implementation.
+ Allows for notification of changes to which layers are included and which are active, allowing clients to take action when the model changes (e.g., repainting GUI).
+ Allows client to combine Map objects easily without losing the previous Map.
+ Allows for multiple threads to work with an instance.
+ Provides lightweight mechanisms for communicating data over a network.
- Limited to using map data as defined in JMapData package.
- Requires that all Items be placed in a layer.
- Requires inclusion of JGeolocation and JMapDatapackages.
- Confounds layer runtime operations with the concept of Map: e.g, activateLayer(Layer).
- Does not provide persistence out-of-the-box.
- Combining maps is slowed down because of cloning operations.
- Cannot be serialized.
- [illegible]

Figure 2 - API Documentation

5. Results

We achieved our target of 20 subjects per group with a mean experience level of 8.6 years and a standard deviation of 5.8. We were able to control for experience successfully, with each group having participants ranging from 3 to 20 years experience as programmers and nearly identical means.

Every item on the scale had a variance greater than 1, indicating that there was some variance in the results. Factor analysis was performed on the 20 items. A scree plot (Figure 3) showed that the eigenvalues decreased more gradually after the first factor. Therefore, we ran a factor analysis for one factor. We found that one factor accounted for 37.4% of the variance. Adding more factors did not significantly change the explained variance; adding a second and third factor changed the explained total variance to 48.6% and 55.8%, respectively. In addition, removing any one item from the scale did not significantly change the variation explained by one factor, with the highest result being 38.8%

and the lowest being 36.5%. The Cronbach's alpha for the results was 0.91, which is high, and shows that the scale was internally consistent. Excluding items resulted in an alpha within the range of 0.90 – 0.91, so we kept all of the scale items.

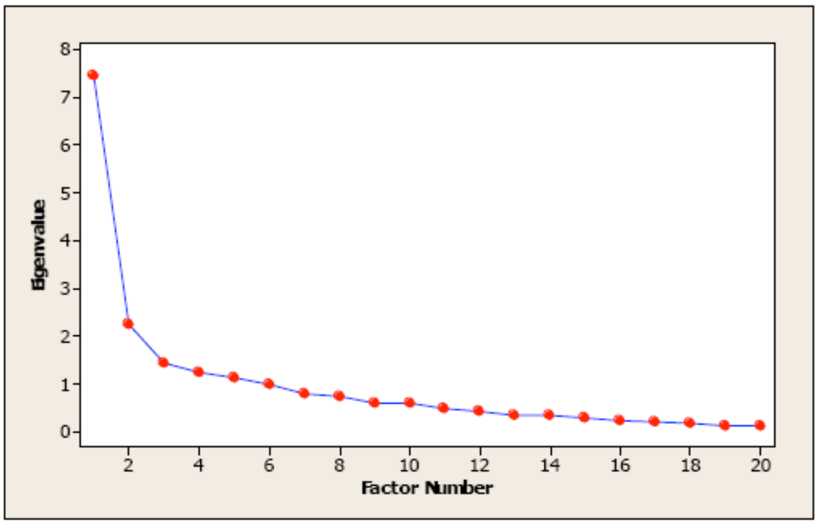


Figure 3 - Scree plot for the self-efficacy scale

We examined the probability plot of each item on the scale, and they all showed normal or nearly normal distribution. This also held for each group and there were no outliers.

We checked for equal variance and used a one-way ANOVA to compare the means of the three groups. Our F was 1.78 (which is a safe value for 3 treatments and 20 subjects). With a p-value of 0.320, there is not a statistically significant difference between the groups.

The results did not show significance, but we chose our sample size because of limitations in available subjects. Although student programmers are relatively easy to recruit, we would rather have participants that more accurately reflect the profession. This limited our sample size, so we did a power analysis. With three groups, an alpha of 0.95, a maximum difference between group means of 0.605 (between the basic documentation group and the scenario with claims group), and an overall standard deviation of 1.269, 113 subjects would be needed. This means that if the distributions we obtained in this study hold true, 113 subjects per group would be needed for the results to be statistically significant.

Finally, as data exploration, we examined the relationship between experience and self-efficacy. We computed Pearson's product-moment correlation coefficient for the entire data set and within each group. We found no evidence of a significant correlation between experience and PSE. In each group, either the correlation was too weak to be significant (e.g., -0.026 for the control group) or the correlation was not statistically significant (e.g., 0.128 for the vignettes group).

6. Discussion

In this work we had two research questions. First, we sought to develop an instrument for exposing programmers' PSE for using an API. Notably, in exposing this construct, we incorporated the social dynamics of API use within real-world systems. Second, we sought to evaluate the extent to which information (we used code vignettes and claims) impacts the exposed construct.

We found that the PSE scale we developed was able to capture programmers' PSE effectively for using an API. The very significant Cronbach's alpha (0.91) shows that the scale is internally consistent. One factor accounts for 37.4% of the variance between participants, and no other factor is significant. Thus, the construct being measured is a simple and unitary construct.

Intuitively, one might expect that over time, programmers have a higher PSE for API use. However, we saw that there was not a statistically significant correlation between self-efficacy and experience.

In the strongest case, we are only 87.2% sure there is a correlation coefficient of -0.352 between years experience and self-efficacy when scenarios are in the documentation. However, this is a weak case for a marginally weak correlation. If the scale was more dependent on experience than the API design itself, then we would have to conclude that the scale measured a programmer's general PSE for API use as opposed to measuring the programmer's PSE for a particular API. However, if we make the assumption that a programmer's PSE for API use in general goes up over time, then we must conclude that our scale is indeed measuring PSE for the API in question.

Looking only at the means of the groups, the results suggest (though insignificantly) that code vignettes may actually decrease the self-efficacy of the subjects and that claims did so to an even greater degree. Our initial presumption was that vignettes and claims would increase the usability of the API. However, upon reflection, the real target outcome of vignettes and claims is that the programmers can make more accurate judgements about the API. Thus, it is possible that our API had poor design elements, and the programmers were able to assess the API more accurately based on the additional information.

We found that we could not significantly alter the PSE results with significant documentation changes. Thus, we also know that the PSE being measured is fairly robust. Unfortunately, this robustness also means that we cannot use PSE as a way to evaluate small design decisions such as the inclusion of vignettes and claims in documentation.

Given that the exposed construct is robust, it may hold value with respect to endeavours that require a more robust construct. First, API usage performance in general may be related to our construct. Evaluating programmer performance is a long-sought goal in the study of programming. Unfortunately, a PSE instrument is easily manipulated if programmers are aware that they are being evaluated against their responses. However, if a relationship does exist, it would still provide a significant contribution to the psychology of programming. Further, its utility with respect to evaluating the development of novice over time may hold value. Although experience was not correlated with PSE in our study, we focussed on professional programmers.

Second, since it is not impacted by small design decisions, the PSE construct might be useful at a higher granularity with respect to technology. For example, could it have shown the Java platform design team that their Calendar API needed significantly more work, while the Collections API satisficed? At present, the only mechanism we have for making such decisions is a subjective analysis of importance with respect to how often the API will be used (Stylos and Myers 2007).

We only examined the relationship between PSE and the information given a programmer about an API. We did not examine the relationship between changes in an API itself and PSE or changes in the tools leveraged in API use (e.g.. Integrated Development Environments). It is possible that these may still significantly impact PSE.

We studied professional programmers who, over time, have developed expectations of API documentation. Some or all may have developed distrust for documentation, thus ignoring everything except the structure of the API (e.g.. class name, method names, and method return types). This would have led to scenarios and claims having no impact. Even if they did trust the documentation, the inclusion of the scenarios and claims may have thrown them off. This is information they are not used to seeing in documentation. Second, engineers are presumably extremely analytic. They may have over-analyzed the questions presented in the self-efficacy scale in ways that did not surface in the sandbox pilot. It is also possible that some APIs would benefit from scenarios and claims, while others do not. This study only examined one API due to constraints in study length. The API selected may have not been an API that benefits from scenarios or claims. Finally, the scenarios and claims included in the API could have been different. Perhaps we did not construct scenarios and claims that significantly contributed to the use of the artefact. Again, we were constrained due to study length.

7. Conclusion

Our analysis supported H1. Interestingly, we had no support for H2 and H3. But, our results do translate to the following significant results:

> **Result 1:** Our perceived self-efficacy scale appears to measure a single construct, which we presume to be the perceived self-efficacy of a developer in using an API.
>
> **Result 2:** Programmer's PSE for using an API is robust, seemingly unaffected by changes to documentation.
>
> **Result 3:** Experience does not play a significant role in developers' perceived self-efficacy for using APIs.

Given the robustness of the scale developed, several research opportunities arise from this work. First, the scale needs to be applied against other APIs so that we can assess whether it has utility in answering other important questions about programming. For example, one could compare the results for a notoriously poor API design (e.g., something like the Java Calendar API) and a good API design (e.g., something like the Java Collections API) to evaluate the effect that different APIs have on a programmer's PSE. Second, given the robustness of the scale, similar scales could be developed that are at a lower level. Indeed, Bandura warns that the granularity of PSE scales can heavily impact their utility. Specifically, an API use debugging PSE scale would be much more sensitive to small design changes than the broad API usability PSE scale we developed.

Finally, and perhaps most importantly, we need to develop a more thorough understanding of the role documentation plays in programming. We did not include such a test in our experiment, but it is safe to assume that had we included an API with no documentation (to include meaningful package, module, and member names), it would have received a very low score by programmers. How can one even begin to use such an API, much less ascertain things like its goodness of fit and testability? Although we have all been told (and we tell our students) that documentation is important and names should have meaning, we don't have an understanding of how these API elements support the information needs of API users.

Currently, we are seeking to develop a better understanding of the API design decision space described by Stylos and Myers (2007). In order to explore the relationship between psychological constructs and API design decisions effectively, we must be able to understand the dimensions of API design and use with respect to particular API features.

8. Acknowledgements

We would like to thank Mary Beth Rosson, Steve Haynes, Umer Farooq, Matt Peters, and Josh Gross for their advice and support in the design and execution of this work. John Carroll would also like to thank Edward M. Frymoyer for his financial support. We would also like to thank Dwight Berry for his comments on a prior draft.

9. References

Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., and Angel S. (1977). *A Pattern Language*. New York, NY: Oxford University Press.

Bandura, A. (1997). *Self-efficacy: The exercise of control*. W.H. Freeman and Company.

Bloch, J. (2001). *Effective Java*. Second Edition. Upper Saddle River, NJ: Addison-Wesley.

Bloch, J. (2005). How to Design a Good API and Why it Matters. keynote in Library-Centric Software Design. Retrieved July 1, 2006 from Library-Centric Software Design Web Site: http://lcsd05.cs.tamu.edu/slides/keynote.pdf.

Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. (2009). Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, 26(5), pp. 18-24.

Carroll, J. M. (2000). *Making Use: Scenario-Based Design of Human-Computer Interaction*. Cambridge, MA: MIT Press.

Carroll, J. M. and Rosson, M. B. (2003). Design Rationale as Theory, in J. M. Carroll (Ed.) *HCI Models, Theories, and Frameworks* (pp. 431-461), San Fransisco, CA: Morgan Kaufmann.

Clarke 2004. Measuring API Usability. *Dr. Dobb's Journal*. May 2004. pp.

Cockburn, A. (2004). *The End of Software Engineering and The Start of Economic-Cooperative Gaming*. Retrieved January 28, 2006, from Alistair Cockburn Web Site: http://alaistair.cockburn.us/crystal/articles/teoseatsoecg/theendofsoftwareengineering.htm.

Cwalina, K. and Abrams, B. (2005). *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Upper Saddle River, NJ: Addison-Wesley.

de Souza, C.R.B., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. (2004). Sometimes You Need to See Through Walls – A Field Study of Application Programming Interfaces. *Proceedings of the 2004 ACM Conference on Computer Supported Collaborative Work (CSCW 2004)*. Chicago, IL: ACM Press, pp. 63-71.

Eisenberg, D.S., Stylos, J., Myers, B.A. (2010). Apatite: A New Interface for Exploring APIs. *Proceedings of the International Conference of Human Factors in Computing Systems (CHI 2010)*. Atlanta, GA: ACM Press. To appear.

Farooq, U. and Zirkler, D. (2010). API Peer Reviews: A method for evaluating usability of Application Programming Interfaces. *Proceedings of the 2010 ACM Conference on Computer Supported Collaborative Work (CSCW 2010)*. Savannah, GA: ACM Press, pp. 207-210.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley.

Fischer, G., Henninger, S. and Redmiles, D. (1991). Cognitive Tools for Locating and Comprehending Software Objects for Reuse. *Proceedings of 13th International Conference on Software Engineering (ICSE'91)*. Austin, TX: IEEE Computer Society, pp. 318–328.

Mooty, M., Faulring, A., Stylos, J. and Myers, B.A. (2010). Calcite: Completing Code Completion for Constructors using Crowds. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2010)*. Madrid, Spain: IEEE Computer Society, to appear.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*. 15(12), 1972, pp. 1053 – 1058.

Parnas, D.L. (1994). Software Aging. *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 279-287.

Parnas, D.L. (1998). Successful Software Engineering Research. *ACM SIGSOFT Software Engineering Notes*, 23(3), pp. 64-68.

Rosson, M.B. and Carroll, J.M. (1996). The Reuse of Uses in Smalltalk Programming. *ACM Transactions on Computer-Human Interaction*, 3(3), 219-253.

Scheiderman, B. (1980). *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop Publishers.

Software Engineering Institute (2008). Carnegie Mellon Software Engineering Institute Software Technology Roadmap: Application Programming Interface. Retrieved December 15, 2008 from: http://www.sei.cmu.edu/str/str.pdf. SEI 2008.

Stylos, J. (2009). *Making APIs More Usable with Improved API Designs, Documentation, and Tools*. Doctoral Dissertation. Pittsburgh, PA: Carnegie Mellon University.

Stylos, J. and Clarke, S. (2007). Usability Implications of Requiring Parameters in Objects' Constructors. *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007).* ACM Press, pp. 529-539.

Stylos, J., Faulring, A., Yang, Z., Myers, B.A. (2009). Improving API Documentation Using API Usage Information. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2009)*. Corvallis, OR: IEEE Computer Society, pp. 119-126.

Stylos, J. and Myers, B.A. (2007). Mapping the Space of API Design Decisions. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2007).* Coeur d'Alène, ID: IEEE Press, pp. 50-57.

Sun Microsystems (1997). *Java Collections API Design FAQ*. Retrieved March 2, 2010 from the Java 1.4 documentation: http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html.

Zimmerman, B. J. (1995). Self-Efficacy and Educational Development, in A. Bandura (Ed.) *Self-Efficacy in Changing Societies* (pp. 202-231). Cambridge, MA: Cambridge University Press.

Enhancing Comprehension by Using Random Access Memory (RAM) Diagrams in Teaching Programming: Class Experiment

Leonard J. Mselle

School of Informatics
The University of Dodoma
mselel@yahoo.com

Keywords: RAM diagrams, teaching programming, novice programmers, close tracking of code, programming comprehension, programming skills.

Abstract

This paper presents results of an experiment in which Random Access Memory (RAM) diagrams were used to teach novice students C programming. Students were divided into two groups that were differently instructed. The control group was instructed in the traditional way while the experiment group was instructed with the aid of RAM diagrams employed throughout the course. Examination results from the two groups were compared. Statistical analysis was done and the *Z* value was calculated. The results suggest that the use of RAM diagrams improves programming comprehension and programming skills.

1. Introduction

A substantial number of researchers conclude that mastering programming is difficult for majority of students. Dehnadi and Bornat (2006) report that a substantial majority of students fail in every introductory programming course in every UK university. They argue that despite a great deal of research into teaching methods and student responses, the cause are not yet established.

Yousoof et al. (2007) contend that the source of difficult in programming can be attributed to cognitive overload. Cognitive overload happens in programming due to the nature of the subject which is intrinsically over-bearing on the working memory. It happens due to the complexity of the subject itself. They conclude that the problem is made worse by the poor instructional design methodology used in teaching and learning programming. Regarding cognitive load, Ala-Mutka (2003) points out that in programming, students are required to contend simultaneously with a number of issues. These include syntax, semantics, algorithm design, problem solving and paradigm specifics. Marcia (1992), Tudoreanu (2003), Du Bolay et al. (1986), Vainio (2007) are among those who assert that mastering programming is not easy.

1.1 Research on different approaches in teaching programming

Research works on alternative approaches to teaching programming include studies on the use of different methods to conduct lessons. In this direction, alternatives such as improving effectiveness of lectures combined with discussion groups, problem solving approaches, watching examples of running codes, predicting what happens next and learning by doing have been tried. Other alternatives are the use of graphics and graphical metaphors in program visualization. Another focus has been on new concepts such as roles of variables (Kuittinen and Sajaniemi 2003). So far, none of these studies have claimed to have entirely solved the problem.

This research is aimed at introducing and testing the effectiveness of a new tool which combines in one single object; the computer memory (RAM), together with the syntax, semantics, and the variables. This tool is a modification of trace tables which were used to teach programming in early days. Hoc (1989) observes that the lack of "Representation and Processing System" (RPS) closely related to the computer operations constitutes an obstacle for novice programmers in learning. RAM diagrams are designed to function as RPS.

There is evidence to support the notion that carefully designed tools can aid programming students to develop accurate mental models and hence facilitate their comprehension of programming (Scott et al. 2007). RAM diagrams as a pedagogical tool are designed to address these issues.

To demonstrate and test the effectiveness of RAM diagrams, this paper discusses program visualization in section 2. The concept of trace tables is revisited in section 2.1. RAM diagrams are introduced in section 2.2. They are demonstrated in section 2.3. Advantages of RAM diagrams are discussed in section 2.4. Their ability to foster problem-solving skills is discussed in section 2.5. Experiment about effectiveness of RAM diagrams is discussed in section 3. Results are discussed in section 4 and conclusions are presented in section 5.

2. Program visualization

Program visualization is an approach to teach programming by showing (animating) the code, line by line while vividly reflecting results of its execution. Use of visual representations is not new in programming. Flowcharts have traditionally been used to visualize program structures (Scott, et al. 2005). Napes et al. predict that visualizing the execution of programs and showing a full life cycle of objects would probably help students (Kuittinen et al. 2008).

Research on development, use and effectiveness of animation tools have been covered extensively by Ben-Ari et al. (2001), and Sajaniemi et al. (2003) among others. Online visualization tools such as Jeliot 2000, PlanAni and BlueJ are among the current program animators. Research on program visualization is currently an area of great interest (Ben Bassat 2001), (Scott et al. 2005), (Stutzle and Sajaniemi 2005).

2.1. Trace tables

Trace tables are among the first tools that were employed to teach programming. As far back as 1960s they were regarded as useful tools in teaching programming to beginners. Trace tables were being used for debugging and teaching programming when Pascal and Fortran were the teaching languages (Tailor 1977). Their mechanism is as shown in figure 1.

```
int x = 0;
int i;
while ( i<5{
x = x + i;
i++;
}
```

I	X
1	1
2	3
3	6
4	10

Figure 1- Example of Trace Table

For reasons not yet clear, trace tables do not feature in modern programming books, teaching notes or syllabuses. No apparent reason is given for this abandonment. In this research, a survey carried out on 56 programming books at four universities in Tanzania and Rwanda found that there was no single title that had made reference to trace tables. Trace tables are simple paper and pencil variable-tracing tools. However, they do not explicitly include RAM in their mechanism.

2.2. RAM Diagrams

RAM diagrams, as a tool for teaching programming, are a modification of trace tables. Like trace tables they focus intensely on variables and their values. In addition, RAM diagrams display the code while mimicking RAM in relationship with the change in the behaviour of variables along with the algorithm. RAM diagrams can be used by the instructor to simultaneously reflect the syntax, semantics and the algorithm. This simplifies the task to describe the internal and logical aspects of programming such as variable declaration, data feeding, sequence, bifurcation, iteration, parameter passing and file handling. The first part of a RAM diagram is the header/footer, stating the aspect of programming that is being demonstrated i.e. ***variable declaration, data feeding, selection, iteration, etc***. The second part is the RAM-image, which is represented by an array of cells (rectangles). This gives them ability to mimic the computer RAM in association with variables and their behaviour when the code executes. The third part is the piece of code that is being discussed. This gives them ability to represent the syntax and the algorithm of the problem being solved or described.

Du Boulay et al. (1989) argue that there are two approaches in teaching programming. The first approach is called *black box approach*. Under black box approach, the mechanisms by which the computer operates are hidden from the user. The second approach is *called glass box approach*. In this approach, the user attempts to understand what is going on inside the computer. Each command results in some change in the computer and these changes can be described and understood. Users do not need to become electronic experts. There is an appropriate level that Mayer (1986) refers to as the "transaction level". As a pedagogical tool, RAM diagrams are designed to enable the instructor and the learner to pursue the *glass box approach* in teaching and learning programming. They provide possibility for close tracking with absolute precision.

2.3. Demonstration of RAM diagrams

Consider the following code:

```
/*Program 1*/
 main()
 {
int x;
int y;
x=4;
y=7;
x=x+y;
}
```

Using RAM diagrams, *program 1* can be close-tracked as shown in figure 2.

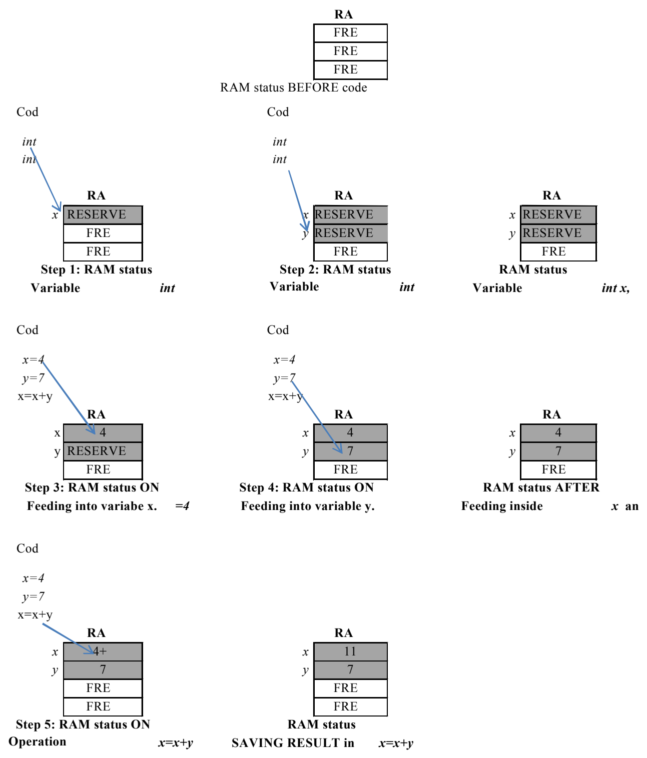


Figure 2 - Example of RAM

2.4. Advantages of RAM diagrams

Du Boulay et al. (1989) offer two important properties for making hidden operations of a language clearer to novice: (1) *simplicity*- there should be “a small number of parts that interact in ways that can be easily understood”; (2) *visibility*- novices should be able to view selected parts and processes” of the model “in action”. RAM diagrams have been designed in consideration of the following properties:

1. In order to employ or understand RAM diagrams, students are not required to learn any new concept. For example, to use flow charts, novices are required to understand symbols, their connectivity, and their correspondence with the logic of the code.

2. RAM diagrams portray a direct relationship between code-statement and its effect on the memory (RAM) and the variables. Flow charts and trace tables are devoid of code-RAM relationship.

3. Roles of variables as discussed by Sajaniemi et al. (2003) can explicitly be expressed by RAM diagrams. With RAM diagrams a novice is able to see a gatherer gathering, a stepper stepping, a fixed-value fixed, a follower will be seen following, etc.

4. RAM diagrams are not machine dependent. They are applicable to any programming language. They can be employed within or outside computer environment. This gives them advantage of portability, flexibility and scalability. A programmer does not need to be tied to a specific computer environment (as may be necessary for simulators) to check the precision of the code. With pencil and paper the novice is able to verify the precision of the code segment.

5. RAM diagrams can serve as a code designing and code testing tool. They can be used as code studying tool and debugger. Novice programmers can use RAM diagrams to study other people's programs or their own programs to figure out why an instruction appears, where it appears.

6. By explicitly linking code-writing to variables, RAM diagrams can facilitate schema formation due to their ability to associate machine-memory and code development. According to Ala-Mutka (2003), schema formation is one of the most challenging objectives of training novice programmers. Associating programming practice to computer memory at an early stage can help the novice to fathom the reality that the machine is just a passive recipient of programmers' brain-work. This in turn can enhance ability to design algorithm and schema formation.

7. RAM diagrams can provide an alternative in the way teaching materials are presented. Programming books and notes can incorporate RAM diagrams to provide students with a tool to study and design codes.

2.3. Problem solving and use of RAM diagrams

Johnson-Laird (1959) defined mental models as a way of describing the process which humans go through to solve deductive reasoning problems. His theory included the use of a set of diagrams to describe the various combinations of premises and possible conclusions (Haden and Mann 2003). Numerous studies have concluded that most programming students are overwhelmed by the hurdle to build problem solving skills and proper mental models that will enable them to design and write programs. Scott et al. (2005) contend that for many novice programmers a key weakness lies in their problem solving skills. Many novices engage in program development without possessing an appropriate model of an algorithmic solution. The same views are expressed by Garner and Howell (Stutzle 2005).

Using diagrams to explain and describe phenomena, has been employed in different disciplines to enhance comprehension and building mental models. RAM diagrams, have the ability to reflect the image of variables in the computer memory. Like a map for a navigator, they guide the novice towards the solution in an incremental manner. Using RAM diagrams to visualize the effect of code statements on computer RAM, will enable the programmer to achieve the following:

- Organize ideas in a common theme
- Capture what is going right or wrong in the code
- Understand how the process works
- Understand the way factors (statements and variables) affect one another.

These, in turn, happen to be problem solving steps which, if mastered by novice programmers at the early stage, their abilities to devise algorithms and develop mental models could be largely enhanced.

Pekins et al. (1986) categorize novices programmers into movers, stoppers and tinkerers. Using RAM diagrams, movers can rectify their codes as they move on. Stoppers can use RAM diagrams to chart out new direction. Tinkerers can use them to verify their bearing. For all categories, RAM diagrams constitute an ideal tool for close tracking code (Soloway and Spohrer 1989).

3. Experiment

To test hypothesis that, ***consistent use of RAM diagrams in teaching programming will increase student's ability to devise algorithms and write codes***, a class experiment that included two groups of students, (n=100) was carried out. The groups consisted of first year students who were pursuing introductory programming course at Kigali Institute of Science and Technology (KIST). Both groups comprised novice students who were being taught Programming in C. The syllabus and teaching hours were equal for both groups.

3.1. Method

The control group, comprising 39 students, was instructed by a lecturer who followed the traditional approach. Lectures were allocated 30 hours while laboratory and tutorials were allocated 45 hours. The experiment group, consisting of 61 students was instructed by a lecturer who employed RAM diagrams. During lectures and tutorials the instructor of experiment group, consistently employed RAM diagrams to explain the code. Equally, during laboratory sessions, students were encouraged to close track their codes using RAM diagrams. At the end of the course both groups attended the same final university examination.

3.2. Subjects

The subjects were undergraduate first-year students studying Introduction to Computer Programming: C Language. The experiment group consisted of students taking food science major. The control group comprised students studying computer science. None of the students had prior exposure to programming. Participants were not made aware of the experiment.

3.3. Materials

The examination consisted of eight questions. Each question carried a total of twenty marks. All questions involved parts of coding and problem solving. The examination was designed to ensure that aspects of sequence, selection, iteration, functions and file handling are well covered among the eight questions from which students were required to select five. The examination and marking scheme were jointly written by a panel of examiners. For the purpose of this experiment, the lecturer for the experiment group decided to exclude himself from setting the examination. However he was present in setting the marking scheme. Scores were distributed to provide the measure of ability to devise algorithms and convert such algorithms in syntactically correct codes. Among questions that featured were: (a) Given the equation; $y=x^2$, write a code to solve it (4 Marks). (b) Write a code that will solve $y=8+2x^2$ (6 Marks). (c) Given the following scores, {60, 45, 34.5, 67, 45, 60}, write a code that will store them in an array and calculate their total. (5 Marks) (d) Using a *while loop*, write a program to store names and addresses of five students in a file (5 Marks). Questions were broken down into at least 4 parts. The total score for any of 5 questions was 100 marks.

3.4. Procedure

The examination duration was three hours. Answer scripts were handed to the invigilator who latter handed them to examinations department. From there, they were collected for grading by the lecturer of control group who was not aware of the experiment. Grading for each question was carried out using a common marking scheme which, for each question was framed based on the following criteria:

(1) Ability to understand the question and evolve a correct algorithm (50% of the marks)

(2) Ability to write a syntactically correct code corresponding to the algorithm (50% of the marks)

After marking, results were handed to the examination department for recording. The researcher collected the results from examinations department for analysis. Assuming that the scores would reflect comprehension of programming and better programming ability, statistical analysis of scores of all, (n=100) students was carried out.

The null hypothesis is therefore stated as Ho: There is no difference in performance between the groups.

The alternative hypothesis is stated as Ha: Performance of the experiment group will be better than that of the control group.

4. Results and discussion

Results of the experiment are summarized in Table 1.

Experiment Group N = 61		Control Group N = 39	
Mean	SD	Mean	SD
64.67	9.58	60.02	11.98

Table 1 - Results obtained from the experiment

The average score for students in the experiment group was 64.67% while that of control group was 60.02%. The standard deviation was 9.58 for the experiment group while that of control group was 11.98.

A two-tailed statistic test at 95% level of confidence was worked out thus $|Z|>1.96$ yielding the value of 2.01. This is a substantial statistical difference which, allows the null hypothesis to be rejected at 0.05 level of significance.

Samurcay (1986) asserts that programming is not only about producing a solution, but also to make explicit the procedure producing the solution. Teaching programming while consistently using RAM diagrams to understand codes, enables the teacher to make explicit the procedure while pursuing the solution (Soloway and Spohrer 19890).

Pekins et al. (1986) posit that close tracking of program is an essential skill in programming. It helps novice to find out bugs. RAM diagrams qualify to be an easy tool for close tracking. RAM diagrams provide the novice with a handy tool to understand programming primitives. Understanding of primitives is the foundation for mastery of more complex issues. As demonstrated, RAM diagrams provide a means for stoppers to reason why the machine is behaving as it is behaving. Tinkerers may use it as a clear guide to proceed (Soloway and Spohrer 1989).

5. Conclusion

As demonstrated in section 2.2 and 2.3, the strength of RAM diagrams as a visual tool, emanates from their simplicity and ability to bundle in one unit the memory (RAM), variables, the code (syntax) and the flow of control of a program. Used effectively, they strengthen the sense of programmer being at the centre of programming as opposed to the feeling that the computer is responsible for anything going wrong about the code. They provide an off-line verification tool.

However, there are questionable issues associated with this experiment. While there is a significant difference in the examination scores of the two groups, this could be explained by at least two other reasons: one, it could be that students in the experimental group were smarter or better able to learn programming. Second, it could be that the teacher of the experimental group was just a better teacher irrespective of the RAM diagrams. To address these issues, more class experiments are being carried out in different settings, to determine the effectiveness of the tool.

6. References

Ala-Mutka, K. (2003) Codewitz, Needs Analysis. [On line]. Available: http://www.cs.tut.fi/~edge/literature_study.pdf.

Ben-Ari, M. and Sajaniemi, J. (2003) Roles of variables from the perspective of computer science educators. [Online]. Available : http://cs.joensuu.fi/pub/Reports/A-2003-6.pdf

Ben Bassat, L. R. et al. (2001) An extended experiment with Jeliot 2000. Proc. First International Program Visualization Workshop, University of Joensuu Press, Pavoo Finland, 131-140.

Kann, C. et al. (1997) Integrating Algorithm Animation into a Learning environment. Computers and Education, 28(4), Elsevier Science Ltd. Oxford, 223-228.

Dehnadi, S. (2006) Testing Programming Aptitude. P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 18.

Dehnadi, S. and Bornat, R. (2006) The camel has two humps (working title). School of Computing, Middlesex University, UK.

Haden, P. and Mann, S. (2003) The Trouble with Teaching programming. Proc. of the 16th. Annual NACCQ, Palmeston North, New Zealand.

Kuittinen, M. et al. (2008) A study of the development of students' visualizations of program state during an elementary object-oriented programming course. ACM Journal of Educational Resources in Computing, 7(4).

Kuittinen, M. and Sajaniemi, J. (2003) First Results of An Experiment on Using Roles of Variables in Teaching. M. Petre & D. Budgen (Eds) in Proc. Joint Conf. EASE & PPIG, 347-357.

Leslie, J. And Waguespack, Jr., (1989) Visual metaphors for teaching programming concepts. ACM SIGCSE Bulletin, 21(1), 141-145.

Lim, M. and Michael, C. (1992) The case for case studies for programming problems. Communication of the ACM, 35 (3), 120-122.

Sajaniemi, J. and Hu, C. (2005) Teaching programming: Going beyond “objects first”. [On line]. Available: http://www.ppig.org/papers/18th-sajaniemi.pdf

Scott, A. et al. (2005) A Step back from Coding – An Online Environment and Pedagogy for Novice Programmers. [On line]. Available: http://www.ics.heacademy.ac.uk/events/jicc11/scott.pdf.

Soloway, J. and Spohrer, C. Studying the Novice Programmer. Laurence Erlbaum Associates: Hillsdale, New Jersey, 1989.

Stutzle, T. and Sajaniemi, J. (2005) An empirical evaluation of visual metaphors in the animation of roles of variables. [On line]. Available: http://inform.nu/Articles/Vol8/v8p087-100stut.pdf.

Tailor, R.T. (1977) Teaching Programming to Beginners. ACM SGCSE Bulletin, 9(1), 1977.

Tudoreanu, M. (2003) Designing Effective Program visualization tools for reducing users cognitive effort. Proceeding of 2003 ACM Symposium on software Visualization, ACM Press, San Diego, California.

Vainio,V. and Sajaniemi, J. (2007) Factors in novice programmers' poor tracing skills. *ITiCSE 2007*, 236-244.

Yousoof, M. et al. (2007) Measuring Cognitive Load - A Solution to Ease Learning of Programming. Proc. Of World Academy of Science Engineering and Technology, 20.

Evaluating Scratch to introduce younger schoolchildren to programming

Amanda Wilson and David C. Moffat

School of Engineering and Computing,
Glasgow Caledonian University,
Glasgow, Scotland, UK
D.C.Moffat@gcu.ac.uk

Abstract. The Scratch system was designed to enable computing novices, without much programming experience, to develop their creativity, make multimedia products, and share them with their friends and on a social media website.
It can also be used to introduce programming to novices. In this initial study, we used Scratch to teach some elementary programming to young children (eight years old) in their ICT class, for eight lessons in all. Data were recorded to measure any cognitive progress of the pupils, and any affective impact that the lessons had on them.
The children were soon able to write elementary programs, and moreover evidently had a lot of fun doing so. An interview with their teacher showed that some of the pupils did surprisingly well, beyond all expectations. While the cognitive progress is moderate, the main advantage to Scratch in this study seems to be that its enjoyability makes learning how to program a positive experience, contrary to the frustration and anxiety that so often seems to characterise the usual learning experience.

Keywords: POP-I.A. learning to program; POP-I.B. choice of methodology; POP-II.A. novices, schoolchildren; POP-III.B. smalltalk; POP-III.C. visual languages; POP-III.D. visualisation; POP-IV.A. exploratory; POP-V.B. case study; POP-VI.E.

1 Introduction

Computing technology is increasingly important in the modern world, which could not function without it. One might expect greater numbers of students to want to learn about computing; but numbers of students at school and university are falling in the industrialised world.

The situation in the UK, for instance, is approaching crisis point, as recently documented by the UK's Computing Research Committee. According to their report (UKCRC, 2010), the numbers of school pupils taking Computing or ICT (Information and Communication Technologies) courses has "collapsed" by about a third in less than five years; and the consequences for university intake have been severe.

One of the major problems identified is that Computing in schools is typically confused with ICT, and pupils are taught basic skills in office applications like word-processing and spreadsheets. Their teachers themselves often have no formal education in computing, and cannot communicate enthusiasm or understanding about what happens inside a computer to make it work. In particular, there is little introduction to programming in some schools, and what there is can easily lead to intimidation of the pupils rather than enlightenment. As a result, they may leave school feeling that programming is mysterious and difficult, or frustrating and boring. It is no wonder then, if they choose not to pursue computing at university and in the workplace.

The problem may be tackled by making introductory programming both easier and more fun, and there are several attempts to achieve this. The Scratch[1] system from MIT (Resnick et

[1] Home website: http://www.scratch.mit.edu/

al., 2009) is a simplified visual programming system in which it is relatively easy to manipulate multimedia objects, without much preparation. It is a leading candidate to help introduce children to programming, and the subject of the present study.

Scratch has been used by some enthusiastic teachers in schools in the USA and the UK for extra-curricula activities (like after-school clubs), and anecdotally they are pleased with the experience. It has been used for introductory programming at some universities, which have gone so far as to publish evaluations of it; but there is little evaluation to date of its use with the intended age group of middle school pupils.

In the present study, we made an initial evaluation of Scratch for school pupils, where we deployed it in their IT lessons for eight weeks.

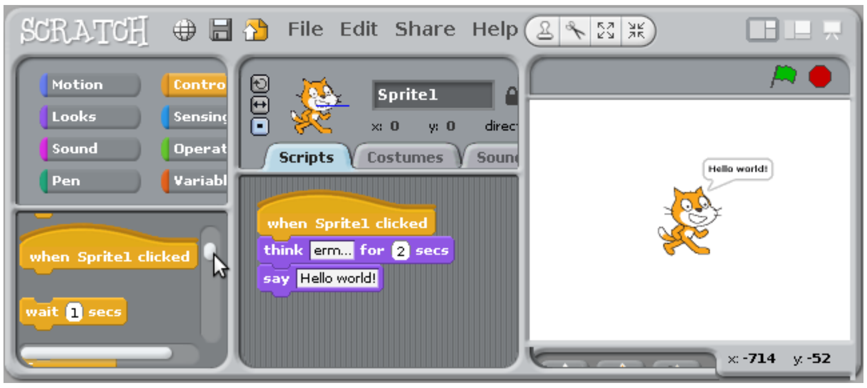
1.1 What is Scratch?

Originally inspired by Papert's work (Papert, 1980), Scratch was intended by Resnick to support creative work with multimedia (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) in "computer clubhouses" or after-school learning centres for children from deprived communities, and was first deployed in 2005. Those children enjoyed multimedia "mash-ups," like the sampling techniques used in the pop music they liked, which is why the new system came to be called "Scratch."

The focus of Scratch is on making multimedia products, and sharing them in the large and active online community hosted by the project website. This is intended to enable and develop children's creativity, but also to introduce them to programming, in a fun way.

Visual programming The way programs are written in Scratch is by fitting "blocks," together rather like toy Lego bricks; or pieces of a jigsaw puzzle. In this respect, the programming language in Scratch is a "visual language", (Green & Petre, 1996).

The blocks can only fit in ways that make sense, because of their shapes, so it is not possible to get error messages from the compiler. This is a great relief for introductory programming, and saves the learner from much of the heartache traditionally forced on them by textual languages. Learners in Scratch are not bullied by the compiler when they forget a semicolon or have mismatched brackets, because such errors are not possible. To the extent that novices get frustrated or daunted by floods of compiler errors, the visual language in Scratch gives it strong appeal for educational purposes.



```
when Sprite1 clicked
think erm... for 2 secs
say Hello world!
```

Fig. 1. Screenshot of a classic "Hello World!" program in Scratch

A traditional first program is shown in Fig. 1, where the default character (on the right, in the canvas window) responds when you click on the sprite, with a thought-bubble that appears for two seconds, and then says “Hello World!” in a speech bubble. The script that does this is shown in the bottom middle window, which was made by drag-and-drop from the palette of blocks in the left window.

Other palettes are available in the upper left window, which include blocks for program control logic (selected), blocks to move sprites around the canvas, blocks to draw on the canvas, blocks to sense events like collision detection, blocks for playing sounds, and so on. There are blocks for arithmetic, boolean and string operations, and for variables too.

1.2 Related work — evaluation of Scratch for novice programmers

The Scratch system has been a big hit with its intended users, in computer clubhouse environments, as reported by the developers (Maloney et al., 2008). The children spent more time working on Scratch than with any other package they had available to them. It seems clear that Scratch succeeds very well in fostering creativity and in social sharing of the multimedia products.

It was envisaged from the outset that while this project was to introduce computers to deprived areas, the educational benefits would be researched at a later date (Resnick, Kafai, & Maeda, 2003). Because Scratch is a new system, there have only been a few studies of its use in teaching programming, so far.

In one study, Scratch was used at Harvard university (Malan & Leitner, 2007) , where it was used to introduce novices to programming before their transition to Java. There was an almost total approval of Scratch amongst the learners who were true novices; and the only learners who disagreed that it was useful to them were the few people who had already some experience of programming.

Another pilot study was in the USA with 8th grade girls at middle school, where the aim was to see whether the pupils would learn to appreciate the basics of programming in the span of a three-hour workshop (Sivilotti & Laugel, 2008). The girls were not complete novices, all having used either Scratch or Lego Mindstorms or Logo before. They reported feeling that they had learned something worthwhile and how much fun they had had (average 3 and 3.4 respectively, on a 4-point scale).

2 Method: to try Scratch out in a real classroom at primary school

The existing studies above have evaluated Scratch either informally, in after-school activities, or more formally with older or more experienced students. It was generally observed that Scratch was fun to use, and there were some observations about learning ocurring.

The purpose of the present study is to evaluate the use of Scratch in school lessons as an introduction to programming for total novices, in a younger age-group at primary school. It focuses on two possible kinds of benefit: cognitive and affective; we are interested to know whether Scratch teaches concepts well, and whether it is fun to use for the younger age-group in a school context.

2.1 The school and pupils

The primary school chosen for this study is in a relatively deprived area of Glasgow, in Scotland. The class has twenty-one pupils, who are all eight or nine years old. One of us (AW) approached the school teacher to offer taking her IT lessons for the whole term of eight weeks. The teacher agreed, being interested to see how well the lessons would go with Scratch, as compared to the normal ICT lessons that she gave the pupils, in which they would typically use office applications or surf the web.

2.2 The lesson plans

Scotland is currently renewing the schools curriculum, and so the lesson plans were drawn up with the new *Curriculum for Excellence* (LTS, 2010) in mind. That way, the teacher can continue to use the lesson plans later on, if she so chooses, because the lessons satisfy the desiderata of the new curriculum.

For each of the eight weeks, there was a one-hour lesson. At the start of each lesson, we showed all the pupils what they would do next, on a whiteboard at the front of the class. Then they went to work in pairs at the computers, to try and achieve the task using Scratch.

The tasks were to make a sprite move around the canvas, either to make patterns, or to visit certain locations in turn and end up at a target location. Each week's lesson was a little more complicated than the previous one.

In order to illustrate to the children what task to achieve for each lesson, the demonstration was given either on the whiteboard, or using a small remote-control toy, which was a toy robot. Some children would call out instructions to the one with the remote control, who would then control the toy robot. By this kind of concrete programming (Demo, Marciano, & Siega, 2008) the children can think through what sequence of actions is required to get the robot to its destination, and they are then ready to try the task with Scratch.

First visit – set baseline of understanding Find out what the children might already know about programming.

Illustrate the concept of algorithm with an example of making breakfast: (1) Get cereal box, (2) get bowl, (3) get milk, (4) pour cereal into bowl, (5) pour milk into bowl, etc. Emphasize the importance of getting the order right (to help understand sequencing later on).

Lesson one – introduction to Scratch Introduce the children to Scratch, with a worksheet that shows a couple of program "blocks": (a) to "move 10 steps" and (b) to "turn right 90 degrees". The children can experiment with the effects of these blocks on the cat character, and they can try different numbers of steps to move or degrees to turn. Then they can look at the other blocks available in the palette and come up with their own ideas to try out for the rest of the lesson.

Lesson two – introduce "sequence" Hold a class discussion about how a program can make shapes on the canvas, by using the pen (as with turtle graphics). Then demonstrate the program with a remote control toy, and let the children go to the computers to put the programs into Scratch and try them out.

Lesson three – first class test The exercise is to write a program to move the sprite (cartoon character) across the canvas, while on the way passing over each of the coloured shapes that have previously been drawn on it by the tutor. This cannot be done with a single straight line, so the children have to make a route out of straight segments joined by 90 degree turns (see Fig. 2). Before trying to do this in Scratch, they first work out the path they want and the instructions required to draw it, and they write their little program on the paper worksheet.

Their worksheets are collected for later marking to monitor the pupils' progress. Marks for this test (out of eleven) are awarded according to how much of the path the child manages to produce:

– did he or she draw the line from start to finish?
– did he write down correct instructions, that get the sprite to the end?
– did he put in the correct turns, in direction and in degrees?

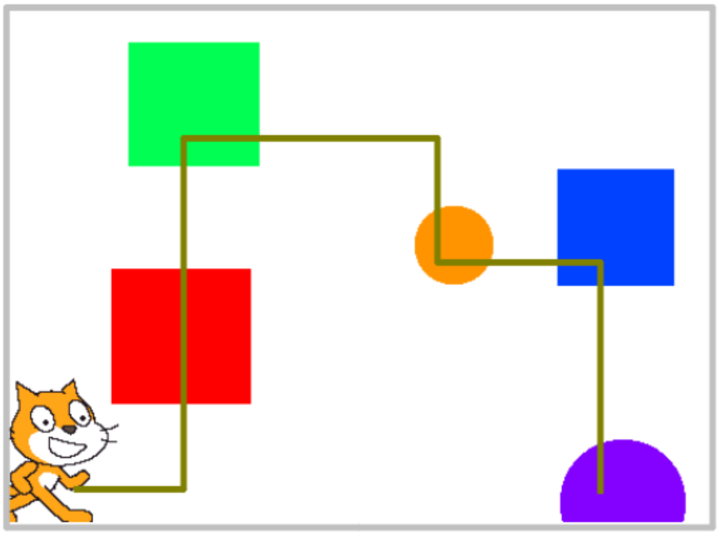
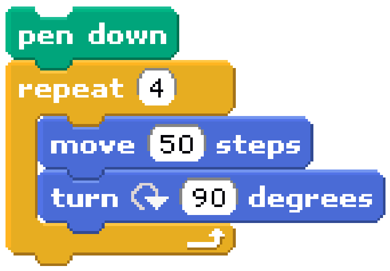


Fig. 2. Lesson-3 exercise, showing a route that visits all the shapes

Lesson four – iteration Introduce the children to the "repeat" block, as a way to make repetitive scripts shorter. Show an example script of a line segment followed by a quarter turn, and then enclose it inside a repeat block (see Fig. 3), that runs it four times... to make a square.



```
pen down
repeat 4
    move 50 steps
    turn ↻ 90 degrees
```

Fig. 3. Lesson-4, showing a repeat-block for iteration

Lesson five – selection Introduce the class to conditionals, by using *if- else*-statements to make their sprite rebound when it collides with the endge of the canvas, or another sprite.

Lesson six – coordination and synchronisation Using the "broadcast" block, which sends messages to any other blocks that care to listen, the children can make a short animated sequence in which two sprites talk to each other (see Fig. 4).

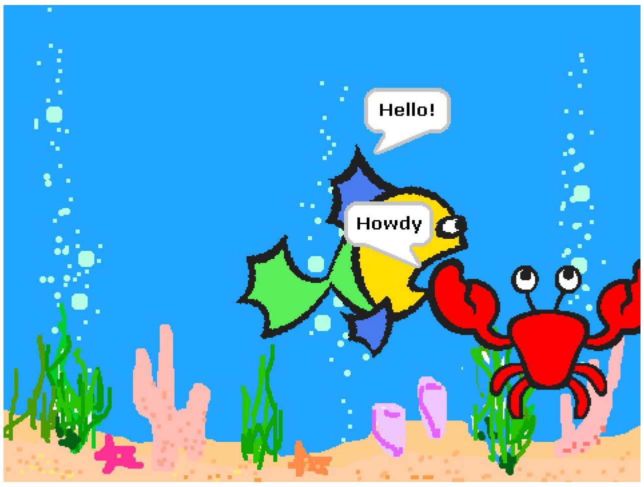


Fig. 4. Lesson-6, showing an animation with sprites conversing

Lesson seven – the Scratch cards A set of twelve cards is available to download and from the Scratch website, each of which helps the learner to explore another feature of Scratch. One card shows how to play sounds, for example, and another one shows how to make a sprite follow the mouse cursor.

Show the children how the cards suggest things to try in Scratch, and let them work their way through the set.

Lesson eight – second class test The exercise in this lesson is to be marked afterwards, and includes several tasks, of which one is comparable to the first class test from Lesson-3. The latter task is similar to the one from the first class test, except that it has a set of shapes in different locations, necessitating a different path to negotiate them.

Unfortunately, because the exercises in this lesson were longer, the children did not all finish them, and some rushed their answers. For this reason, we did not use the results to compare with Lesson-3.

Week nine – after the lessons, a final class test We set the class another test (Test-3), similar to the first class test in Lesson-3, and this time without extra time-pressure. The results from this test were used for further analysis (see below).

2.3 The measurements taken

We wanted to know how the lessons compared with the class's other, normal lessons in ICT. The two major factors were *cognitive* (how effectively they learned) and *affective* (how enjoyable the experience was, and how motivated by it the pupils were).

As well as some simple questionnaires for the pupils, their behaviour was observed during the lessons, and the teacher was interviewed for her reactions and opinions, as she knows the pupils well.

Cognitive measures In order to measure learning progress, the pupils were set some questions at two points during the term: the middle and the end (lessons 3 and 8). The questions were inspired by the Cambridge "ICT starters" syllabus for assessment of early progress in ICT skills (University of Cambridge International Examinations (CIE), 2010).

While the “ICT Starters” curriculum covers ICT skills from word processing and spreadsheets to email and web-browsing and authoring, it also includes *control*, which is more closely related to programming concepts. Children are to demonstrate control by giving simple commands to a device; and by using a sequence of commands to control a device, including inputs and outputs. The programming language to use for these activities is *Logo*.

Our tests and scoring schemes were based on their assessment ideas, which allowed the childrens’ work to be judged and quantified as to the level of skill demonstrated. In order to show that children have developed some facility for control of a device, the curriculum requires that they produce a sequence of instructions that involve at least a certain number of line segments, and a certain number of 90-degree turns. The class tests were devised to embed these requirements into the tasks set for the children, in making a sprite navigate around the canvas, visiting various locations on the way.

Affective measures In order to measure how enjoyable the children found their lessons with Scratch, or whether they were growing at all frustrated, they filled in a brief log-sheet after each lesson, to say what they did in the lesson and how they felt about it. Rather than ask such young children to describe their feelings, the log-sheet had three cartoon faces (sad / neutral / smiling) which they could mark with a cross (see Fig. 5).

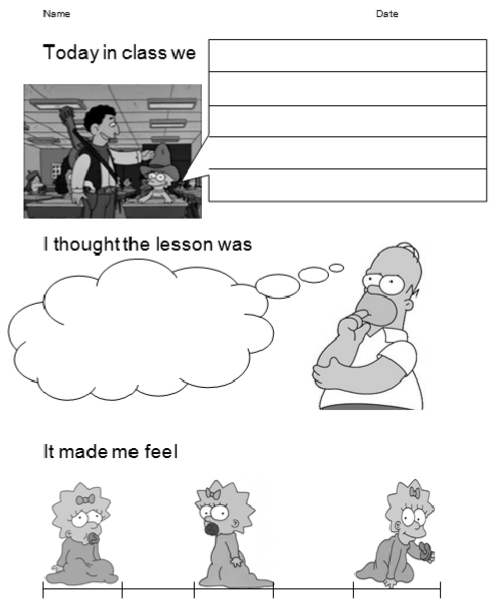


Fig. 5. Affect measure: a log-sheet that young children can easily understand

3 Results

There were twenty-one children in the class (5 girls and 16 boys), but some did not attend all the lessons. All were eight years old, except for the four nine-year-olds.

Nineteen pupils had a computer at home, but none of them knew what a computer program was before the lessons. They had never done any form of programming before, neither at school with a teacher, nor at home. The pupils mostly thought that a program was something to do with the internet, or with computer games.

3.1 Cognition and learning

The tasks that we used as tests, and marked, were given in weeks three, eight and nine. Unfortunately the scores for the second test were low, because the children ran short of time in that class, so we use the scores from Test-3 instead. There was a problem with that test as well, in that it took place near the holidays, and several children were absent that day for that reason, and because of an infection that was going through the school at the time.

Leaving out the missing values, there were only $N = 12$ pupils that had scores for both Test-1 and Test-3. The mean score for the tests were 52% and 64%, respectively. Although this shows an improvement in the pupils' performance, the difference is not statistically significant at the 95% level (paired t-test, t = -1.741, df = 21.202, p = 0.09617).

3.2 Affective experience of pupils

At the end of each lesson, the children marked on their log-sheets how they felt about the lesson with Scratch. Answers were on a 3-point scale, shown by three cartoon faces which were either sad or neutral or smiling. The result averages are shown in Table 1, where the missing values for the lesson in week-2 are shown as blanks: there was no time to fill in the log-sheets that week.

Table 1. How the pupils felt about their lessons (sad, neutral or happy)

lesson :	1	2	3	4	5	6	7	8
happy	19	-	20	20	18	18	18	15
neutral	0	-	0	0	2	1	2	3
sad	0	-	0	0	0	0	0	0
absent	2	-	1	1	1	2	1	3

It is clear from these results that all the pupils enjoyed the lessons hugely. Nobody was ever sad, a few were neutral for at most two of the weeks, and all other marks were for smiley faces. In fact only five of the twenty-one pupils ever marked a lesson down to neutral.

3.3 Teacher's views

All the lessons were lead by one of us (AW), while the teacher watched and helped, because this was new to her as well as to the class. At the end of the term, she was interviewed for her personal assessment of her pupils' progress, because she knew them well and could compare their performance and enjoyment in our lessons with the way they were in other classes. Except for people's names, her answers are transcribed here *verbatim*, as follows:

Question: What expectations did you have at the beginning, and have they been met?
Answer: Without a doubt – they have been exceeded!

Question: How do you feel your class performed in the Scratch lessons compared to how they would perform in normal ICT lessons?

Answer: Far more enthusiastically. They picked it up very quickly and easily; I don't know why. Maybe it was the Scratch system, or maybe it was your tutoring. But they were more enthusiastic than in their other subjects. They were very keen, kept at the task, and didn't have to be told to keep quiet.

Question: How were they compared to how they would perform in maths lessons?
Answer: Much better.

Question: From at the test results, did any children do better or worse than you'd expected?
Answer: Yes:

- one did much better, as he seems to be really good on the PC. He is good at maths too, but finds language difficult.
- Another one made a great improvement.
- But I was surprised that two others did much worse than average in this class, while they are in the top group.
- *It's obvious to me the less able pupils are doing better with Scratch.* I'm surprised at those not doing well academically doing really well with Scratch. Some of them have language barriers as well.

Question: Did you enjoy the lessons?
Answer: Yes I did.

Question: Would you recommend Scratch to your colleagues as a tool to teach computing? Will you use Scratch yourself in future?
Answer:: Yes, without a doubt — it's a great tool for teaching. We'd like you to come again and show us teachers more about it.

4 Discussion and conclusion

It is clearer from the teacher's answers than from the other data just how much better the progress and behaviour was in these Scratch lessons, compared to other classes. Consider in turn the affective and cognitive factors at play.

4.1 Was affect important?

The level of enjoyment was consistently high, for all pupils and for every week. It was noticeable that the pupils were laughing quite a lot, and showing their work to each other, and to the teacher. Some might think this emotional side to be unimportant, or much less important that the cognitive effects such as evidence of learning; but we do not. Affect is important for learning, and not just as an accompaniment. Learning will hardly progress without motivation, and that is stirred and maintained by positive affect. The teacher of these children also seems to prize the fun that she sees in the class when they learn to use Scratch.

The teacher's remarks about some of the less able pupils doing very well were not entirely suprising to us – we had suspected that might happen for one or two children who were otherwise difficult to reach. But it certainly was a surprise to us that a couple of the academically strongest children did conversely: rather poorly. While they were normally in the top group, in our lessons their performance was amongst the lowest in the class. We hope to discuss this matter again with the teacher, and until then we cannot explain it.

There is little doubt that Scratch, in combination with the lesson plans we used, was a big hit with the whole class; all the pupils, and the teacher too. Our formal measure of affective reactions clearly showed that the lessons were enjoyable for all the pupils.

Was it merely about novelty? It is of course possible that this effect is entirely due to novelty, as the pupils had never seen Scratch before, nor their new tutor. That issue can only be entirely settled with a longer study. However, we note that eight weeks is quite a long time for a pure novelty effect to sustain, without the slightest fall. Note also that the "new tutor" had the pupils' attention for the first 5-10 minutes of each class, and after that they went to work on the computers in pairs, as usual. Perhaps it might be argued that the lesson plans introduced more novelty each week, with new facets of Scratch programming to explore, and that it was the continual recycling of novelty that sustained the novelty effect. By that argument, however, all forms of learning would be self-reinforcing by means of novelty cycling alone. The argument is thus a facile one, and we reject it in favour of concluding that Scratch has shown itself to be beneficial (and fun) for learning programming, and for more reasons than mere novelty.

4.2 Is there good news on the cognitive front?

Our formal measure of cognitive progress did show some improvement, but not enough to be statistically significant. The results were hampered by missing values; and perhaps it was too much to expect in only eight hours of lessons, albeit spread over eight weeks.

It may not have been the best idea to use our Test-1 and Test-3 in order to measure progress, because both tests were about the same subset of skills. That makes for a convenient comparison, but on the other hand it also misses any other learning that may take place. For example, in Lessons four and five the children learned about iteration and selection statements; but there is no need for them in the class tests, so any learning there would be missed by them. The only way that the tests would show learning is if the children improve their performance on the basic skills by practicing with Scratch while learning the more adanced ones.

In future work it would be an idea to plan out a more open-ended set of challenges, that would allow pupils to use things like iteration and selection, if they knew them; but would let them achieve success in a simpler, more tedious way, if they didn't know them yet. For example, a program with an iterative loop may be equivalent to a longer one where the loop has been unfolded. A sensitive marking scheme would be needed, to reward any flashes of inspiration and novel attempts that don't happen to work, but are still evidence of insight. Until then, we must conclude that the advantage of the Scratch system, now applied to teaching programming, has not been formally demonstrated in this study. Quantitative results did not significantly support it.

In order to prove that Scratch is better than any usual alternative, a controlled study where the alternatives are played off against each other would be required. Without doing that, however, we can still make a case in favour of Scratch. **First:** note that in only eight weeks the young children tackled the key elements of programming (sequence, iteration and selection); and more, besides. They were also introduced to synchronisation between scripts, and elementary multimedia effects. **Second:** imagine trying to achieve the same progress in the same time with a standard alternative – say with Java and Swing. Give the eight-year olds a nice IDE as well if you like, to help them along. **Third:** no, there is no third step, since the second step is already incredible, isn't it? The present authors certainly cannot imagine it happening, unless perhaps with child geniuses.

Other curious observations It was good to see that the Scratch lessons were able to reach some of the pupils who are known to have difficulty with other classes; with "language" in particular.

However, it is puzzling that some of the ordinarily stronger pupils did uncharacteristically poorly in the Scratch lessons. This issue is a matter for further investigation.

For the time being, it seems safe to conclude that Scratch looks like a rather successful way to introduce programming concepts to children as young as eight years old; even including some who suffer from a degree of learning difficulty.

It remains to be seen whether the pupils will maintain their enthusiasm about programming when they later encounter more conventional languages, with fussy syntax; but at this point we are encouraged at the prospect. We are thinking of ways to use Scratch in future as part of our outreach programmes, for example to help schools learn how to deploy it in the classroom.

4.3 Ideas that may help explain success

Future work could look at the features of Scratch, and attempt to break down which ones are the most crucial in achieving the generally happy results. The two major features of Scratch are the visual language, and the multimedia environment. Each one may well have a double benefit, as we shall now speculate.

Benefits of visual language Because the programming language is visual, it benefits both the cognitive and affective sides of learning. The essence of programming includes the key concepts of sequence, iteration and so on; matters of syntax in a textual language are relatively superficial, and so are less important. In order to learn the key concepts, however, the pupil has to first master enough syntax to support them. Therefore, the material has to be learned in the wrong order, with the lower priority syntax taking precedence. Using a visual language does not solve the problem of syntax, since the student will have to tackle that later when learning a second, more traditional language; but it does postpone the problem until the student has grasped the fundamentals. In this way, the memory load is kept manageable at all times, instead of overloading the mind all at once with barely sensible detail. This is a big *cognitive* benefit.

The *affective* benefit of learning a visual language is simply that a lot of heartache is avoided, which is to say that the potential for negative affect is neutralised. It may well be more important to prevent negative emotions from intruding into learning, than to encourage positive ones. Therefore, this benefit is particularly strong.

Benefits of multimedia platform The *affective* benefit of the multimedia environment is fairly obvious: it's fun to play with. Consequences of that are that the pupil will happily spend more time with the system, including leisure time, and will also tend to explore the system more, and try out new program blocks and other things that a more sober system would not encourage. Such exploration, naturally, should accelerate learning.

The chief *cognitive* benefit of the multimedia elements, that are fairly easy to manipulate, may be the instant and vivid feedback that makes the internal workings of the programmed system so much more apparent to the learner. This is because the program statements can be so closely related to the multimedia elements that the sprites and sounds and events in the 2D virtual world are (virtually) the program itself in motion.

A more traditional program for a novice task would involve manipulation of data structures, some calculations, and eventual printouts of results; but to make the internal workings of the algorithms more visible, suitable print statements would need to be set in the program, which can be a tedious process that also requires some astute thinking on the part of the poor novice to place them optimally. Yet again, the learner could become overwhelmed; either that or could become less ambitious, and rather reconciled instead to slower, more tedious progress, soon to be followed by boredom.

It is noteworthy that these affective and cognitive benefits are not independent; rather, they tend to feed back into each other. This reinforces our view that the affective side of programming is in its own way at least as important as the cognitive side. If that is so, then an ideal educational system to help learn how to program should be designed with as much attention paid to the learner's emotional state as to the cognitive dimension.

How many such systems or approaches are out there? We know of at least one.

5 Acknowledgements

Thank you to the teachers at the school and their pupils who took part in this study; and many thanks also to the kind and careful reviewers.

References

Demo, G. B., Marciano, G., & Siega, S. (2008). Concrete programming: Using small robots in primary schools. In *Proceedings of 8th IEEE international conference on advanced learning technologies* (p. 301-302). IEEE computer soc.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, *7*(2), 131–174.

LTS. (2010). *Curriculum for Excellence. Technologies: experiences and outcomes* (Govt. Rep.). Learning and Teaching Scotland (LTS). (Available at website: http://www.ltscotland.org.uk/curriculumforexcellence/technologies/)

Malan, D., & Leitner, H. (2007). Scratch for budding computer scientists. In *38th sigcse technical symposium on computer science education* (Vol. 391, pp. 223–227).

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. *Proceedings of the 39th SIGCSE technical symposium on Computer science education – SIGCSE '08*, 367.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas* (2 ed.). Basic Books.

Resnick, M., Kafai, Y., & Maeda, J. (2003). *A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities.* (Proposal to the National Science Foundation, USA; project funded 2003-2007)

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., et al. (2009, November). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–68.

Sivilotti, P. a. G., & Laugel, S. a. (2008, February). Scratching the surface of advanced topics in software engineering. *ACM SIGCSE Bulletin*, *40*(1), 291.

UKCRC. (2010, January). *Computing at School : the state of the nation.* Available at website: http://www.ukcrc.org.uk/.

University of Cambridge International Examinations (CIE). (2010).

Students' early attitudes and possible misconceptions about programming

David C. Moffat

School of Engineering and Computing,
Glasgow Caledonian University,
Glasgow, Scotland, UK
D.C.Moffat@gcal.ac.uk

Abstract. Programming can be unpopular with some university students of computing, who may then go on to graduate without good programming skills. This unpopularity threatens student recruitment into the core computing courses and professions, and may weaken the economy. It may be that negative attitudes harm the student's interest and confidence in programming, making for an unsatisfying learning experience.
In this pilot study, student attitudes towards programming, and possible changes in attitude, were investigated by means of a survey on a university's introductory programming course.
Results indicate that some students have negative attitudes toward programming, and programmers; and this applies to school pupils as well. A minority of the students questioned retained their frustration and dislike of programming throughout the course, but others came to love it in the end. Interpretation of the results leads to speculation regarding the quality of the teaching of programming, both at school and at university.

1 Introduction

In the UK and some other industrialised countries, student recruitment onto computing courses is falling. There may be various reasons for this, including demographic changes, curriculum changes at secondary school level, inadequate resources devoted to computing subjects, possible "dumbing down" and so on. But attitudes and possible misconceptions about programming might also have something to do with it.

When at university, following a computing degree, some students display an aversion to programming; when given the choice, they opt for project work that does not involve software development. Reasons given include being "uncomfortable with" programming, or not being able to program competently. This has been my own experience, in supervising student projects, or leading other kinds of student coursework in which programming is optional in some sense.

It may seem paradoxical, to suggest that some computing students dislike programming, because then it would be strange for them to choose to study computing at university in the first place. However, we should remember that students do not know the subject very well, and initially they do not share the more educated views of their teachers. Moreover, there are differing views amongst the teachers themselves. (At my institution, for example, I fall in the "hard-core" camp, because I consider that programming is the heart of computing; but there are colleagues who disagree with that.) Finally, the situation at schools (in the UK at least) is still worse, for there it is quite common for teachers to confuse computing with information technology. Being unclear about the difference, it would be no wonder that their students are confused, too.

Anecdotally, from talking with colleagues in staff room conversations, whether they see programming as essential to computing or not, it is generally accepted that some students show a distinct aversion to programming. However, there is little literature concerned with this issue. The present study is an initial attempt to explore student attitudes toward programming, and whether they have an impact on their learning.

While there is much attention devoted to the cognitive aspects of learning how to program, the affective aspects are relatively unexplored. Bennedsen and Caspersen (2008) sought correlations between some affective variables, including perfectionism, self-esteem and optimism, and performance in computer science courses. They found some interesting results, such as that self-esteem was correlated with success, as measured by good course grades. However, in present study is directed at a different set of affective variables, or attitudes: namely, self-efficacy, and social stigmas.

1.1 Confidence and self-efficacy in learning

It is known that *self-efficacy* (Bandura, 1994) is important in learning. The concept of self-efficacy relates to how people perceive their own capacity to achieve their goals, or goals that have been set for them. In an educational context, for instance, students have to learn things that may be quite difficult, and have to undergo examination to see whether they have learned successfully, and to what standard. While students are learning, they do not know whether they will succeed, in general, both because they are not yet familiar with the material to be learned, and because it is set at a fairly difficult level which should stretch the average student.

Self-efficacy is important in determining whether a person will succeed at a difficult task, because the strength of the belief can encourage the person to persevere, and so be more likely to overcome obstacles that appear along the way. It is related to confidence, but Bandura did not like to use the word "confidence" in his writings, because it can be used in a vague and general way, which is why he introduced "self-efficacy" as a new technical term. However, in ordinary language the concept may be rendered as a person's self confidence in being able to achieve a certain level of performance at some specified task, including both the strength or certainty of that belief, and the degree to which the level of attainment might be exceeded. In the context of learning how to program in a university course, for instance, a student may strongly believe that he will pass the course, but with only a low mark; or may initially believe that he could pass with a high mark, but believe that only weakly, so that he could be easily discouraged by an early setback.

The strength and degree of self-efficacy that a student has can clearly have a large impact on learning, therefore. There is also an interaction with affect or emotion, as minor successes and failures may confirm or defeat expectations. The emotion in turn can change self-efficacy as perceived by the student, and so regulate further learning performance. Bandura sees emotion as an important aspect of self-efficacy.

Programming is a challenging new skill to learn, for novices who have no experience of doing anything similar as far as they know, and so they can be extremely uncertain in their self-efficacy. Therefore one aim of this study is to question the nature of students' self-efficacy as regards learning how to program.

1.2 Negative attitudes about programmers may play a part

There are other reasons to be interested in the affective attitudes of students toward programming, such as any stereotypical views of programming and programmers. If students at a younger age, when still pupils at school, adopt society's prevailing attitudes toward programming, some of which may be seen as negative, then clearly there could be a large impact on study and career choices. Furthermore, if any students at university still retain such unfortunate attitudes when they have joined a computing course, then their learning could be retarded as a consequence, unless and until their attitudes can be corrected by experience. Any social stigma that may attach to a profession could undermine student motivation to choose that career, and study that subject.

1.3 Related research

Although there is not much literature about the affective attitudes of students toward programming, an interesting and related line of research is in the differences between boys and girls in scientific and technical subjects, and specifically in computing.

Beckwith and Burnett (2004) surveyed the literature on gender and learning, and drew several inferences about how best to design software development systems to benefit the strengths of both sexes. Beckwith et al. (2006) investigated a sex difference in tendency to "tinker" with, or playfully explore the features available in software products. Males tend to tinker more, which may account for their sometimes faster progress in learning to program; whereas females seem to be more reluctant to tinker with features they don't know about. This could be seen as a difference in learning style; or it might be interpreted as a form of risk-aversion in females, which would be a difference in affective attitudes of the sexes.

Differences between the sexes in self-efficacy in computing were found by Busch (1995), which may be attributable to the different personal histories, in that males had more experience of computing, and had benefited also from more encouragement from family and friends. In that case, gender stereotyping may be amplifying any difference in ability, via an affective route.

In a study of the personal histories of fifteen women who excel in their chosen careers of mathematics, science and technology, Zeldin and Pajares (2000) found that familiar role-models (male or female) and social encouragement were both factors in determining the career choices of the successful women. Each factor contributed to higher self-efficacy, which in turn led to greater resolution and higher performance.

The present study is not focused on women in computing, or on gender in any way; but the above literature is nevertheless relevant to our concerns, because it indicates that self-efficacy is a common thread in determining a person's success. Noting that apparent gender differences might not always be actually gender-specific, we can hypothesise that the above noted gender differences in self-efficacy and its causes may also be reflected within each of the sexes. Just as encouragement may help a girl to choose to study computing, and social stigma may put her off, the same factors may play a role with some boys more than others. For example, a boy who thinks that programming is difficult, and that programmers are unpopular or "nerdy," can be put off computing just as a girl might be.

This study is to investigate *all* students' attitudes to programming, therefore, and how they may change from school through to university. Do they have misconceptions, or experience misapprehensions about programming to the point that their morale and confidence suffer? By querying the students at key points during the semester, changes in their opinions may be tracked as they learn to program.

2 Method: to survey students about their current and previous attitudes

A questionnaire was given out to all students on an introductory programming module, which is to teach the object-oriented language C#. The questions are about experiences and opinions of programming at different times, as far as they can be recalled. In chronological order, these times were (a) from high school, (b) then just before the beginning of the module, (c) and then towards the end of the module (three quarters of the way through it).

2.1 The students

Glasgow Caledonian University is a "wide access" institution of higher education, which takes all sorts of students, including many from families that have never been to university. Consequently, they may be unusually prone to low self-efficacy because of the lack of role models in their families. Students can also enter the university at second year directly, coming from local colleges of further education. Some of the latter students had good exposure to programming in languages like Java, and fair programming skills. They needed more practice, exposure to the C# language, and some deeper grounding in the principles of programming.

The introductory module that all the students were now following is a second-year module in object-oriented software development, in which the chosen language is C#. The students who started in the first year had a brief introduction to programming in that year, but it was slight in many cases, and some students can therefore be considered to be new to programming.

The module was taught in a way that would probably be considered unusual, compared with similar introductory modules (or courses) at other universities in the UK or elsewhere. There were some lectures, but the emphasis was on practical work, in which the students were expected to work in teams of four, and each team was to make a different, small, 2D video game. To do this they used a basic game engine written in C# and made publicly available by DigiPen,[1] which is a higher education institute for computer gaming, based near Seattle, USA. The DigiPen codebase uses DirectX technology, which was also new to the students, who were told the minimum they needed to know about it in order to be able to use the DigiPen libraries.

This all provided for a more authentic experience than is usual in student programming modules, and allowed them to take on a task that would show them what programming can really empower them to do. The coursework was therefore a relatively exciting assignment, giving the students wide scope for creativity and challenge, but also may have been daunting for some students. It will be useful to bear these details in mind when interpreting the results later on in the paper.

2.2 The questionnaire

The questionnaire was delivered on-line within the virtual learning environment that was being used to support the course. Questions were mainly of two formats. There were some Likert-style questions about skill or competence, and about attitudes and emotional reactions, like anxiety. There were also open-text questions, for less predictable responses. Some of them asked students to write in their own words what surprised them about programming, for example, or what their high and low experiences were.

The first few questions were about the student's background and previous programming experience. This was to see how many of the students were novices, and how many had significant programming skill.

Then, most questions were intended to detect changes in attitude from one time to the next, by asking students to recall their opinions about programming and programmers at earlier times in their lives. The three times of inquiry, and how they were expressed in the questions, were:

school "This question is about when you were starting your final year at school."
begin "This question is about your opinions in about August 2009, just before this module began."
now "This question is about now."

The questionnaire was answered by students about three-quarters of the way through the semester, after eight weeks of the intoductory module on (object-oriented) programming. This was the time referred to by "now" in the above.

[1] http://www.digipen.edu/gamers/tutorials/introduction-to-2d-video-game-development/ (accessed 07/May/2010)

3 Results

The survey was made optional to the students, and so not all of them took it. A total of 53 students started the module, and they were invited to answer the questionnaire on-line, in their own time. One student withdrew during the module, twelve did not attempt the survey at all, and two did open the survey online to look at it, but answered no questions. The response rate was therefore 38 out of 52, which is 73%.

Of those 38 students, eight answered only the first few questions, but then did not continue to finish the survey. Therefore only 30 of the 38 students answered the survey fully.

Because not all the students answered the survey, and some of them only answered some of the questions, the following results have been aggregated over all 38, in order to use all the data that was available.

3.1 Questions about programming competence

In order to find out about the background of the students, and their earlier self confidence, the following question was asked about their time at school, and the same question about the other two, later time points. It queries the student's *self-efficacy* regarding ability to perform a fairly straightforward programming task.

– This question is about when you were starting your final year at school. It is also about writing a small command-line, console program to implement a simple telephone directory, with name and number for each person. Back then, did you have the ability to write such a program?

Answers were given in the form of a Likert-scale, selected from: (s-ag) meaning "strongly agree"; (ag) meaning agree; (neut) for neutral, or neither agree nor disagree; (disag) for disagree; (s-disag) for strongly disagree; (na) for not applicable; and finally (un) for unanswered.

The time points (**t**) in the table are represented by **s** for school, **b** for beginning of the module, and **n** for "now" (which was towards the end of the module).

t	un	na	s-disag	disag	neut	agree	s-ag	median
s	0	2	6	10	6	12	2	neut
b	5	2	1	2	15	10	3	neut
n	8	0	0	0	2	20	8	agree

The figures in the table show how many Ss (students) answered the questions at each point on the Likert-scale: for example, six Ss strongly disagreed that they could write a small console program when they were at school. The total number of Ss who answered the survey was 38, but for each question some might not have answered, and so any missing values are shown in the **un** column. In the following calculations, the "not applicable" responses are also left out. For clarity, the resulting N values are shown below. The median point for each question is given in the last column.

From this table it appears that the students now are more confident than they were at school, or even at the beginning of the module, in their ability to write a small program. To confirm this a standard nonparametric test was run to compare each pair of rows. According to the Wilcoxon-Mann-Whitney test, the scores for **now** were (significantly) higher than for **beginning** (W = 275, N_b = 31, N_n = 30, p = 0.001133 two-tailed); and yet more significantly higher than for **school** (W = 274, N_s = 36, N_n = 30, p = 0.000084). The same trend is evident from **school** to **beginning**, but not significantly so (W = 421, N_s = 36, N_n = 30, p = 0.07458).

Most students have clearly increased their confidence in their ability to program — somewhat, from **school** to **beginning**; and a good deal more so, from **beginning** to **now**. We cannot say why their abilities improved in the first case; but the module has obviously been the reason for their most dramatic improvement (thankfully).

Note that the key phrase "confidence in their ability to program" deliberately obscures our assessment of their ability, by including the word "confidence". Because the students are answering the question themselves, and we are not independently assessing their programming skill at the earlier time-points, we cannot conclude anything directly about their skills, strictly speaking. This question is therefore interpreted as querying the students' *confidence* alone.

However, we may choose to infer that students have a fair idea of their own abilities in such cases, in which case we would indeed conclude that their increased confidence is justified by genuine learning. Then it is clear that, of those students that could not program before, most successfully learned how to, during the module.

One thing to note, on the other hand, is the small, hard core of students who still have low programming confidence (or skill) even towards the end of their first full programming module. There are two Ss who are still neutral on the question, even **now**.

We also note the wide range of abilities before the module began. At the beginning of the module, after their first year of higher education (or equivalent), already half the students were able to program. Some of them had learned quite recently, but most of them already knew some time beforehand, towards the end of their school education.

Therefore, quite a large spread of aptitudes is revealed, highlighting another problem that we have in teaching introductory programming at university. When there is a very wide range of abilities in a class, the teacher finds it very difficult to pitch the lessons at a good level – there can be no optimum level. This generally means that the weakest students will be daunted, while the strongest students could become bored and grow disdainful. The effect might be to put the students off programming even more.

3.2 Questions about general opinions of programmers

The following questions were about the students' former attitudes towards programmers and programming, to see if there was a perceived stigma attached to computing subjects, which might put students off choosing them.

First there were two Likert-scale questions regarding their opinions when they were at school, and now.

- Did you think of programmers (software developers of any kind) as cool?

t	un	na	s-disag	disag	neut	agree	s-ag	median
s	5	2	1	2	15	10	3	neut
n	8	1	0	3	8	10	8	agree

It appears that there has been an improvement in attitude, with a shift in median from "neutral" to "agree", suggesting that more Ss now think that programmers are "cool".

This appearance was checked with the Wilcoxon-Mann-Whitney test, but not found to be significant at the 95% level (W = 343, N_s = 31, N_n = 29, p = 0.09839).

.

In order to let the students express their opinions in their own words, the following question was asked about each of the three time-points, to which responses could be typed in as free text.

– This question is about when you were starting your final year at school. What were your opinions about programmers and programming? (E.g. clever, boring, difficult, easy, nerdy, lucrative, important, misunderstood . . . ?) But in your own words or phrases, just off the top of your head.

There were too many answers to repeat, but here is a representative sample of what the students said, regarding each of the time-points. The responses were grouped into positive / neutral / negative categories, according to my judgement about the mood expressed in the attitudes. The selected responses reflect the sizes of the categories, and are shown in order from most positive to most negative.

– School [13 positive, 7 neutral, and 5 negative]
  1. Very easy and fun
  2. Intelligent, Difficult, Fun, Enjoyable, Creative
  3. Misunderstood, intelligent, problem-solvers
  4. Intelligent, i never thought i would be able to do the things they could
  5. "Taught what we needed to pass the exam, not what you need to program."
  6. I thought programming was boring, hard and the thing that someone else should do. Only the real nerds could understand it at all
– Beginning (before the module) [10 to positive, 13 no change or other, 1 negative]
  1. Programming is not as hard as i initaly thought
  2. not so difficult as my last year of school but i feel like i gotten rusty not programming as much i should have
  3. My orignal opinion that programming would be difficult to learn. I realised after some time that most programming languages are the same. It's just a case of learning the sytax. Just like having to learn grammer, when learning a new language.
  4. "The realisation of what a programmer actually does and how they have evolved"
  5. We only got visual basic and it was poorly taught as you didnt understand what it was really doing. So I was still anxious about starting programming. No change. {Note: this student seems to have been to college before entering the university directly into second year.}
  6. It gets far more complicated and complex than I thought, I no longer enjoyed it.
– Now (near end of module) [12 positive, 7 no change, 2 negative]
  1. The biggest change is that I am no longer afraid of it
  2. Programming isn't as difficult as i thought, it just requires a step back and some thought
  3. Using C# has shown me a different side of programming. I'm more used to the the web side of it - design, program and play, where as C# is design, program, compile, wait, re-build . . . I like it, but still prefer web development.
  4. My views have not changed
  5. No change.
  6. None really. Im still very confused with the whole thing

It was not possible to divide the responses into clear categories, because a matter like "difficulty" can be interpreted as positive or negative. When at school, the students had positive views of programmers in that they thought that they must be intelligent; but this could mean that some were discouraged from choosing the subject due to lack of confidence in their own ability.

Another issue is that the students did in fact choose to pursue computing at university, and so their views are not representative of those of typical students at school. It is notable however, that several students said that programmers were "misunderstood" by school pupils.

The question purposely prompted the students with some words that might be considered to be leading questions. For example, the words "boring" and "difficult" did recur in free-text answers as descriptions of programmers and programming. However, some other words did not (like "lucrative" and "important"). The word "clever" is in the question, and did occur in some students' answers; but so did the word "intelligent", which is a synonym, and a lot more often. There were also several new words that did not appear in the question in the form of synonyms or even antonyms, such as "fun", "enjoyable", "creative", and "problem-solvers". These observations taken together, while nothing like a formal analysis, nevertheless suggest that there was not a great overlap between the words in the the question and in the answers, and that the students were not being led to a great extent, but rather felt able to use their own words to express their views.

It is not shown which of the responses were from students who were familiar with programming or not, because the data is aggregated over the whole class. However, judging by their content, the more negative comments seem to be from pupils with some experience of learning how to program at school. There is a suggestion that they were not well taught.

Comments regarding the later time-points indicate that most students realised that programming was not so difficult to understand, but that the complexities can make it tedious, and slow to learn.

Coming up to date, the changes in opinion are mostly positive, with some that did not change, and sadly two that stayed negative, including the one student who remains entirely confused.

Some students have gained true insights about the nature of programming, suggesting that they might have previously suffered from significant misconceptions.

3.3 Questions about students' feelings toward programming

Turning to the students' own, personal attitudes toward programming, the following Likert-scale questions were about their possible anxieties at previous time-points, and about whether they like it now.

– Back then, would the thought of having to learn how to program make you anxious?

t	un	na	s-disag	disag	neut	agree	s-ag	median
s	7	1	1	9	4	12	4	agree
b	8	0	4	11	2	9	4	disag / neut

The median for **beginning** is exactly between the categories for "Neither agree nor disagree" and "Disagree", suggesting that students are not as anxious right at the beginning of the module than they were at school. This difference is not significant, however (Wilcoxon-Mann-Whitney, two-tailed: W = 521, $N_s = N_n = 30$, p = 0.2728).

Significant anxiety is clearly evident before the module began. Now that the students have learned much programming, the same question again would not be meaningful: however, they do appear to have changed their attitudes to the good:

– Do you like programming?

t	un	na	s-disag	disag	neut	agree	s-ag	median
n	8	0	2	1	4	8	15	s / agree

Nevertheless, there are still two who, whether they can now program or not, emphatically do not like or enjoy it. In fact, on checking the raw data, those two students are the same ones who, above, did not agree that they could write a simple console program.

3.4 Other questions

Some other questions included the students' experiences of learning programming, and more detail of their feelings about it, in their own words.

There is no space to detail the results from these questions here, commenting on all the answers individually. In summary, there was more talk of the difficulty of programming, but that on the other hand that it could be fun to make programs that work; also that it does not suit all personalities equally.

There was also a range of responses showing that some students came to find programming easy; and others find it still more difficult. Whether this is due to differences in personality or in aptitude is not clear.

4 Interpretation and further work

The purpose of this exploratory study was to see whether the programming students had negative attitudes towards programming, including whether it is difficult, or boring; and whether programmers themselves suffer from social stigma such as being boring, or nerds, or uncool. It was also a question whether the research method employed would be valid or useful, because asking people about their earlier attitudes are different times in the past, and comparing to their attitudes today, may be prone to confusion.

On the latter point first, we can conclude that the survey method has not failed, because trustworthy differences were indeed found between the various time-points. However, it may still be that there is some confusion about time-points, or memories may be faded, and if so then the results could only be weakened by that. Thus, we would not expect any false positives to result, but we should be aware that true differences may be harder to see, through the veil of memory. If anything, therefore, the results found by this method are an underestimate of the true effects that have occurred.

If the students on the introductory module are typical, then it will be a general problem that programming students have a wide variety of prior experiences. Contrary to its image of being difficult, programming is a skill that can be self-taught, and so there will always be a wide range of abilities in a programming module, which presents extra challenges to the teacher. In teaching to the stronger students, to keep them progressing, the weaker ones may be intimidated. Other subjects within computing do not suffer from this problem, which may help to account for why programming, in particular, is apparently difficult to teach.

Students certainly appear to have improved in their programming skills, both before they began their first serious programming module, and of course during it. We may infer this from their assessments of their own skills, which is a form of confidence in their abilities rather than direct evidence of them. The increase in (self-efficacy regarding) their skill was very significant, statistically speaking. To that extent at least, the module was a definite success; but there were also two students who still lack self-confidence. It turned out that they were the same two students who declared at the end of the module that they definitely do not like programming. At this stage it is too soon to be able to correlate their answers with their performance in the module, because those results are not ready yet. In future work, however, it would be valuable to examine the relations between affective attitudes and performance.

The affective attitudes that were probed in this study, by Likert-scale, included whether programmers were "cool" or not, and whether the prospect of learning how to program made the students anxious or not. A minority of students agreed that programmers were cool, with many taking no view; but there was also a significant minority that *disagreed* that they were cool. It appears that their views have changed to the good, so that now none of them strongly disagree, and more of them agree than did before, when at school. However, the difference was not statistically significant.

The questions about anxiety also appear to show an improvement between school and beginning the programming module at university, but this difference was not statistically significant either. The evident anxiety about learning to program is still a concern: it consumes the majority of the class, and it does not go away until the module is underway (if then). By the end of the module, however, nearly all the students seem to like programming, and most of them strongly agree with that statement. That is very encouraging, and may show that the students' fears were ungrounded: but this now begs the question as to where those fears sprang from.

In the free-text answers, students expressed their attitudes towards programmers and programming, showing a range of opinion from positive to negative. Again, it is a concern that computing students should harbour any such negative attitudes. However, it may be a normal state of affairs amongst students to be anxious about their learning goals. In further work it would be useful to scan the literature for research that could throw light on that issue, and then look to see if the situation is worse in computer programming than in other subjects.

Evidence from the free-text questions fleshed out the Likert-scale type questions a great deal. The range of responses from novice programmers showed some good depth of learning, and some impressive insights for young people with little experience. The affective content of their answers may have been cued by the affectively laden words in the question, in some cases, but in general the answers showed an independent turn of thought, and of word choices. The students can probably be trusted when they claim to find programming to be difficult but fun; or programmers to be misunderstood, or intelligent (or nerds). Some found programming boring, overly complex, or confusing. Even at the end of the module there were still some students with significantly negative attitudes, who dislike programming. They were only two out of the full 38, but one could say that is two too many.

One possible conclusion could be that the teaching on the module was just not good enough, and two or more students were not reached. On the other hand, it may be that there always are a few students on every course who do not like the subject, and programming might not be exceptional in this. Also, just because a student dislikes a subject does not mean that he is bad at it in general or that he would let it stop him from mastering the discipline. Further work could explore this issue, too. However, in this study at least, the two students who dislike programming at the end are the same ones who have weak self-confidence in their programming ability, and so we may hypothesise that negative affect correlates with lower ability. Whether this correlation holds up would be another matter for further work; along with which factor is a cause and which the effect.

Another possible conclusion could be that some students are simply bad programmers, and always will be, because they don't have the necessary mentality. For example, they might not be able to deal with abstractions well. Such a hypothesis is tempting for those teachers who may be reluctant to examine their own teaching, but there is no evidence in this study to support it.

.

In conclusion, this small study shows that negative affect is indeed a common and worrying factor in the psychology of student programmers.

The *effects* are not clear, but there may be impacts on performance when at university; and on career choice at or before university. A general stigma that programmers are "nerds" could discourage students from choosing computing subjects, but it appears that certain types of personality will tend to be attracted to programming even so. It may well be that some people have not only a preference but also a definite aptitude or talent for programming, and these are the ones who will really enjoy it.

However, even amongst students who have chosen computing for a career, it is deeply concerning that a proportion of them do not lose their fear of programming. They may seek refuge in other areas of computing, but will in that case always suffer a considerable disadvantage in their careers.

The *causes* of the negative affect that has been observed in this study are as mysterious as its effects, though one can speculate. Students may have acquired their attitudes from society at large, from Hollywood films, from friends and family, or even from their own computing teachers. Sometimes students complain of poor teaching at school, and in this study too there were a few comments suggesting that. It is credible, too, given that schools may easily confuse computing (where programming is a core skill) with information technology (where it is nothing of the kind).

However, there is no hard evidence in this study to suggest that the way school teachers teach how to program is any worse than, say, the way university lecturers (such as I am) do it. On the contrary, there was evidence in the class of this study that that a few students were not helped by their teacher (me) at all; in fact their state might have been made worse by him, if they have been turned away from programming as a result.

Whether at school or university, then, it is suggested by this study that poor teaching may be a significant factor in making pupils and students feel bad about programming. It is not an easy conclusion to accept, but it will be harder still to find out how it is failing. It is not likely to be the fault of programming teachers specifically, for they teach other subjects as well or better. Rather, we might blame the subject itself, and conclude that programming is simply difficult to teach. In that case, further research may help us to diagnose why that should be, and then we could do something about it at last.

5 Acknowledgements

Thank you to the students who answered questions about their attitudes, and to Thomas Green, John Richards, and the other participants at the PPIG-wip in Dundee for their comments. Warm thanks also to the reviewers of this paper, who took pains to make more suggestions than I could respond to in time for publication.

References

Bandura, A. (1994). Self-efficacy. In V. S. Ramachaudran (Ed.), *Encyclopedia of human behavior* (Vol. 4, p. 71-81). Academic Press.

Beckwith, L., & Burnett, M. (2004). Gender: An important factor in end-user programming environments? In *Ieee symp. visual languages and human-centric computing (vl/hcc'04)* (pp. 107–114).

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., et al. (2006). Tinkering and gender in end-user programmers' debugging. In *CHI 2006.*

Bennedsen, J., & Caspersen, M. (2008). Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science. *Computer Science Education*, *18*(1), 1–19.

Busch, T. (1995). Gender differences in self-efficacy and attitudes toward computers. *J. Educational Computing Research*, *12*, 147–158.

Zeldin, A., & Pajares, F. (2000). Against the odds: Self-efficacy beliefs of women in mathematical, scientific, and technological careers. *American Educational Research Journal*, *37*(1), 215–246.

Characterizing Comprehension of Concurrency Concepts

Zhen Li Zhe Zhao Eileen Kraemer

Computer Science Department
University of Georgia
{zhen, zhe, eileen}@cs.uga.edu

Keywords: Concurrency, Software Visualization, Empirical study, Misconceptions

Abstract

A comprehensive understanding of students' common difficulties in understanding synchronization and concurrency is a prerequisite for developing tools and educational materials to alleviate these difficulties. In this paper we briefly present a study through which we identified students' misconceptions about concurrency and synchronization, categorized their misunderstandings into a misconception pyramid, and built subject profiles through which we were able to discover the nature and frequency of the misconceptions exhibited by the students in this study. Based on these findings, we developed metrics to capture the breadth and severity of individual subject's misconceptions. We describe these metrics and show how they correlate with other measurements of understanding of concurrency and synchronization.

1. Introduction

As early as 1986, researchers worried that "the complexity of (concurrent) programming--all those processes active at once, all those bits zinging around in every direction--is simply too great for the average programmer to bear" [Gelertner1986]. Today, it is generally agreed that multi-threaded programs are difficult to design and comprehend, and that concurrency and synchronization concepts are difficult for students to master [1, 2, 3]. We believe that the development and use of appropriate external representations has the potential to help students better comprehend the dynamic and non-deterministic nature of these programs. However, to properly design and evaluate such representations, we must develop a detailed understanding of what aspects of these concepts students find difficult and what misconceptions they harbor. Prior work by our group [4, 9] and by others [3, 7, 10, 11] provides some insight.

We conducted a new study that sought to obtain detailed information about the reasoning processes that students engage in when dealing with concurrent software. We analyzed student responses, identified misconceptions, and then categorized these into a "misconception pyramid." We then constructed per-subject profiles that captured the nature and frequency of misconceptions exhibited by each student, and developed metrics that we believe capture the breadth and severity of misconceptions held by a particular subject. In this paper, we briefly describe our study, our analysis, and the misconception pyramid and define the metrics for breadth and severity of misconceptions. We present the most common misconceptions in the sample group and explore the correlation of our proposed metrics with other measures of comprehension of concurrency and synchronization concepts. Finally, we propose new diagrams to aid in the comprehension of concurrent program executions, and future studies to further evaluate and refine this work.

2. Related Work

In the early 1990s, Resnick[7] recognized that realizing the potential benefit of concurrent programming would depend on the ability of people to effectively learn, use and understand concurrent programming constructs and languages. He developed a concurrent extension to Logo (MultiLogo) and conducted an experiment with a group of elementary school students who used MultiLogo to control simple robots built from LEGO bricks. He then evaluated their work and found three types of bugs: problem-decomposition bugs, synchronization bugs, and object-oriented bugs.

While he believed that object-oriented bugs might have been due to aspects of Multi-Logo, he suggested that difficulties inherent to thinking about concurrency were at the root of the problem-decomposition and synchronization bugs.

Kolikant performed empirical studies of students learning about concurrency [3]. Her results show that students develop pattern-based techniques to solve synchronization problems and then have trouble in solving non-familiar synchronization problems, perhaps as a result of their reliance on those pattern-based approaches. She found that student misconceptions were often the source of their difficulties, writing "we were able to uncover reasonable, yet faulty connections that many students had made ... these connections were the source of their difficulties."

Fleming, et al. [9] performed a think-aloud study of students in a graduate-level computer science class to study the strategies that students apply in corrective maintenance of concurrent software. He collected think-aloud and action protocols, and annotated the protocols for certain behaviors and maintenance strategies. He looked at whether study participants performed diagnostic executions of the program and whether they engaged in failure trace modeling (modeling how the system transits among various internal states, at least one of which is a clear error state, up to the point of failure). He found two key attributes of the most successful participants: they detected a violation of a concurrent-programming idiom and they constructed detailed behavioral models of execution scenarios.

Xie, *et al*. [4] performed an instructor survey and observational study and identified a core set of difficulties that students encounter in learning about concurrency. Common problems he identified included: 1) Thread inter-leavings are difficult for students to comprehend.; 2) Students often forget that context switches can happen when the thread is in a monitor or critical section and have trouble correctly applying that knowledge when they do remember; and 3) Students have trouble reasoning about why the implementations of synchronization primitives lead to correct synchronization behavior.

Recently, Armoni and Ben-Ari [11] performed an in-depth survey of the concurrency-related concept of non-determinism, how it is defined and used, and how it has been taught. They present a taxonomy of the ways that non-determinism can be defined and used, the categories of which are domain, nature, implementation, consistency, execution and semantics. Their survey of educational materials and practices on this topic leads them to the conclusion that "the treatment of non-determinism is generally fragmentary and unsystematic," and they go on to suggest various strategies for teaching non-determinism in the CS curriculum.

Lu, *et al*. [10] studied real-world concurrency bugs rather than student behavior or reasoning. They looked at four open-source applications and randomly selected 105 real world concurrency bugs. They found that one-third of the non-deadlock bugs involved violations of the programmer's intended order of operation, and that another one-third of the non-deadlock concurrency bugs involved multiple variables. In examining the bug-tracking records, they also found that many of the fixes to the bugs they studied were not correct at the first try, providing further support for the idea that reasoning about concurrent executions is difficult.

Each of the above studies attempts to gain insight into the question of what students and programmers find difficult in learning about and in managing concurrency and synchronization. Our study is most similar to that of Kolikant, in that we attempt to identify both the difficulties that students encounter and the reason for those difficulties. We get at this information by not only asking study participants to answer questions that evaluate their comprehension of the potential behaviors of a concurrent program execution, but by also asking them to explain their reasoning. It is in these explanations that we gain insight into their understanding of the meanings of concurrency-related terms, their mental models of the relationships among the objects and constructs by which concurrency and synchronization are achieved, and their comprehension of the consequences of thread activities and interactions.

Another element of our work is the evaluation of diagrams designed to support comprehension of multi-threaded program executions. Related work describes concurrency-related aspects of UML diagrams, proposes variations on UML diagrams to better support concurrency, or evaluates UML diagrams or their variations. For example, Schader and Korthaus described features of UML that support the representation of concurrency [12]. Mehner and Wagner [13, 14] added shading conventions on activations to indicate when, and within which activation, threads are ready or running. Xie, *et al.* [4, 15] developed an extension to sequence diagrams that uses colored activations to indicate the state of each thread (i.e., running, blocking, or ready), among other features. Most recently, Fleming [16] proposes a variation on UML sequence diagrams in which hatching of the activation bar denotes thread state and object states denote the effects of operations on mutexes and condition variables.

3. Experiment

The overall goal of our experiment was to compare the use of different types of UML diagrams (UML 2.0 sequence diagrams and UML 2.0 state diagrams) for different tasks related to the comprehension, implementation, and debugging of concurrent software. The participants were fifteen Computer Science students drawn from upper-level undergraduate classes and from graduate classes during the spring semester of 2010. Students were volunteers and were paid $50 for their time. The study materials included a demographic survey, six computer-based training modules, five pre-tests (one quiz for each of the first five training modules), and a post-test. Part I of the post-test comprised 24 comprehension questions that involved reasoning about what could happen next in a particular execution scenario. Part II questions involved identifying errors, evaluating and creating models and diagrams, and writing code.

In this paper we provide a detailed analysis of participant explanations of their answers to Part I questions, in which they were asked to supply both a yes/no answer to whether a particular set of program events could occur next and in the stated order, and also to explain their reasoning. These explanations of student reasoning provided the basis for our identification of misconceptions. Questions in this part of the post-test were based on the "Single-lane Bridge" problem. The problem states that a bridge over a river is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any one time. To simplify this problem, we define the cars that move from left to right as red cars and those that move from right to left as blue cars. To avoid a safety violation, only one kind of car is allowed to be on the bridge at a time. Cars exit the bridge in the order in which they entered and the leading car may exit the bridge at any time. We structure this system so that each colour of car is implemented as a thread, and the shared bridge object is implemented as a monitor with two associated condition variables **okToEnter** and **okToExit.** The basic functions for entering and exiting the bridge are *redEnter(), redExit(), blueEnter()* and *blueExit()*. We assume a C++ implementation using the *pthreads* library, in which explicit calls to lock() and unlock() are invoked on mutex locks. Then for each of the given scenarios, we asked whether a particular event sequence could happen next.

4. Analysis

Although Part I of the post-test consisted of objective questions, we initially found it difficult to evaluate the responses in a way that accurately reflected the students' understanding of the system. Consider question 1.b, shown in Figure 1 and describing a scenario in which two threads, **redCar1** and **redCar2,** exist in the system. Thread **redCar1** invokes the *redEnter()* method and has already returned when a context switch occurs and the **redCar2** thread begins to run. One of the sub-questions asks whether it is now possible for the **redCar2** thread to invoke the *redEnter()* method and block on the monitor lock. The answer to this question should be NO. Only two threads exist in the system and **redCar1** should have released the monitor lock before it returned from the *redEnter()* method. Thus, it is not possible for **redCar2** to block on the monitor lock.

```
1.  Suppose that only two threads exist in the system: redCar1 and redCar2. Suppose further
    that redCar1 has invoked the redEnter() method, and has returned. A context switch occurs
    and the redCar2 thread starts to run.

Could the following event sequence happen next?  Circle YES if the sequence is possible;
otherwise, circle NO.  Then please provide a brief explanation of your reasoning.

(b) redCar2 invokes redEnter(), then blocks on the monitor lock.
    YES       NO
```

Figure 1 -- Question 1.b

In answering this question, 9 out of 15 subjects chose the correct answer (NO). However, in looking closely at their explanations, we found that 7 of them thought that the monitor lock would only block blue car threads and regarded the monitor lock in the question as an **okToEnter** condition variable. One of them misunderstood the meaning of the term "block" as "own" or "has" and thought that **redCar1** already owned the monitor lock since it was on the bridge and that **redCar2** could thus not own the same lock. Another student, however, did not understand the question and thought that **redCar2** should not "block" on the monitor lock but lock the monitor lock. Thus, by reading the explanations given by the students we found that actually none of the 9 students who gave the correct answer really understood the monitor lock and its mechanism.

We also found that although each question was designed to test some specific misconceptions, a failure in one particular question might not actually stem from the misconception the question intended to examine. Instead, the failure might be rooted in some other misconceptions. We found further that some misconceptions could cause general failures in reasoning about many different scenarios. Consider questions 4.d and 4.e as an example (Figure 2).

```
4.  Suppose that only three threads exist in the system: redCar1, redCar2 and blueCar1.
    Suppose further that redCar1 is running and has just invoked the redEnter() method and the
    redEnter() method has returned. A context switch occurs and the redCar2 thread begins
    running and invokes the redEnter() method. redCar2's invocation of the redEnter() method
    has not returned.

Which of the following event sequences could happen next?  Circle YES if the sequence is
possible; otherwise, circle NO.  Then please provide a brief explanation of your reasoning.

(d) A context switch occurs, and the redCar1 thread begins to run. redCar1 then invokes
    redExit() and this invocation returns.
    YES       NO

(e) A context switch occurs, the redCar1 thread begins to run. redCar1 then invokes the
    redExit() method and blocks on the monitor lock.
    YES       NO
```

Figure 2 -- Question 4.d and 4.e

These two questions are aimed at testing the subjects' ability to consider multiple possible interleavings in an execution. The answer to both of the questions should be YES since the question only describes that **redCar2**'s invocation of the *redEnter()* method is interrupted by a context switch but does not mention whether **redCar2** holds the monitor lock or not when interrupted. Three possible interleavings exist here. One is that **redCar2** has invoked the method but has not yet obtained the monitor lock. The second is that **redCar2** invoked the method, holds the monitor lock and has not yet released it. Another possibility is that **redCar2** has already released the monitor but not yet returned from the *redEnter()* method. The first and the third situations could lead to event sequences described in 4.d and the second situation could lead to event sequences described in 4.e.

Organizing students' answers to these two questions, we have the following table (Table 1).

	4.d	4.e	Subjects
1	YES	YES	102, 139, 132
2	YES	NO	108, 109, 113, 122, 126,138, 141, 142, 145
3	NO	NO	110, 119
4	YES	No answer	128

Table 1 – Subjects' Answers to Questions 4.d and 4.e

Apparently, most of the students were not able to answer both of these questions correctly and the majority failed on question 4.e. However, by looking closely at their explanations, we found the reason for the failure does not truly stem from students' inability to consider the possible interleaving, as expected. Actually, all 9 subjects failed in 4.e because of misconceptions about the monitor lock. Some of them confused it with the **okToExit** or **okToEnter** condition variables. Others were ignorant of the mechanism of the monitor lock so they succeeded in question 4.d, which does not deal with the monitor lock concept but failed in 4.e. Also worth noting is that most students reasoning about these two questions was based on "story-level" understandings, as seen in explanations such as "**redCar1** is free to exit" or "nothing blocks **redCar1** to exit", etc. Actually, none of them considered the event sequence at the implementation level, which again highlights their misconceptions of the context switch and its properties.

Thus, we found students' misconceptions about concurrency and synchronization cannot be captured in a simple list of confusions or misunderstandings of concepts, terminologies and mechanisms. Rather, they are correlated with one another, interacting in a seemingly hierarchical architecture so that it is not possible to examine higher level misconceptions without first teasing out the impact of lower-level misconceptions, or ensuring that participants first have a firm grasp of lower level concepts. In other words, to understand higher level concepts, students must first rid themselves of lower level misunderstandings.

4.1 Misconception Pyramid

We introduce a misconception pyramid (Figure 3), which captures common misunderstandings that students exhibited when reasoning about a concurrent system, and the hierarchical structure of the misconceptions according to the difficulty and dependency relations of understanding the concepts in that level. Understanding concepts at higher levels of the pyramid requires an understanding of the concepts at lower levels first. Descriptions of the types of misconceptions one might find at each level are presented in Table 2, which was constructed based on misconceptions identified in the literature and also those that we encountered in our analysis of subjects' explanations of their reasoning in this study.

The bottom level of the pyramid is the description level and includes misconceptions such as misunderstanding of the requirements, constraints and other details of a concurrent system at the level of the "story" about the red cars and blue cars. For example, some subjects wrote explanations such as "**redCar2** should wait for **redCar1** to invoke *redEnter()* method first" or "**redCar1** should block the bridge first" demonstrate one common misconception at this level: that the thread labels **redCar1** and **redCar2** were the actual running order of the threads.

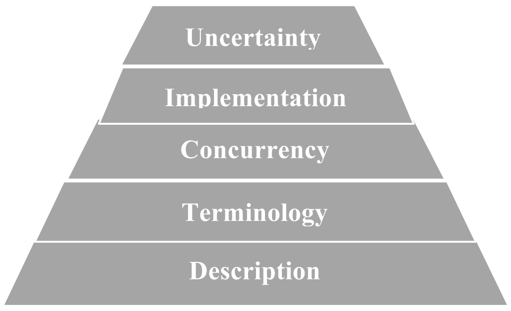


Figure 3 -- Pyramid of Misconceptions

The next level of the pyramid includes misconceptions related to terminology we used in describing concurrent scenarios. A typical example is the misunderstanding of the meaning of "block on" a conditional variable/monitor lock as "hold/own" a conditional variable/monitor lock. This kind of misconception can be seen throughout the explanations given by subjects in our study. Most students who held this kind of misconception did so consistently, causing them to fail on a particular group of questions. Typical students' explanations that illustrate this level of misconception include but are not limited to "**okToEnter** is already blocked" or "monitor is already blocked by **redCar2**".

The third level of the pyramid is the concurrency level, which includes misconceptions about basic thread behaviors such as context switching and the thread life cycle. For example, some students seemed to think that a context switch could not happen while a thread was executing in a critical section and many students thought that a context switch is not allowed during the execution of a method and regarded the whole method body as uninterruptible. Some typical students' explanations are "**redCar2** should receive return call then switch out" or "because **redCar2** has not done its activity (so it cannot be context switched out)".

Description Level	
D1	Misconceptions of system and/or problem descriptions
Terminology Level	
T1	Misconceptions of the meaning of "invoke/call" a method
T2	Misconceptions of the meaning of "return" from a method/invocation
T3	Misconceptions of "block" on a monitor lock as "hold/has" a monitor lock
T4	Misconceptions of "block" on a conditional variable as "hold/has" a conditional variable
Concurrency Level (thread behavior)	
C1	Misconceptions about context switching
C2	Misconceptions about the thread life cycle
Implementation Level	
I1	Misconceptions about conditional variables and the wait/signal mechanism
I2	Misconceptions about monitor lock
I3	Misconceptions about block and unblock mechanism
Uncertainty Level	
U1	Confused about space of executions and thread interleavings

Table 2: Misconception Pyramid Table

```
Invoke
……
Lock the monitor lock
    Check conditional variables
        Access and modify shared variable
        ……
Release monitor lock
    Signal on conditional variables
……
Return
```

Figure 4 – Basic Monitor Programming Function Structure

The fourth level of the pyramid is the implementation level, which is related to detailed implementation mechanisms such as the monitor lock and condition variables and their functionalities. By investigating the subjects' answers and explanations in our study, we found that few subjects were clear on the basic monitor programming structure shown in figure 4. We believe that this is greatly related to students' misunderstandings in the three previous levels. If students do not understand the context switch, they are not able to appreciate the actual purpose and corresponding mechanism of the monitor lock. Misunderstandings of different terminologies also lead to confusion about the workings of monitor programming structures and functions.

The top level of the pyramid is concerned with failures in dealing with uncertainty; that is, the inability to envision or manage all the possible threads interleavings and execution scenarios. While this problem is often cited as the main source of difficulty in the comprehension of concurrent program executions, we found that this level of difficulty was not seen in our study, as students tended to fail much earlier in the pyramid, and thus were not even exposed to these higher-level issues. Whether a detailed investigation of participant reasoning processes would find the same to be true in other studies of comprehension of concurrent program executions is an open question.

An alternative representation of the pyramid might combine the two lower levels them into a single level, in which Description and Terminology sit side-by-side, supporting the Concurrency level. Further, another approach to layering might think of the top layer of the pyramid, which we term "Uncertainty" as dealing with dynamic analysis issues, and the second layer of the pyramid as dealing with static analysis issues, with both layers together dealing with implementation-related issues.

4.2 Subject Profile

Next, we introduce the subject profiles shown in Table 3. These subject profiles reflect the types and frequency of occurrence of each subject's misconceptions. The first column of the table indicates the subjects' ID number. The other columns correspond to items in the misconception pyramid table. Each cell of (subject, item) is the number of (answer, explanation) pairs of that subject that demonstrate the corresponding type of misconception.

Subject	D1	T1	T2	T3	T4	**C1**	C2	**I1**	**I2**	I3	U1	Total
102	2	1	2	1	2	1	1	5	9	0	0	24
108	3	2	1	0	0	3	1	3	10	0	0	23
109	3	0	0	0	0	8	1	0	11	0	0	23
110	7	3	4	4	2	2	1	8	9	0	0	40
113	2	2	1	1	1	6	1	12	11	0	0	37
119	1	0	4	0	2	4	1	8	11	0	0	31
122	0	0	4	0	0	1	0	0	9	0	0	14
126	0	7	0	0	0	2	1	4	14	0	0	28
128	NA											
132												
138	1	4	0	0	1	8	1	2	9	0	0	26
139	1	4	7	1	2	9	1	7	9	0	0	41
141	2	4	5	2	6	4	1	7	9	0	0	40
142	0	0	1	2	0	1	0	3	10	0	0	17
145	0	0	0	0	0	1	1	1	14	0	0	17
Avg	3	3	4	2	2	**7**	2	**9**	**19**	0	0	

Table 3: Subject Profile Table

While 13 of the 15 subjects provided sufficient explanations for us to build profiles, 2 out of 15 (subjects 128 and 132) provided almost no explanations for their answers, which made it impossible to evaluate their misconceptions. Perhaps the most noticeable characteristic of the subject profile is that no misconceptions of items **I3** or **U1** are found, but that subjects show a very high frequency in demonstrating misconceptions in **I1** and **I2**. This reinforces the idea that students' misconceptions form a hierarchical structure in which lower level failures not only cause higher level misconceptions but also isolate students from higher level concepts.

Another interesting characteristic of the subject profile is that the most common misconceptions are **I2, I1** and **C1, which** are misconceptions about monitor locks, condition variables and context switching. Causality relations exist among these misconceptions; for example, a subject's incomplete understanding of when and how a context switch could occur causes their misunderstanding of the functionality and mechanism of monitor lock, which thereafter causes them to confuse monitor lock with condition variable. We plan to conduct additional studies to further explore the validity of this idea.

Based on the collected data, we can make some statements about particular subject's comprehension of concurrency. For example, we could generalize that subject 139 is almost ignorant of concurrency concepts and synchronization mechanisms since he demonstrated all kinds of misconceptions at different levels, while subject 122, who just showed consistent misconceptions in a limited range of items, apparently has a much better comprehension of concurrency. This is also validated by the Part I scores of these two subjects, as seen in Table 5 and illustrated in figure 8.

4.3 Subject Evaluation

Although a subject profile allows us to characterize both a single subject's understanding of concurrent systems and the whole subject sample, we introduce two metrics to better quantify the evaluation. One is the breadth of range of misconceptions (denoted as Metric B) and the other is the weighted severity of misconceptions (denoted as Metric S).

$$B_{subject} = \frac{count\ ((subject, item) > 0)}{count\ ((subject, item))} \qquad S_{subject} = \sum_{items} (subject, item) * W_{item}$$

Figure 5 – Evaluation Metrics

Metric B for a single subject is the percentage of misconceptions the subject has regarding the whole pyramid of misconceptions, as illustrated in figure 5. For example, subject 122 exhibited misconceptions in 3 of 11 categories, so B_{122} = 3/11 or 0.27, while subject 139 exhibited misconceptions in 9 of 11 categories for B_{139} = 9/11 or 0.82. With metric B we are able to evaluate how many different misconceptions a particular subject has. A larger B illustrates more widely spread misconceptions of a particular subject.

Level 0: Description Level		
D1	0.3	
Level 1: Terminology Level		
T1	0.268	0.067
T2		0.067
T3		0.067
T4		0.067
Level 2: Concurrency Level		
C1	0.2	0.1
C2		0.1
Level 3: Implementation Level		
I1	0.135	0.045
I2		0.045
I3		0.045
Level 4: Uncertainty Level		
U1	0.097	

Table 4: Misconception Item Weight Table

Subject	Part1	Metric B	Metric S
102	18	0.82	1.832
108	24	0.64	2.086
109	25	0.36	2.295
110	15	0.82	4.036
113	24	0.82	2.67
119	19	0.64	2.057
122	29	0.27	0.773
126	21	0.45	1.579
128	19	N/A	N/A
132	23	N/A	N/A
138	29	0.64	2.03
139	11	0.82	2.958
141	24	0.82	2.959
142	27	0.45	0.886
145	24	0.36	0.875

Table 5: Subject Performance Table

The S metric, however, evaluates misconceptions on another dimension. It is designed to characterize the severity of single subject's misconceptions. Thus, to compute the S metric, we must first assign a weight to each misconception item. As we pointed out before, lower level misconceptions are likely to cause higher level misconceptions. Also, lower level misconceptions impede a subject's understanding of a system more than higher level misconceptions do. Therefore, we simply use an inverse ratio of the level to assign weights. Table 4 illustrates how the weights are assigned.

Therefore, the metric S can be calculated as the expected value of severity of different misconception items according to formula illustrated in figure 5, in which W_{item} is the weight of the corresponding misconception item. Applying these two metrics to subjects in our study, we get the subject performance table (Table 5). For example, subject 122 exhibited 4 misconceptions of type T2 (w=0.067), 1 misconception of type C1 (w = 0.1), and 9 misconceptions of type I2 (w = 0.045). S_{122} is thus 4 * 0.067 + 1 * 0.1 + 9 * 0.045 = 0.773.

To illustrate the validity of these two metrics, Metric B and Metric S, we explore the correlation between these values and students' total score of Part I in the post-test.

Figure 6 illustrates the correlation between metric B and the score of part I. As we see, although the high scores are not strictly determined by metric B, the metric characterizes how poorly a student may perform in reasoning about concurrency and synchronization scenarios, and overall shows the expected negative correlation (the greater the breadth of misconceptions, the lower the score).

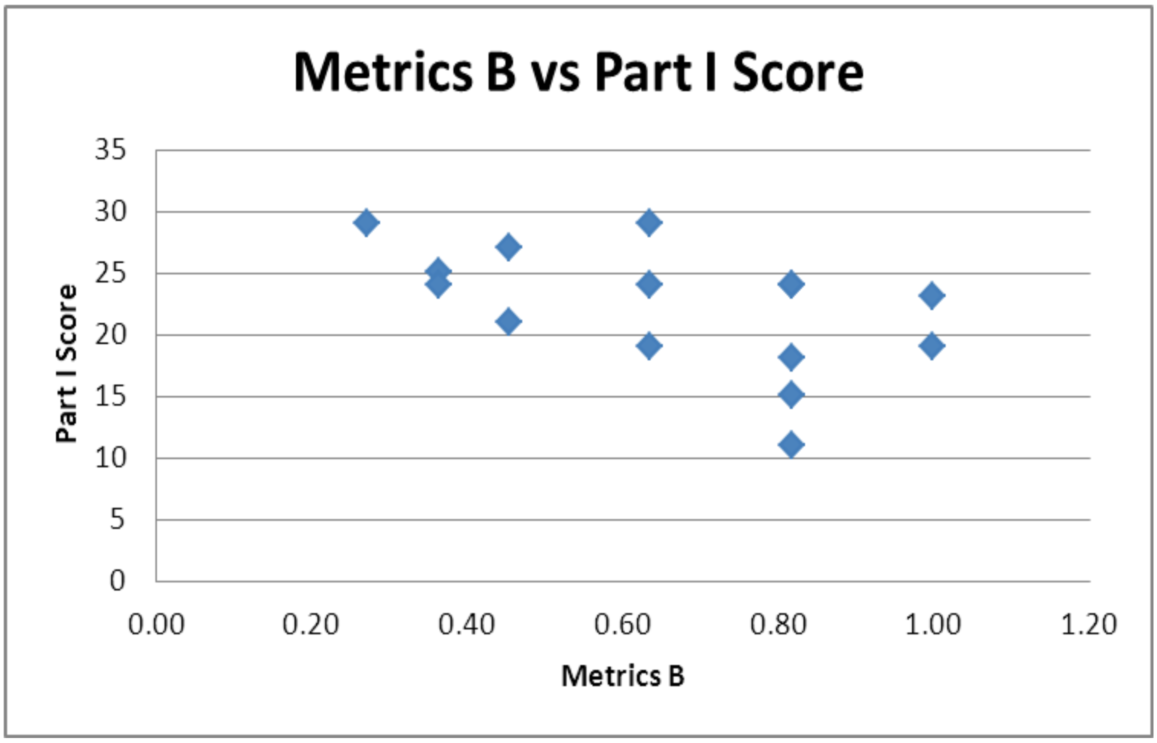


Figure 6 – Correlation between Metric B and Part I Score, Pearson correlation = -0.527

Figure 7 illustrates the correlation between metric S and the score of part I. Unlike metric B, the metrics S seems to have a better (negative) correlation with score when metric S is small. As metric S becomes large, the correlation becomes random. This is reasonable, since when a subject has no idea of a concept in concurrency, they tend to reason about the corresponding scenario based on understanding of one possible sequential execution, which randomly coincides with the actual execution sequence under concurrency.

By regarding metric B and metric S as two orthogonal vectors that characterize an individual subject's misconceptions in concurrency and viewing the origin point in a coordinate system as an ideal expert who does not demonstrate any misconceptions in understanding a concurrent system, we are able to calculate the Euclidian distance of a particular subject from the ideal expert. This Euclidian distance may be regarded as a combination of metric B and metric S. In figure 8, we plot this new evaluation with the total score of part I for every subject. Regardless of the two subjects, number 132 and number 128, who did not given enough clues for us to conclude their misconceptions, other subjects tend to form a reverse correlation of their Part I score and the Euclidian distance from an ideal expert.

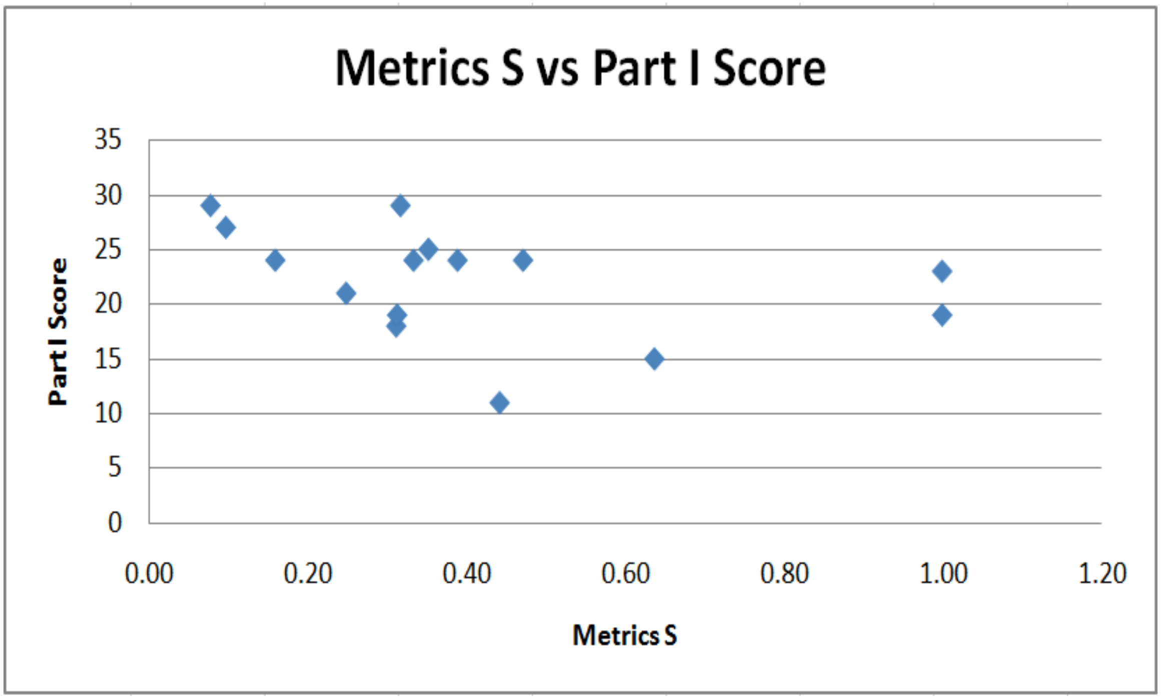


Figure 7 – Correlation between Metric S and Part I Score, Pearson correlation = -0.386

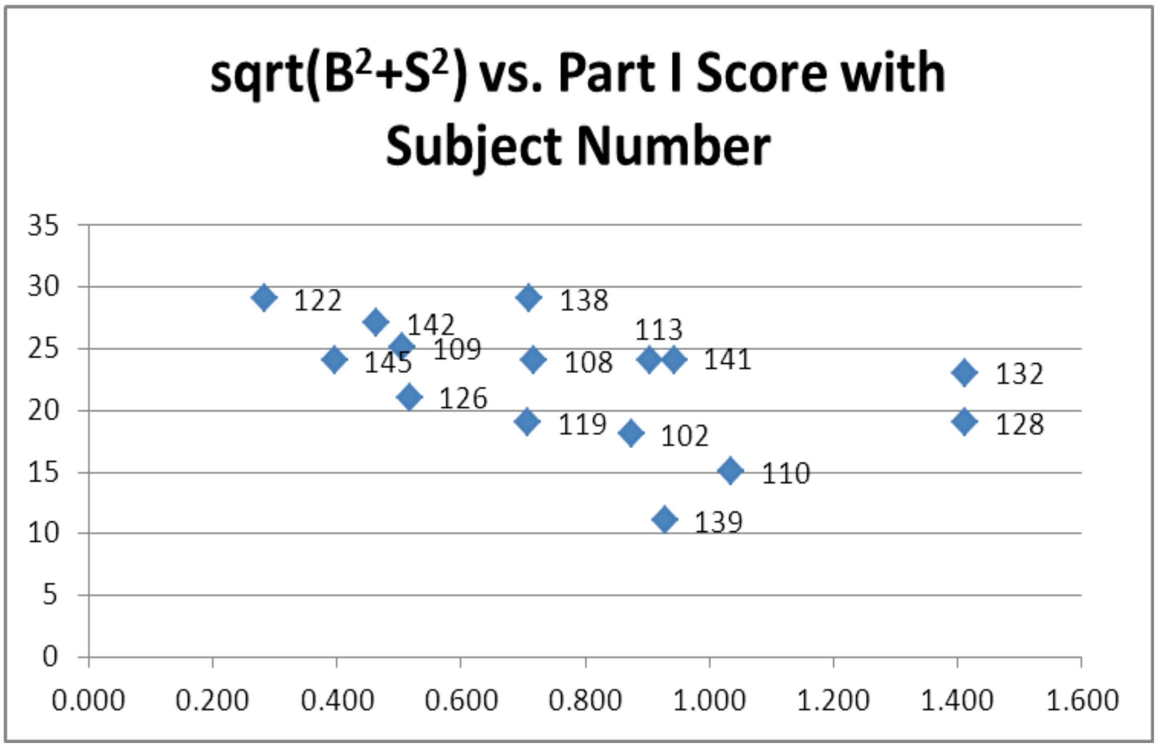


Figure 8 – Correlation between sqrt(B^2+S^2) and Part I Score, with Subject Number

.

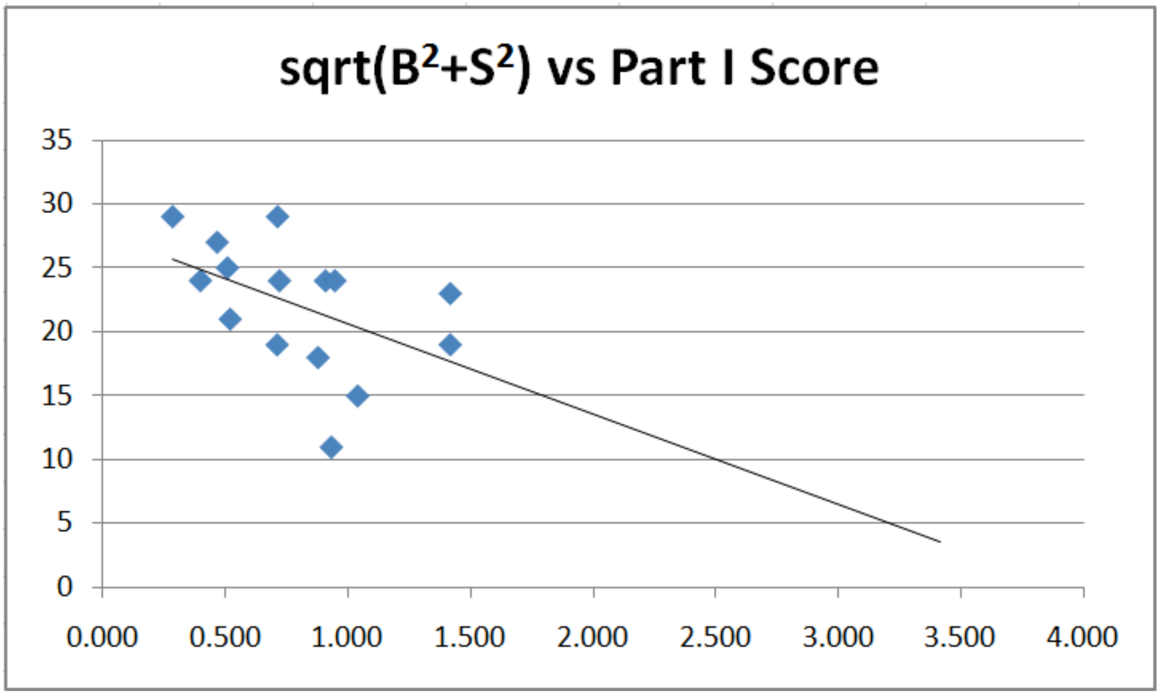


Figure 9 – Correlation between sqrt(B^2+S^2) and Part I Score with Linear Prediction, Pearson's correlation = -0.476

In figure 9, we plot the linear prediction of data in figure 8, which illustrates an expected inverse correlation between the evaluation of subjects' misconception and the actual performance of a subject.

Overall, we believe that metric B and metric S do a reasonable job of capturing the breadth and severity of misconceptions exhibited by individuals or by a group of individuals. The calculation of such metrics and the use of the misconception pyramid have the ability to guide instructors in assessing whether a concept or group of concepts has been sufficiently mastered by a student or class of students. The structure of the pyramid provides some insight into the order in which these concepts might be taught and suggests that intermediate evaluations be performed before moving on to higher-level concepts.

6. Conclusions and Future Work

We have presented here an initial analysis of a relatively small study of students engaged in reasoning about the execution of multi-threaded programs. We have identified a number of misconceptions exhibited by study participants, and based on these findings, have proposed a hierarchical structure of misconceptions, and metrics for evaluating the breadth and severity of these misconceptions. We present arguments to support the validity of the hierarchy and of the metrics. We propose to conduct additional studies with larger groups, to further evaluate both the pyramid and the metrics, and to further flesh out the ways that students think and learn about concurrency and synchronization.

7. References

[1] S. Carr, J. Mayo, and C.K. Shene, "ThreadMentor: A pedagogical tool for multithreaded programming", *Journal of Educational Resources in Computing*, 3(1), 2003.

[2] Cunha, J. C. and Lourenço, J., “An integrated course on parallel and distributed processing” *In Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education,* Atlanta, GA, 1998.

[3] Kolikant, Y. B.-D., “Learning concurrency: evolution of students’ understanding of synchronization,” *International Journal of Human-Computer Studies*, 60(2), 243–268, 2004.

[4] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In Proc. ICSE, pages 727–731, 2007.

[5] T.R.G.Green, M. Petre. “Usability Analysis of Visual Programming Environments: a ‘cognitive dimensions’ framework,” *Journal of Visual Languages and Computing*, 7(2), 131-174, 1996.

[6] Maria Kutar, Carol Britton and Trevor Barker. “A Comparison of Empirical Study and Cognitive Dimensions Analysis in the Evaluation of UML Diagrams.” In J.Kuljis, L. Baldwin & R. Scoble (Eds). *Proc. PPIG 14*, June 2002.

[7] Mitchel Resnick, “MultiLogo: A Study of Children and Concurrent Programming,” *Interactive Learning Environments*, 1(3), 153-170, 1990.

[8] Gelertner, D. “Domesticating Parallelism, “ *Computer*, 19(8), 1986.

[9] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In Proc. 30th Int. Conf.Software Eng. (ICSE 2008), pages 759–768, 2008

[10] Lu, S., Park, S., Seo, E., and Zhou, Y. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News* 36, 1 (Mar. 2008), 329-339. DOI= http://doi.acm.org/10.1145/1353534.1346323

[11] Armoni, M. and Ben-Ari, M. 2009. The concept of nondeterminism: its development and implications for teaching. *SIGCSE Bull.* 41, 2 (Jun. 2009), 141-160. DOI= http://doi.acm.org/10.1145/1595453.1595495

[12] M. Schader and A. Korthaus. Modeling Java Threads in UML. In The Unified Modeling Language: Technical Aspects and Applications, pages 122–143. Physica, 1998.

[13] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In Software Visualization, pages 163–175. 2002.

[14] K. Mehner and A. Wagner. Visualizing the synchronization of Java-Threads with UML. In Proc. VL, pages 199–206, 2000.

[15] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In Proc. ICPC, pages 123–134, 2007.

[16] Scott D. Fleming, Eileen Kraemer, R.E.K. Stirewalt, and Laura K. Dillon. Debugging Concurrent Software: The Importance of External Representations. To appear, VLHCC10.

The Construction of the Concept of Binary Search Algorithm

Sylvia da Rosa

Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
darosa@fing.edu.uy

Abstract. The general goal of this paper is to describe how Piaget's theory-*Genetic Epistemology*-can contribute to computer science education research. Using, as an example, students' understanding of the binary search algorithm, this paper illustrates how theoretical principles are applied to identify some obstacles that students face while solving an instance of that algorithm and to help students to surmount these obstacles. The methodology of research consists in conducting students interviews in which they are encouraged to express their solutions in natural language. The specific goal of the paper is to show how Piaget's ideas are used as guidelines in the design of the interviews and in the analysis of the gathered information. Selected excerpts from students' interviews are included.

1 Introduction

The study presented in this paper is based on Jean Piaget's theory *genetic epistemology*, especially on Piaget's and collaborators' works regarding the psychological evolution of mathematical concepts and theories[12, 13]. Their investigations are supported by an extensive amount of empirical work. The importance of Piaget's work for computer science education lies in the fact that *the methods* that the subject employs to solve instances of problems play a central role in the explanations about the construction of knowledge. In this sense, the theory gives satisfactory support to the study of the psychogenetic evolution of algorithms, from which invaluable lessons can be taken when it comes to preparing instructional proposals.

The main theoretical principle in which our research is based is: the source of the instrumental knowledge is the interaction between the subject and the structure over which the algorithm is applied. This interaction is governed by the *general law of cognition*, that explains how instrumental knowledge can be transformed into conceptual knowledge. In Piaget's theory, the principal instrument of the whole process is called *reflective abstraction*[8, 11, 13].

In this paper we present an example of application of the above principle to the construction of knowledge about the binary search algorithm. The goal of the study is to gather information about the process of transforming the instrumental knowledge constructed by the students while solving an instance of the problem of searching an element in an ordered list into conceptual knowledge about the employed algorithm. The aim of the paper is to show how Piaget's explanations about the instruments and mechanisms involved in the construction of knowledge are used as guidelines of the study.

In this sense our work intends to contribute to computer science education research, accordingly to [2]. There, the authors give the following definition of the expression "computer science education":

The academic discipline computer science education consists in focusing research on the application of principles from educational-related disciplines -pedagogy, epistemology, curriculum studies and psychology- to the teaching and learning of the scientific discipline computer science as a school subject.

They add that the strong connection with educational-related disciplines constitutes the theoretical argumentation of the research as a means of providing evidence of its effectiveness.

The methodology of research consists in conducting students interviews, in the manner of Piaget and collaborators[5]. In the interviews the students are asked to search a word in a dictionary and are encouraged to explain in natural language how they did it and why they succeeded, as a first stage in the conceptualization. All the interviews were audio-taped and transcribed and in this case they have also been translated to English. The methodology of research has been inspired by APOS theory, a theory of learning mathematics elaborated by mathematics education researchers, based on Piaget's genetic epistemology[7, 1].

The paper is organized as follows: in the second section, the main ideas from Piaget theory are briefly described. The section also includes a summary of the identification of stages in the construction of the concept of the algorithm of binary search as a school subject. In the third section, the study is described including the design of the questions to the interviews, in the fourth section, the guidelines of the analysis of the responses of the students, selected excerpts from the interviews and the introduction of the formalization of the algorithm are included, and in the fifth section some conclusions and further work are presented. The sixth section includes acknowledgements.

2 Related works

The main ideas presented by Piaget and collaborators in works published in the sixties/seventies by the Series of "Etudes d'Epistemologie Genetique" (Presses Universitaires de France)[8–10] and in [11–13] constitute one of the sources of our investigation. Many of these works contain complete descriptions and analysis of several experiments, including selected parts of conducted interviews. In the experiments several types of problems are posed to determine the psycho-genetic evolution of concepts, such as modus ponens, modus tollens and the transitivity of the implication.

The construction of these mathematical concepts follows the general law stated in "The Grasp of Consciousness"[8], according to which the process of construction of consciousness that accompanies the interaction between the subject and the object, always goes from the periphery (the result of the actions) towards the center (the intern mechanism of the actions). This general law explains why, at all levels, the conceptualization of the operations of the subject always occurs later than nonreflective use of the same operations.

On the other hand, in [13], the authors describe the instruments common to all acquisitions of knowledge and the processes that result from applying these instruments. The general source of the instruments are the processes of *assimilation* of representations of objects or events to the mental schemata of the individual, and the *accommodation* of the mental schemata in accordance with the representation to be assimilated. The instruments of knowledge generated by assimilation and accommodation are *abstractions and generalizations* in their diverse forms. The authors distinguish between the empirical abstraction, that draws information from the objects themselves, and the *reflective abstraction* that draws information from the actions and operations of the subject. As a result of abstraction, a reorganization into a new conceptual structure takes place, which is assimilated into the earlier one, allowing its generalization. The authors recall that *constructive generalization* is not only assimilation of new contents to already constituted forms, but generation of new forms and new contents [10].

And this is where the factor that plays a fundamental role as motor of the process comes into the scene: *the search for reasons and the inherent necessity.* The authors point out that a construction of the subject is not satisfactory, in the final analysis, unless it acquires intrinsic necessity, through explicit reasons [13].

The works of Piaget and collaborators have important pedagogical implications for the learning of mathematics and computer science concepts. The example presented in this paper illustrates the identification of three main stages in the construction of the concept of the algorithm of binary search as a school subject:

- An *instrumental* stage in which the students construct knowledge interacting with an instance of the problem without consciousness about how the solution is achieved and why it works. This knowledge constitutes the source of conceptual knowledge. For instance, the individuals solve the problem of searching a word in a dictionary (an instance of searching an element in an ordered list) but have difficulties in describing the steps and are unaware of the reasons of the success.
- A stage of *conceptualization* in which the evolution from the instrumental knowledge to conceptual knowledge starts by the grasp of consciousness and the reflection about the actions involved in the developed method and about the reasons of the success (or failure). This reflection leads to an understanding of the relationship between the structure of the elements over which the method is applied (the smaller parts of the dictionary) and the components of the method (choose, compare, search). This process ends in the comprehension of the algorithm and prepares the mind for the next stage.
- A stage of *formalization* which consists of constructing a correspondence between students' concept and a universal system of symbols. This is the stage that transforms a concept into a school subject. Students become aware of ambiguities and/or errors in their specifications and correct them gradually approaching a formal definition of the solution of the problem.

These stages act in a pro and retro-active manner, influencing the development of each other. For instance, the interaction between defining the method (formalization stage) and applying it to particular cases (instrumental-conceptualization stages) allows both to refine the definition and to improve the understanding of the algorithm.

3 The study

The study consists in conducting interviews to ten entering students of an introductory course of programming. The students are required to look up a word in a dictionary and to explain in natural language how they did it and why they succeeded. The problem is an instance of the general problem of searching an element in an ordered list and the solution is an instance of the algorithm of binary search. It is well known and automatically solved by the students. In the interviews they are encouraged to think about both the coordination of their actions (the method) and the modifications that these impose to the object (the structure). Accordingly to theoretical tenets, in this interaction the successfully done actions are transformed into operations leading to the conceptualization of the algorithm[8].

This task is adequate for the investigation because of the following characteristics:

- the algorithm of binary search is commonly applied in solving this task, that is to say, all students know very well the application of this algorithm to this particular case, and all of them success in searching a word,
- the task is (almost) automatically done, what gives us the opportunity of analyzing its grasp of consciousness and conceptualization in detail from the origin of the process,
- no numeric domain is involved and the role of school is minimal which diminishes the influence of preconceived ideas,
- this algorithm is one of the most important methods of searching and it is taught in all courses of programming, traditionally introducing some formalism to implement it, in the form of explanations given by the teacher to the students.

It is expected that the students solve the problem using the algorithm of binary search, without being aware of what that means. It is expected that the interviews help the students in understanding the algorithm with respect to:

- the structure of the dictionary as an ordered list of words,

- the actions that compose the algorithm: choosing a word, comparing words and a new instance of the search itself,
- the reasons of success: each new instance of the search is applied on "smaller" parts of the dictionary (do the same), all the smaller parts hold the property of containing the searched word (invariant), the search ends when a special case is achieved (termination).

Accordingly, the series of questions of the interviews is divided into the following segments (Q1 means question 1, etc):

- Q1 to Q4: Structure
- Q5 to Q6: Method (at the beginning of the interview)
- Q7 to Q8: Reasons
- Q9: Method (at the end of the interview)

3.1 Designing the interviews

The enumerated questions listed below were used as a basis for the interviews. In some cases, other questions were added at the moment or the same question was formulated in another way, depending on the development of each interview. The whole content of the interviews can be found in [3]. The Spanish word for "cat", that is, *gato* is maintained as a remainder of the original language of the interviews.

Questions 1 to 4: The first four questions are aimed, on one hand, to establish a fluid contact between the student and the interviewer and on the other hand, to induce the student to reflect about the structure of the dictionary as a list of words alphabetically ordered. It is the existence of this order relationship that determines the method of searching and consequently, we think that it is a relevant concept that has to be assimilated by the minds of the students as a requisite for understanding the method and the reasons of success.

Q1: What is a dictionary?
Q2: Knowing that a word, for example *gato*, is in the dictionary and also in this novel, where do you think that will it be simpler (easily and quickly) to find it?
Q3: Why?
Q4: What makes the difference then between a dictionary and any other book?

Questions 5 and 6: The goal of the following questions is to apply Piaget's ideas about the conceptualization of automatically done actions. Interrupting the action and introducing the need for thinking forces the student to direct his/her thought from the result of the action towards the intern mechanism of the coordination of the actions that has given rise to that result. This movement "from the periphery to the center", that Piaget calls *The general law of cognition* [8], allows the subject to construct better representations, on the one hand, of the objects and on the other hand, of his/her own actions which are transformed into operations (methods). In this case, the conceptualization of the method demands to interrupt the automatic application and to mentally identify the sequence of other involved methods: to choose a word, to compare it with the searched word, to make a decision according to the result of the comparison, to do the same (another instances of application of the same method).

Q5: Look up the word *gato* in the dictionary.
Q6: Describe, step by step, how you achieved it. (Eventually, ask them to do it again).

Questions 7 and 8: The goal of the following questions is to apply Piaget's ideas about the role of the process of "searching the reasons of success" in the conceptualization. The constant motor impulsing the subject to complete or to replace the observables of facts by deductive or operative inferences is the search of *reasons* for the obtained result [10, 11]. That means that "the search of reasons" impulses students' thought towards the interaction between the coordination of his/her actions and the modifications imposed to the object, reaching an equilibrium generating a mental representation of the algorithm. That means, for the case of this algorithm, make the students aware of the relationship between the arguments to which different instances of the method are applied: each new instance of the search is done on "smaller" parts of the dictionary (all holding the searched word), until a special case is achieved (termination).

Q7: Knowing that a word is in the dictionary, is it always possible to find it?
Q8: Why?

Question 9: This question is essentially the same as Q6 but is posed at the end of the interview. By comparing the responses to both questions (Q6 and Q9), the impact of the interview on the levels of conceptualization of the algorithm of binary search can be appreciated. The goal of the question is to determine whether the interview has helped the students to improve their levels of conceptualizations with respect to, on one hand, the decomposition of the algorithm in its components and on the other hand, the description of a general algorithm. The epistemological motivation arises from the fact that the ability of detaching the thought from particular cases and comprehending the intern mechanism of the actions is considered an advance in the conceptualization by Piaget.

Q9: If you had to explain to a little child -who knows how to read and knows the alphabet- how to look up any word in a dictionary, what would you say?

4 Analysis of the information gathered in the interviews

As expected the students solve the problem using the algorithm of binary search, without being aware of what that means. Through the interviews, the students transform this "know how to" into knowledge about "what is done" and "why it works", by the process of reflective abstraction [12].

The analysis of the information gathered in the interviews is organized in the following segments:

– Responses to questions on structure and method (Q1 to Q6).
– Responses to questions on reasons of success (Q7 and Q8), including excerpts from the interviews.
– Responses to the question on the method posed at the end of the interview (Q9), including responses to Q6, to facilitate the observation.

The analysis of every segment is based on descriptions and explanations contained in the respective references. Selected excerpts from the interviews are included, in which relevant comments relating theoretical concepts to the responses of the students are indicated in italics. A short description of introducing the formalization of the algorithm using as start point the conceptual knowledge constructed by the students is included following the analysis.

4.1 Responses to questions on structure and method (Q1 to Q6)

The responses to questions Q1 to Q4, constitute examples of how the thought advances from the periphery (inferences focused on the characteristic of the dictionary related to its use) towards the center of the object (inferences focused on the structure of the dictionary), induced by the corresponding questions. This movement is governed by tenets called by Piaget *the general law of cognition* and is the start of the process of conceptualization [8].

Only one student characterizes the dictionary by its alphabetic order relation. The other students refer to what one can do with the dictionary or what it is for, in other words, their responses are adequate to questions like "what can you do with the dictionary?" or "what is the dictionary for?". Because of the obvious character of the property of being an ordered list, it is more difficult to conceptualize. Faced to questions Q3 and Q4 (why is it easier to find a word in a dictionary than in a novel, inducing to think about the difference between both structures), all students recognize the alphabetical order, which reveals that they become aware of the relevant property of the object.

The answers to question Q6 reveal that the students have a very weak conceptualization about their method of searching a word. The students apply binary search as expected, but they are not aware of the different actions they do to achieve the result. The questions following Q6 (Q7 and Q8) play an essential role in the conceptualization of the method, inducing the students to reflect about the reasons of success. The goal is that they achieve on the one hand, the decomposition of the method in its components: *choose a word, compare words* and *do the same*, and on the other hand, the transformation of the dictionary in smaller parts, which are the central questions of the algorithm. To facilitate the observation of advances in the conceptualization, some responses to question Q6 are included together with responses to question Q9.

4.2 Responses to questions on reasons of success (Q7 and Q8)

The analysis of the responses of the students to questions Q7 and Q8 revealed that it is hard for the students to understand the reasons by which the binary search works. New questions (no numbered in the excerpts below) were added in order to help the students to surmount this difficulty: they were asked to use another method (called "the second method" in the following) which consists in asking the students to open the dictionary and to look if the searched word is there. If it is not, (which will happen in essentially), they are asked to close the dictionary and to begin to search for the word again, that is, in the whole dictionary. This strategy was effective: after using the second method, all the students immediately became aware of the changes that their actions impose to the object: the sequence of parts of the dictionary, *each time smaller* all of them holding the searched word, and the termination case.

Below, selected excerpts from the interviews illustrate about:

- the difficulties of the students in finding the reasons of success before using the second method (student 1),
- the effectiveness of the strategy of using the second method (students 2, 3 and 4),
- advances in the conceptualization of the algorithm with respect to: the decomposition of the method in its components (choose, compare, search) (student 2), the invariant property (students 3 and 4), the termination case (students 1 and 3).

The responses of the students to questions Q7 and Q8 are classified on the following types, accordingly to the factor that they point out in searching the reasons of success:

- type 1: both the method and the dictionary,
- type 2: the method,
- type 3: the dictionary.

The goal of the questions is to induce the movement of the students' thought between types 2 and 3 above, accordingly to the general law of cognition [8].

Selected excerpts from the interviews

Student 1
Q7: Knowing that a word is in the dictionary, is it always possible to find it?
R: Yes.
Q8: Why?
R: Because it will always be in that order. Because looking for in this way and due to the order the dictionary has, I will find it.
(He refers to the method and the dictionary: type 1)
Q: Good, we look for the word *gato*. Here, we are searching here, in this piece as you said and, what is it like regarding the whole dictionary?
R: The rest.
Q: And what is this rest according to the whole dictionary?
R: More useful for me.
Q: Yes, and with respect to the number of words that has?
R: Greater.
Q: ... !! This part is with respect to the whole dictionary?
R: Smaller than the complete dictionary.
Q: And now, how do you go on searching?
R: (He does it)
Q: And now what happens with all this part?
R: I do not need it.
Q: And where will you search?
R: Only on this page.
Q: And what is this page like according to the previous section you had?
R: Much smaller.
Q: Then what happens with the dictionary when you are still searching?
R: Some parts are being discarded.
Q: And it becomes ...?
R: Smaller.
Q: When you find the word, what happened to the dictionary?
R: It is useless, it only helps me for that word only.
Q: So, what was the dictionary transformed into?
R: Into only one word. *(Termination case.)*
Q: That is why you find it, because you search in sections that become smaller and smaller within the dictionary.
R: Oh, of course.

The first type of response (illustrated in this excerpt) reveals at first sight a better conceptualization of the reasons of success because both the method and the object are mentioned. However, observe the difficulty of the student, in constructing a mental representation of the relationship between the parts of the dictionary. This can be explained by the fact that this student was interviewed before the strategy of using the second method was implemented.

Student 2
Q7: Knowing that a word is in the dictionary, is it always possible to find it?
R: Yes.
Q8: Why?
R: Because I always get the same method to look it up, I always apply the same method by which ... everybody uses it for the dictionary.

(*He refers just to the method: type 2*)
Q: And that method you apply and that everybody uses, why does it guarantee you that you will find it?
R: If it is in the dictionary?
Q: Yes.
R: ...
Q: What does this method have that guarantees you find it?
R: It never fails.
Q: Why?
R: ... because ... because it holds all letters in the dictionary, I mean I always ... never will get lost ... as long as I search one ... then I will always follow the same order until I find the word, by each letter I go on searching.
Q: I propose another method. (He does it using the second method).
R: Yes, of course, we never finish.
Q: Which is then the difference between this method and the other?
R: I based myself on the principles (*He is thinking of his own action now, trying to decompose it*), that is, I open the dictionary and I based myself on what I find at first sight, (*choose*) then I start checking if I have to go forwards or backwards (*compare*).
Q: Suppose you go forwards, what happens to the rest?
R: I discard it.
(*Observe that he is still thinking of his own action. However, to advance in conceptualizing his thought has to move to the transformation imposed to the object because of his action. That is what next question induces.*)
Q: Then, what happens with the dictionary while you are looking up?
R: I start discarding until I find the word I am looking for.
Q: Then, what happens to the dictionary while you are searching?
R: They are being eliminated, it becomes reduced (*"smaller" arguments*).

Student 3
Q7: Knowing that a word is in the dictionary, is it always possible to find it?
R: Yes.
Q8: Why?
R: Because I trust myself, because yes, (she laughs).
Q: You'll find it rather quickly or you will take the whole day.
R: Quick.
Q: Why?
R: Because I have practice. *(Observe that she is thinking of her own actions, response of type 2.)*
Q: Now we are going to use another method (the second method). Open the dictionary (she opens it).
Q:Is *gato* there?
R: No. (We do it again many times).
Q: What do you think of this method with respect to the other?
R: It's much more difficult.
Q: And what is the difference between them?
R: That in the first one I start marking between which and which and I can shorten the limits (*"smaller" arguments*) and in the other it is at random.
Q: Why are you sure that with your method you will find it and quickly?
R: I have fewer bounds, fewer limits. (*She begins to think of the dictionary*).
Q: What happens with the group of words in which you search?
R: They start shortening.
Q: Until?

R: There is one word, the searched one (*Termination case*).
Q: If it is not?
R: It is not possible, I choose the parts in which the word is ... (*The selected part contains the searched word*).

Student 4
Q7: Knowing that a word is in the dictionary, is it always possible to find it?
R: Knowing that it is in the dictionary, yes.
Q8: Why?
R: Due to the order it has.
(*Response of type 3 above, he attributes the success to the dictionary.*)
Q9: Good, let's use now another method to find the word *gato* with the same dictionary having the same order. We close the dictionary and I ask you to look up the word *gato*.
R: (He does it).
Q: Is the word *gato* there? (Where he opened).
R: No.
Q: Close it and look up *gato* again.
R: (He does it).
Q: Is it there?
R: No. (It is repeated some times).
Q: With this method the dictionary keeps its order and however, do you think that in this way we'll find the word *gato* easily?
R: No.
Q: Instead, with your method?
R: I find it fast.
Q: What is the difference between the two methods?
R: One is safe and the other isn't.
Q: Why?
R: Because there is an order.
Q: No, in the second method the order remains in the dictionary.
R: But I didn't follow it.
(*Observe that now his thought has moved to his actions.*)
Q: What does "following the order" mean? Let's use your method again.
R: (He does it.)
Q: Why are you browsing in that direction?
R: Because there is letter "e" and I know that letter "g" is after.
Q: What happens then with all this part of the dictionary?
R: I discard it. I know that here "g" is not and I discard it.
Q: Let's go on.
R: (Goes on searching and he passes by).
Q: What do you do with all this part of the dictionary?
R: Also, I know that it is not in this part.
Q: What happens with the dictionary?
R: There are parts which I know that it is not there and I discard them. (*The selected part contains the searched word.*)
Q: So what happens to the dictionary?
R: It becomes smaller (*"smaller" arguments*).

4.3 Responses to question Q6 and Q9

The responses to questions Q6 and Q9 allow to some extent to measure the impact of the interview in students' conceptualization of the algorithm, because they are essentially the same question, posed at the beginning and at the end of the interview respectively. Evidence of the advances are the references to a general algorithm and better descriptions with respect to the decomposition of the method in its components actions: *choose a word, compare words* and *do the same.* Selected responses to the questions Q6 and Q9 follow.

Student 1:

Q6: I check the beginning letter, with "g", there is an alphabetical order then, more or less I know that it is in the middle upwards, then I check which letter I am, for example: If I open to the "e", I know that "g" is after, that is the order, I go forwards, I go on checking and if I advance too much, I come backwards and so on, *until I find "g" and after that I search.*

Q9: First that he opens the dictionary, that he reads one word (*choose*) and he defines the relation, that is what he tells me if the letter with which he started is before or after than the one he wants to find and he tells me that if it is after (he passes by) I ask him to choose another that is before (*compare*) and in this way successively (*do the same*).

Student 2:

Q6: First I opened the dictionary in a section that seemed near to "g" and went in order always visualizing the alphabet in my mind. I went up to "g" and then with the 2nd letter, which is the "a", and then the "t" *until finding the word.*

Q9: That the word that he wants to find will be in order according to the alphabet that he knows. Then, what he has to search first is the beginning letter of the word. After having found the first letter, he can start with the 2nd. Proceeding also in the same way.
Q: The kid is learning to search ... Then, when he opens the dictionary, what does he have to do to find the 1st letter?
R: A relation between what he is seeing (*choose*) and what he wants to find and according to this relation (*compare*) he "operates" forwards or backwards.
Q: And when he finds the first letter?
R: He operates again using the same order (*do the same*).

Student 3:

Q6: I was checking on the letter according to the order that I have in my mind, *first, the first letter, secondly the second and so on.*
Q: Doing what?
R: Discarding the ones didn't help, and looking up the words with the letters I was searching for.

Q9: Take the dictionary and choose one word (*choose*) and then with respect to that one, start looking if it is greater or less (*compare*) until finding the 1st letter and after you found the 1st letter, you have to look for the 2nd , in order as well, and go on until you find the whole word (*do the same, termination*).

In the answers to question Q6 the actions involved in the method, are implicitly mentioned for the case of the letters of the word, while in the answers to question Q9 most of students explicitly talk about "a relation between words" (*choose, compare*), "proceeding in the same way" (*do the same*), and in some cases also about the termination condition. On the other hand, responding to question Q9 most of students describe a general algorithm of binary search. The next subsection shows the introduction of a formalization of the algorithm derived from a synthesis of students' responses to the question Q9.

4.4 The stage of the formalization

A week after the end of the individual interviews, a collective class was taught to all students to represent students' descriptions in a formalism similar to mathematics. The meaning of formalizing is to put into correspondence mental constructions (concepts) with some universal system of symbols [9]. Traditionally, in computer science teaching algorithms are described using different formalisms (pseudo-codes, diagrams, flow-charts). But these formalisms are not universal systems of symbols, while mathematics is. The following notation is used to describe students descriptions in a language similar to mathematics: the dictionary is represented by a list of pairs (word, meaning). A definition of the method of searching a word in a dictionary is written as:

```
search(word,[(Wfirst,meaning)...(Wlast,meaning)]) =
      let word' = choose-word([(Wfirst,meaning)...(Wlast,meaning)])
      if word = word' then select-the-meaning
      else if word'<word then search(word,[(word',meaning)...(Wlast,meaning)])
             else search(word,[(Wfirst,meaning) ... (word',meaning)])
```

Each student was provided with a sheet containing his/her responses and once the students have worked this definition out, other questions are discussed, for instance, "How can we define a more efficient method 'choose-word' in the searching algorithm?" The term efficient is not specified to see what the students think about it. All the students have answered that they should take the word from the middle, and that this is more efficient because there is more possibilities of discarding a greater number of words.

At the end of the class, some exercises were handed out to the students to be solved individually or in groups. The exercises asked about problems presenting both similarities and variations with respect to the solved problem, for instance,"What happens if the word is not in the dictionary?" and "How would you do to insert it in the dictionary?" Such questions give the opportunity of comparing different solutions/problems and reasoning about efficiency issues in a context that the students can easily understand.

The analysis of students' responses to those questions allows to investigate how the constructed knowledge is used in solving new problems[10] (not included in this paper).

5 Conclusions and Further Work

The study described in this paper is an example of the contribution of Piaget's theory-*Genetic Epistemology*-to computer science education research. The paper shows how principles from the theory can be applied to learn about students' understanding, using as an example the binary search algorithm. The design of the interviews (section 3) and the analysis of students' responses (section 4) follow the main ideas briefly described in section 2.

The selected excerpts from the interviews reveal the impact on the construction of the concept of binary search algorithm of

– *the process of interaction* of students' thought between the applied method *and* the modifications on the object, and
– students' reflection about *the reasons of success* of their solutions to the problem.

Advances on the conceptualization are illustrated by students' descriptions at the end of the interviews. A synthesis of these descriptions is used to introduce a formalization of the algorithm in a language similar to mathematics, as described at the end of section 4.

The future work focuses on describing parts of the research not included in this paper, for instance, the implementation of the algorithm in the functional programming language

Haskell[1] and the investigation of applying the constructed knowledge to solve other problems, as mentioned at the end of section 4.

To discuss ideas related with the study described in this paper, a series of workshops were conducted during the second semester of 2007 with the participation of computer science educators at Instituto Universitario Autónomo del Sur[2]. In the last years, some innovative methodologies taking into account the activities developed in the workshops have been put in practice[6]. The analysis of these experiences is also one of the points of the further work.

The elaboration of instructional proposals about the learning of algorithms, aimed at providing computer science educators with teaching strategies, is a matter of the further work as well. Many algorithms used in real life to solve problems or perform tasks are instances of general algorithms, taught in programming courses, for instance, searching (binary and sequential), inserting/deleting elements in lists, ordering lists. On the other hand, the students learn how to use algorithms in mathematics courses that can serve the same purpose[4].

6 Acknowledgements

I thank Federico Gómez Frois for correcting the English. The comments of the anonymous referees are gratefully acknowledged.

References

1. Mark Asiala, Anne Brown, David DeVries, Ed Dubinsky, David Mathews, and Karen Thomas. A Framework for Research and Curriculum Development in Undergraduate Mathematics Education. *Research in Collegiate Mathematics Education II, CBMS Issues in Mathematics Education, 6, 1-32*, 1996.
2. Linda McIver Christian Holmboe and Carlisle E.George. Research Agenda for Computer Science Education. *In G.Kadoda (Ed). Proc. PPIG 13, pp 207-223*, 2001.
3. Sylvia da Rosa. The learning of recursive algorithms and their functional formalization, 2005. Website. `http://www.fing.edu.uy/~darosa`.
4. Sylvia da Rosa. Designing Algorithms in High School Mathematics. *Lecture Notes in Computer Science, vol. 3294, Springer-Verlag*, 2004.
5. Sylvia da Rosa. The Learning of Recursive Algorithms from a Psychogenetic Perspective. *Proceedings of the 19th Annual Psychology of Programming Interest Group Workshop, Joensuu, Finland*, 2007.
6. Sylvia da Rosa and Federico Gómez Frois. Una metodologia educativa basada en el trabajo del estudiante. *Proceedings of CIESC 2009, (Congreso Iberoamericano de Educacion Superior en Computacion, Pelotas, Brasil), to appear in a special edition of the CLEI Electronic Journal (see http://www.clei.cl/cleiej/)*, 2009.
7. Ed Dubinsky and Michael McDonald. *APOS: A Constructivist Theory of Learning in Undergraduate Mathematics Education Research.* Springer Netherlands, 2006.
8. Jean Piaget. *La Prise de Conscience.* Presses Universitaires de France, 1964.
9. Jean Piaget. *L'équilibration des Structures Cognitives, Problème Central du Développement.* Presses Universitaires de France, 1975.
10. Jean Piaget. *Recherches sur la Généralisation.* Presses Universitaires de France, 1978.
11. Jean Piaget. *Success and Understanding.* Harvard University Press, 1978.
12. Jean Piaget and Evert Beth. *Mathematical Epistemology and Psychology.* D.Reidel Publishing Company, Dordrecht-Holland, 1966.
13. Jean Piaget and Rolando Garcia. *Psychogenesis and the History of Sciences.* Columbia University Press, New York, 1980.

[1] www.haskell.org
[2] A private educational institution of computer science in Montevideo, Uruguay

Teaching Novice Programmers Programming Wisdom

Dr. Randy M. Kaplan

Kutztown University of Pennsylvania
Kutztown, PA 19530
kaplan@kutztown.edu

Keywords: POP-I.A. learning to program; POP-II.A. novices

Abstract

Teaching students how to write computer programs has remained a challenge. Whether a student is new to programming or has some experience they often have not had enough to develop strategies for solving problems in a computer programming setting. It is akin to being in a rowboat without any oars. Yes, you can get there; it will just take a long time.

There has been a great deal of research over the years into the psychological and cognitive aspects of programming. The reality is that much of this research has not informed our teaching and we are still teaching programming as we did many years ago. Although the academic community may take offense at this statement looking at the way programming is taught today and applying the relevant research to the process can justify it. Without going through that process, it remains easy to see. Programming is largely taught today the way it was taught 60 years ago.

Although this paper is entitled "Teaching Novice Programmers Programming Wisdom," the wisdom referred to are encapsulations of some of the meta-skills that are needed to successfully write computer programs. The list is not meant to be in anyway comprehensive, complete, or all encompassing. It is a list used with some success in entry level programming courses. The purpose of writing and presenting the idea of wisdom for programming is to allow the computer science/education/psychology communities to comment on this wisdom and possibly offer other "nuggets" and approaches that might be employed during novice programming instruction.

1. Introduction: The Need for Wisdom

There are two main issues that are key to successfully teaching students to write computer programs. The first is how do you convey the information needed by the student actually write programs. This is akin to learning how to speak a new language; learning the words of the language and how to construct sentences in the language. The other issue concerns the strategies that students can use while they are learning and using the new language. These so-called meta-strategies go a long way to make learning and using a new language an easier process.

There have been many studies of programming (Blackwell 2002), how people learn to program (Buckingham, Hynd et al. 2004), how novices learn to program (Mayer 1981), and the psychological and cognitive aspects of programming (Sheil 1987). There have also been studies of programming languages that attempt to identify characteristics that may make them easier to learn (McIver and Conway 1996).

Those of us that teach programming have a tremendous challenge. We are required to teach a skill that requires us to impose rules for problem solving in a foreign language (a programming language). We teach programming much in the way we taught it from when programming was first taught. We tell the student about the tools and then we explain how to use the tools to solve specific problems. We do not address the necessary overarching or meta-skills that are required in the programming process. David Gries (1974) states that we do not explicitly teach problem solving when we teach people how to program. Gries observations and the context in which he gives them are important. The comments were given some 36 years ago without supporting evidence or research.He goes on to say "the bright students somehow catch on …" Polya's work (1988) about problem solving stands by itself as a concise model of the kinds of heuristics that are extremely useful to students learning to solve mathematical problems. There is nothing special about the heuristics that Polya provides. The uniqueness of his book is that he sets down on paper certain principles for the teacher and for the student that are extremely useful for both. When reading Polya's book one is continually reminded of their familiarity, yet somehow having them documented in this way makes them more tangible and useful.

In this paper, following in the footsteps of Gries and Polya, statements will be made based on our experience with programming and teaching novice students to program.

When teaching a CS1 course it becomes apparent that most students do not have or are not aware of the meta-cognitive skills that are needed for programming. It became clear that an additional "pre-CS1" course would help make the CS1 course more accessible to students taking it. The purpose for CS0 was to level the playing field for all students. This course afforded an opportunity to cover some of the aspects of programming that are missed in the typical CS1 course. Specifically we would have an opportunity to incorporate teaching of the meta-skills necessary to carry out the programming task. The dilemma was how to present these meta-skills in such a way that they can be remembered and used by the students for the programming process. This paper describes the formulation used for presenting these meta-skills.

2. Introduction: Codifying Wisdom

It would be convenient if we could simply impart wisdom by codifying it in some way that (a) everyone remembers, and (b) everyone uses. In this sound byte, 10 second news story, limited attention span world, it would in fact be excellent because then, when we teach programming, we could impart this wisdom so our students would not find programming as frustrating as they do. Unfortunately codifying the experience of programming is not an easy thing to do.

There are many attempts at codifying programming wisdom. Through the years there are some notable artifacts of programmer wisdom. Examples include Frederick Brooks classic, "The Mythical Man-Month (Brooks 1995)," "201 Principles of Software Development (Davis 1995)," and "Joel on Software (Spolsky 2004)" are all examples of attempts to encapsulate some lessons learned (wisdom) for the purpose of improving the software development process.

Each of these venerable works contains important wisdom for programmers and system developers. On average each has 300 pages and would require a course to cover their contents. For the novice, much of their contents would probably be irrelevant to their task. Another work, "Code Complete" (McConnell 1993) is an excellent text about how to make sure that the code that a programmer writes works. This work (and subsequent editions) are also must reads for the programmer although not very appropriate for the novice programmer.

Statements of wisdom, also known as heuristics, can be gleaned from experience and if properly expressed can be immensely useful to the student. It is often said that the problem with novice students is not that they need meta-strategies; it is more that they need basic domain knowledge. We would claim that this is not an either/or situation but actually an "and" situation in that novices need both kinds of knowledge – especially when it comes to computer programming.

The heuristics or wisdom described in this paper bears some relationship to the work on meta-cognition in programming. Meta-cognitive skills are those that are used as strategies for solving problems or carrying out tasks. These may seem to be intuitively useful in the case where the situation is unfamiliar to the student or practitioner but this is not a confirmed fact. Research carried out by Shaft (1995) shows that when experienced programmers used meta-cognitive skills when attempting to carry out programming understanding, the use of these strategies did not improve the understanding of the programs.

There has also been research investigating what meta-skills may be used when programming. Uncovering what these skills might be has been somewhat allusive. One step in this process would be to classify the level or kind of skill needed to perform a specific programming task. Shuhidian (Shuhidan, Hamilton et al. 2009) attempted to do this but found that classification was more difficult than expected.

3. This Paper's Contribution

How can we codify the meta-skills of programming in a form that can be easily remembered and used by novice students? This was the challenge for a CS0 course. This paper defines a codification of programmer's wisdom suitable for the student beginning their computer science or information technology programs. In fact, this codification is one that can be continuously used by the student. Students who have graduated from our programs have commented that they continue to use these meta-skills. The contribution of this paper is the definition of the codification of a programmer's heuristics.

4. Programmer's Wisdom

4.1 Precedents for Experiential Wisdom (Heuristics)

One of the most notable examples of a presentation of experiential wisdom is the work of Polya mentioned earlier. Early in his work Polya specifies four phases for finding a solution to a problem. Comparable to some aspects of the programming wisdom to follow, Polya identifies the phases as understanding the problem, planning for solving the problem, execution of the plan for solving the problem, and lastly a "looking back" at the process used to solve the problem (Polya 1988), What justifies these phases? Who are they attributed to? Surely they existed before Polya. Polya next explains each of the phases in greater detail. For example, one of the statements made, "The students should consider the principle parts of the problem attentively, repeatedly, and from various sides(Polya 1988)." Again the question arises, where did this come from?

When reading Polya it is clear that the source of these heuristics was his own experience over. The present work has similarities to Polya's. It is an attempt to codify heuristics and wisdom that can be used by novice programmers to solve programming problems much as students solving mathematics problems can use Polya's work.

4.2 Wisdom Explained

In this section the "nuggets" of wisdom (heuristics) will be introduced and explained. Before introducing them, some comments are in order about the kinds of skills that these statements address. We can identify two kinds of statements. The first type of statement is one that is relevant to the programming process. The second type of statement is of a more general type that would apply to both writing computer programs and problem solving. Rather than consider this distinction a strict dichotomy between domain-relevant statements and more general problem-solving statements (or meta-cognitive statements) we can consider a continuum between these two classifications. A particular statement is to a lesser or greater extent of one or the other classification. We assume that statements having more to do with the programming domain will be on the one side of the continuum while those having more to do with meta-cognitive skills on the other side of the scale. Figure 1 shows a classification of the statements of programming wisdom on the continuum.

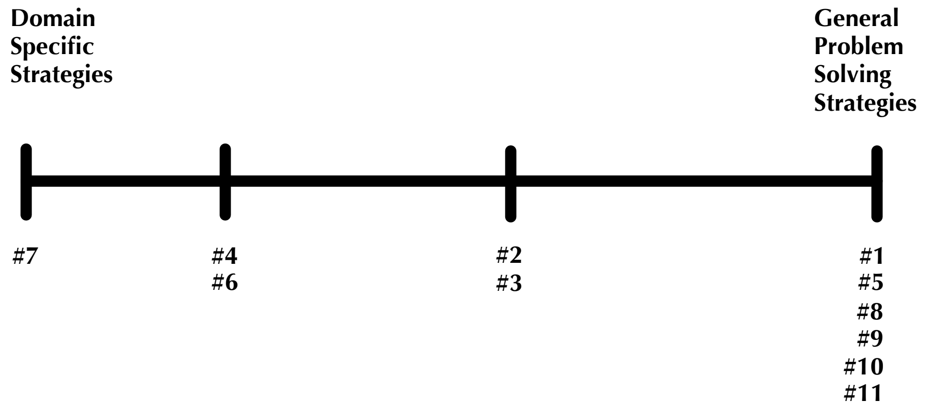

Domain Specific Strategies
General Problem Solving Strategies
#7
#4
#6
#2
#3
#1
#5
#8
#9
#10
#11

Figure 1 – Continuum of Programming Strategies vs. General Problem Solving Strategies

The codification of the programmer's wisdom is divided into 11 statements. There is nothing significant about this number except that it had to be a small enough number of statements to remember. Students readily memorize the 11 statements without difficulty. The 11 statements are as follows.

1. BREAK IT DOWN
2. LEARN THE BASICS WELL, LOOKUP THE REST
3. FIND AND USE EXAMPLES
4. TEST TEST TEST (and test some more)
5. FOLLOW DIRECTIONS
6. SOLVE THE PROBLEM IN ENGLISH FIRST
7. REPRESENT WHAT?
8. WHAT DOES IT DO?
9. MAKE SURE YOU UNDERSTAND THE PROBLEM
10. IF YOU CAN'T SOLVE THE BIG PROBLEM SOLVE A SMALLER ONE.
11. HAVE A PLAN B

5. Wisdom Defined

The 11 statements listed above are explained below. When the students are introduced to the 11 statements, they are given an explanation of each statement. The descriptions below are in lieu of the actual classroom explanations.

Break It Down

When you are faced with a problem that is too complex, one way to approach it is to break the problem down into smaller parts. By breaking the problem into smaller parts you may be able to more readily solve the original problem by solving its pieces.

Learn the Basics Well, Lookup the Rest

Beginning programmers often get hung up on how to do certain things. Because programs have to be very precise it is easy to make mistakes when you first start out. The point is that it is okay to look up what is needed when a student becomes stuck.

Find and use examples

There are thousands upon thousands of coding samples that can be examined to better understand how a specific task is accomplished. Using examples to create programs does two things; first you get to see how something is done, and if you pay attention by seeing how it is done you will learn how to do it. Second, you'll spend less time, at least at the start, writing your programs. Students complain about how long it takes to write a program. If you can find and use examples and you can learn from the examples you might be able to learn faster then if you tried to write a program from scratch.

TEST TEST TEST (and test some more)

Testing is part of programming. You test to get a program to work and you test to make sure the program works correctly. The better you test, the better your program will be.

Follow Directions

Make sure that you are doing what the directions say. The directions tell you what you can and cannot do. Don't overcomplicate or oversimplify what you have to do. Sometimes the directions will even give you part of the solution – so read carefully.

Solve the problem in English (or your native language) first

Learning a new programming language is similar to learning a new foreign language. You just don't jump right into writing the steps in the new language. That is a sure fire recipe for failure. Solve the problem in your native language. Write each of the steps in English or the language that you are most comfortable with. Translate this version into the programming language you are using. That way you will have the problem solved and you can focus on how to express your solution in the programming language.

Represent What?

An important part of solving any programming problem is to figure out how the problem will be represented in data. The variables that are needed and other structures like arrays need to be defined. This is a key aspect of creating a program to solve any programming problem.

What does it do?

Every computer program that was ever written does something. That is the nature of programs. They make computers do things that are desired by the programmer. Now sometimes the author of a program will get it wrong. The computer will do something that was not wanted. There can be lots of reasons for this and one of them is that the programmer really didn't understand what the computer was supposed to do for solving the problem. Make sure you understand what it is that you want the computer to do.

Make sure you understand the problem

If you don't understand what the problem is, how can you come up with a solution? You can't. Besides following directions, understanding what it is you must do is extremely important. If you don't understand what to do, don't even think about starting to work on the solution to the problem. Do whatever it takes to understand the problem you must solve.

If you can't solve the big problem, solve a smaller problem.

This may seem like the first statement (Break It Down) but it isn't. This has to do with complexity of the problem you are to solve. A problem might be so complex that you have no idea how to approach it let alone solve it. So, instead of bashing one's head against a wall (a tried and true method for solving problems), formulate a simpler problem related to the original problem. The simpler problem will be solvable (we hope). Use the learning that came from solving the simpler problem to solve the more complex problem.

Have a Plan B

No matter how well we plan and no matter how completely we plan, there is always something that happens to get in the way of our plan. Therefore if you have a way to solve the problem and you find you have travelled down a path that doesn't have a good or desired ending, you need to have an alternative. Plan B is that alternative. It may be an alternate way of solving the problem. It may be a different breakdown of the problem. It may be a simpler approach. Whatever it is, have a plan B just in case.

6. Why These 11 Statements?

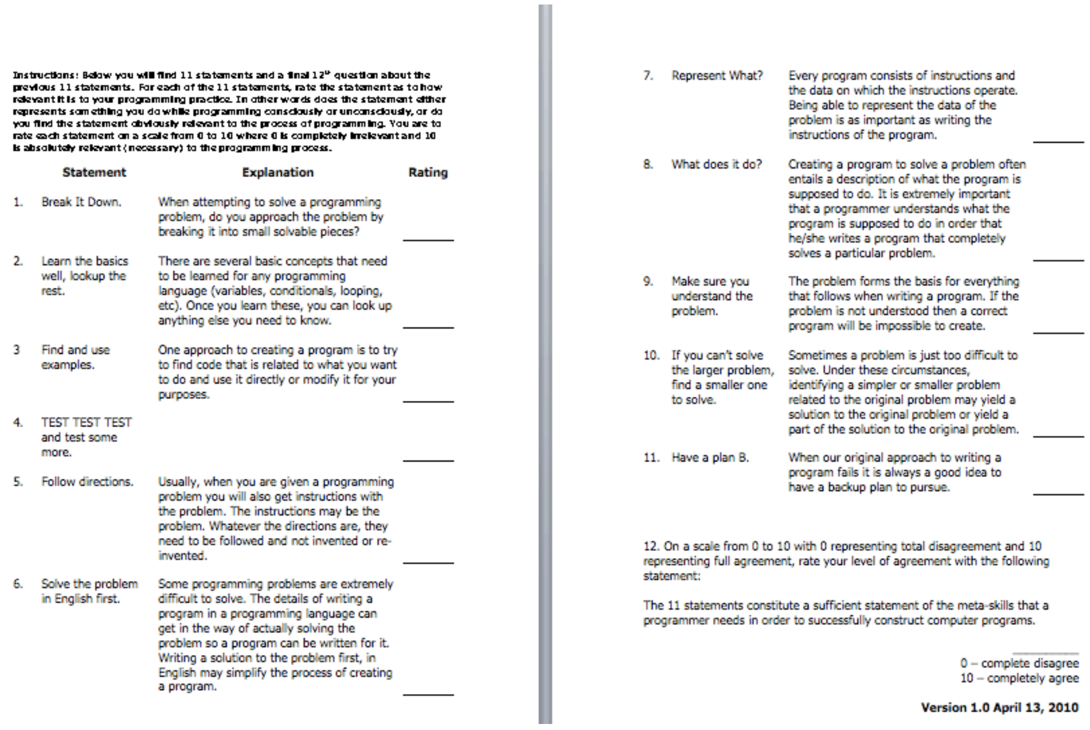
These 11 statements represent a codification of the author's experience in programming and teaching programming. It is extremely easy for students to get lost and frustrated when they first learn about programming. These statements are meant to give the student some strong foundations to stand upon. They address important aspects of the programming process and can be referred to as needed when students are solving programming problems.

The efficacy of these statements can only be measured over a long period of time. We want to know whether knowing these heuristics have the following effects on novice programmers.

> Do the statements result in lesser frustration when learning how to program or solving programming problems?
>
> Do the statements make the programming process clearer?
>
> Do the statements simplify the programming process?
>
> Do you find yourself referring to the statements with any frequency?
>
> Are the statements easy to remember?
>
> Can you apply the statements to all aspects of the programming process?

7. Preliminary Data

To begin the process of analyzing these statements and their relevance to the programming process we asked upper class students in a computer science program to rate the various statements and their relevance to the programming process. The students we asked have been programming for at least two years. Each student had experience programming with at least one programming language. The instrument used for this survey in shown in Figure 2.



Instructions: Below you will find 11 statements and a final 12th question about the previous 11 statements. For each of the 11 statements, rate the statement as to how relevant it is to your programming practice. In other words does the statement either represents something you do while programming consciously or unconsciously, or do you find the statement obviously relevant to the process of programming. You are to rate each statement on a scale from 0 to 10 where 0 is completely irrelevant and 10 is absolutely relevant (necessary) to the programming process.

	Statement	Explanation	Rating
1.	Break It Down.	When attempting to solve a programming problem, do you approach the problem by breaking it into small solvable pieces?	______
2.	Learn the basics well, lookup the rest.	There are several basic concepts that need to be learned for any programming language (variables, conditionals, looping, etc). Once you learn these, you can look up anything else you need to know.	______
3	Find and use examples.	One approach to creating a program is to try to find code that is related to what you want to do and use it directly or modify it for your purposes.	______
4.	TEST TEST TEST and test some more.		______
5.	Follow directions.	Usually, when you are given a programming problem you will also get instructions with the problem. The instructions may be the problem. Whatever the directions are, they need to be followed and not invented or re-invented.	______
6.	Solve the problem in English first.	Some programming problems are extremely difficult to solve. The details of writing a program in a programming language can get in the way of actually solving the problem so a program can be written for it. Writing a solution to the problem first, in English may simplify the process of creating a program.	______
7.	Represent What?	Every program consists of instructions and the data on which the instructions operate. Being able to represent the data of the problem is as important as writing the instructions of the program.	______
8.	What does it do?	Creating a program to solve a problem often entails a description of what the program is supposed to do. It is extremely important that a programmer understands what the program is supposed to do in order that he/she writes a program that completely solves a particular problem.	______
9.	Make sure you understand the problem.	The problem forms the basis for everything that follows when writing a program. If the problem is not understood then a correct program will be impossible to create.	______
10.	If you can't solve the larger problem, find a smaller one to solve.	Sometimes a problem is just too difficult to solve. Under these circumstances, identifying a simpler or smaller problem related to the original problem may yield a solution to the original problem or yield a part of the solution to the original problem.	______
11.	Have a plan B.	When our original approach to writing a program fails it is always a good idea to have a backup plan to pursue.	______

12. On a scale from 0 to 10 with 0 representing total disagreement and 10 representing full agreement, rate your level of agreement with the following statement:

The 11 statements constitute a sufficient statement of the meta-skills that a programmer needs in order to successfully construct computer programs.

______
0 – complete disagree
10 – completely agree

Version 1.0 April 13, 2010

Figure 2 – Survey used to Preliminarily Evaluate Programmer's Wisdom

Each of the statements is shown with a brief explanation. The statements are considered on a scale from 0 to 10 where 0 indicates a statement that has no relevance to the programming process, and where 10 indicates that the statement has significant relevance to the programming process. In addition a 12th question asks the student to rate all 11 statements in terms of their relevance to the programming process. A summary of the ratings given is shown in the next table.

Question		Average	S.D.
1	Break it down	7.9	3.0
2	Learn the basics well, lookup the rest	8.7	3.2
3	Find and use examples	7.6	2.7
4	Test, test, test and test some more	7.8	3.0
5	Follow directions	8.5	1.7
6	Solve the problems in English first	7.1	1.9
7	Represent What?	7.2	1.8
8	What does it do?	8.7	2.0
9	Make sure you understand the problem	8.8	1.1
10	If you can't solve the big problem solve a smaller one	8.1	2.1
11	Have a plan B	7.1	3.0
12		8.9	1.1

Table 1 – Student Wisdom Ratings (N=12)

Looking at this table it is clear that students who have programmed for some time consider the statements representative of the programming process. The lowest average score for a statement was 7.1. Two statements (6 and 11) had this score. The highest score, 8.8, was for statement 9.

8. Conclusion

In order to address the instruction of the necessary meta-skills for the programming process in a concise, memorable, understandable, and usable a series of 11 statements were created. These 11 statements represent a sample of the necessary meta-skills writing computer programs. The 11 statements are not meant to be comprehensive or even complete and one would suspect that over time the statements would possibly evolve or be modified as they are used. This evolution can only be accomplished if they are conveyed to novice students and then considered for their effectiveness.

My primary purpose in this paper is to present these statements as a starting point for others to utilize this formulation of meta-skills. To the extent these statements resonate with other teachers and these teachers use some form of them, the efficacy of these statements can be explored among a wider audience.

The compactness of these statements promotes their presentation in a short period of time – at most two classes. In this time frame, students can learn the wisdom (heuristics), memorize them and apply them to subsequent programming problems.

9. References

Blackwell, A. (2002). What is programming? PPIG 2002, MIT Press.

Brooks, F. P. (1995). The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Addison-Wesley Professional.

Buckingham, B. C., L. Hynd, et al. (2004). "Ways of Experiencing the act of learning to program: A phenomenographic study of introductory programming students at university." Journal of Information Technology Education **3**: 143-160.

Davis, A. M. (1995). 201 Principles of Software Development, McGraw-Hill.

Gries, D. (1974). What should we teach in an introductory programming course? Proceedings of the fourth SIGCSE technical symposium on Computer science education, ACM**:** 81-89.

Mayer, R. E. (1981). "The Psychology of How Novices Learn Computer Programming." ACM Comput. Surv. **13**(1): 121-141.

McConnell, S. (1993). Code Complete, Microsoft Press.

McIver, L. and D. Conway (1996). Seven Deadly Sins of Introductory Programming Language Design. Proceedings of the 1996 International Conference on Software Engineering: Education and Practice, IEEE Computer Society**:** 309.

Polya, G. (1988). How to Solve It A new Aspect of Mathematical Method, Princeton Science Library.

Shaft, T. M. (1995). "Helping Programmers Understand Programs: The Use of Metacognition." Database Advances **26**(4): 25-46.

Sheil, B. A. (1987). The psychological study of programming. Human-computer interaction: a multidisciplinary approach, Morgan Kaufmann Publishers Inc.**:** 165-174.

Shuhidan, S., M. Hamilton, et al. (2009). A Taxonomic Study of Novice Programming Summative Assessment. Eleventh Australasian Computing Education Conference (ACE2009) . Wellington, New Zealand.

Spolsky, J. (2004). Joel on Software, Apress.

Project Kick-off with Distributed Pair Programming

Edna Rosen, Stephan Salinger, and Christopher Oezbek

Institut für Informatik, Freie Universität Berlin
`edna.rosen, stephan.salinger, christopher.oezbek@fu-berlin.de`

Abstract *Background*: More and more software development companies decide to share their workload between teams which are geographically distributed. One of the biggest challenges is to start up work when new team members are introduced at a distant site of a global cooperation. Usually existing development processes do not cover integrating distributed collaboration, hence there is a need to adjust them to make project starts comfortable, easy and fast. A field study was conducted to introduce distributed pair programming (DPP), a derivative of pair programming (PP) in a distributed context, as a new development method to support communication and enhance knowledge transfer right from the beginning of the project. *Objective*: The objective of the study was to uncover relevant procedures and problems of establishing DPP and to collect supporting procedure steps for future project starts in distributed collaborations. *Methods*: A variation of canonical action research (CAR) was used to both establish DPP, gather insights and allow feedback from the developers involved. *Results*: This paper describes the establishment of DPP in a corporate project kick-off. It also reveals some benefits and major problems about distributed collaboration like conflicts in role fulfillment, ambiguity about session goals and missing awareness. *Limitations*: The validity of this study is threatened by the small number of participants and their particular cultural backgrounds.

Keywords: POP-I.A. distributed collaboration, POP-I.B. transfer of competence, POP-II.A. novice/ expert, POP-II.B. coding, POP-V.B. field study

1 Introduction

Software development in the twenty-first century cannot avoid the effects of globalization on production. One of the biggest challenges for distributed software development is to make knowledge available at all necessary locations quickly and efficiently (Braithwaite & Joyce, 2005; Herbsleb & Mockus, 2003). This becomes even more important if distributed collaboration separates the domain experts from newly assigned developers.

To enhance communication and knowledge transfer between stakeholders, a development practice like pair programming (PP) may be introduced for project kick-offs. Usually PP is part of other agile software development practices which are combined to a whole development process called extreme programming (XP) (Beck, 1999). Nevertheless it is also possible to introduce PP as a single new development practice without changing existing development processes (Aveling, 2004). In PP, two programmers work jointly while only using one computer, mouse and keyboard. The developers regularly change between two roles (Williams et al., 2000): One developer is taking the role of the 'driver', controlling the equipment, while the other developer, the 'observer', follows what the former is doing. Although engaging two developers with one task seems to be a lot of additional effort (e.g. Hannay et al., 2009; Nosek, 1998), PP has shown to increase communication and knowledge transfer between team members and to produce code of higher quality (e.g. Bipp et al., 2008; Hannay et al., 2009).

Due to newest technologies it is possible to perform PP in a distributed context, then called distributed pair programming (DPP) (Baheti et al., 2002; Stotts et al., 2003). DPP is similar to PP in that developers are joined (albeit virtually) to collaborate on a given task, but different in that co-developers each have their own computer, keyboard and mouse, which allows them to also work independently. With this advantage though, new challenges arise: non-verbal communication is limited and most actions of the co-developers are not instantly visible (Gutwin & Greenberg, 1999; Hanks, 2008). To bridge this, developer's actions must be made noticeable by awareness functionalities, e.g. code highlighting (Salinger et al., 2010).

In cooperation with the German IT companies Teles AG and her holding SSBG a field study was conducted to establish DPP as an additional development practice for a project kick-off between developers with little to no experience in DPP or PP (an expert in Vienna and two developers at a new office in Bangalore). At each of their local offices the developers still worked integrated in their local teams using a waterfall-based development process.

This paper delineates the establishment of DPP to support a corporate project kick-off. Section 2 highlights what is important about distributed software development and the establishment of a new development practice. Section 3 discusses the research setting including research background, research method and offers a short description of the technical infrastructure. Section 4 presents the results, lessons learned, and an overview of the most significant problems which occurred. Finally, Section 5 contains a conclusion of the establishment process.

2 Related Work

Since the rise of the Internet the software development industry has shown interest in distributed collaboration (Olson & Olson, 2000). Several scientific studies and industrial experience reports have dealt with the desire to optimize global cooperation (e.g. Damian & Lanubile, 2004), offering essential reasons which outline the necessity of distributed collaboration. Poole (2004) and Bass et al. (2007) refer to outsourcing, a desire to employ best available developers from any location, growing global open source communities as well as economic necessity such as cost competitiveness or product strategy, i.e. addressing specific market requirements. Yap (2005) additionally states sharing results and knowledge between locations as a reason for distributed collaboration.

Most of the studies and experience reports agree that there are three primary aspects for successful distributed collaboration (e.g. Poole, 2004):

First, the best applicable practices according to the needs of the development process have to be chosen. Distributed development practices like DPP or loosely coupled development methods such as distributed party programming (Salinger et al., 2010) allow the establishment of single practices in addition to existing development processes, while the establishment of distributed extreme programming (DXP) changes the entire development process to XP in a distributed context. Choosing the best development practices also depends on who will be involved in the distributed collaboration. Some companies may want to change the development practices to create a whole new distributed team from different locations (Yap, 2005), whereas others only bring together experts and newbies temporarily when necessary (Bass et al., 2007; Schümmer & Lukosch, 2008).

Second, the distributed process has to be adapted to integrate into an existing organization (Cohn & Ford, 2003). To this end different perspectives have to be assumed, for instance from the developers, management or other departments involved. Also cultural, psychological and social aspects need to be considered (e.g. Bass et al., 2007; Canfora et al., 2003).

Third, a technical infrastructure must be established. Such infrastructure includes the developing environment, collaboration tools, audio or video connection (Stotts et al., 2003). Individual tool preferences, platform restrictions as well as resource constraints, e.g. available bandwidth, should be considered (Schümmer & Lukosch, 2008).

In the field study conducted, DPP was temporarily established as an additional development practice to transfer expertise from one company site to a new team at a different site. Using DPP as a temporary practice enabled to focus on the developer's needs and establish and improve the technical infrastructure. Cultural, psychological and social aspects were only considered in case they could be attributed to experiences from existing studies.

3 Research Setting

3.1 Project Background

The project emerged as a cooperation between the Institute of Computer Science at Freie Universität Berlin and the German IT companies Teles AG and her holding SSBG. The IT company wanted to kick off a project between an office in Vienna and a new office in Bangalore and looked for a cost-effective and fast alternative to bring together their domain and programming expert from Vienna with the two newbies (experienced developers new to the company) from Bangalore. They wanted to transfer expertise from Vienna to Bangalore without having to change local development processes. To support the IT company in their distributed collaboration, the researchers provided a complete technical infrastructure including voice communication and a tool for DPP (see Section 3.3). This way the developers could easily work jointly and concurrently on their project and otherwise remain integrated in their local teams with individual tasks for each developer and a mostly sequential work flow. To support the developers in adopting DPP as a distributed development process and support the project kick-off, one researcher assumed the role of a process coach.

The main goal of the project kick-off was to develop a prototype based on a provisional specification of a voice recording solution for a VoIP application. To this end, the expert was expected to transfer his knowledge about the domain to his team colleagues at Bangalore and ensure a high level of understanding of the produced software artifacts and its interaction with existing parts for all participants. After the project kick-off, the new developers from Bangalore were supposed to be able to implement and maintain the project on their own.

To achieve these goals, the domain expert in cooperation with the researchers devised the following development process: First, tasks would be assigned by the expert mainly using the existing prescription from the waterfall-based model the company used, i.e. the new developers would receive tasks to implement independent components based on the provisional specification. Additionally critical parts of the project were developed by the expert to show and explain existing coding regulations. Second, it was decided to conduct regular DPP sessions of roughly two hours in length depending on the needs of the development process (which the expert decided to be once a week). Two different session types were envisioned by the expert: (1) The new developers would be asked to perform a code walkthrough (Freedman & Weinberg, 2000) of the code they had written so the expert could assess their progress and knowledge gains. Each developer was expected to present the code and mention critical and questionable points, giving the expert opportunity to comment on his interpretation of the specification in case of deviations. (2) The expert wanted to pair-develop (Williams & Kessler, 2000; Williams et al., 2000) the software, i.e. jointly create or edit artifacts of the software. The goal of this second type was to explain critical parts of the software to the new developers and provide opportunities for asking questions.

Starting in April 2009, sixteen weekly sessions using DPP were conducted over a period of four months. The expert participated in all these sessions, one of the new developers participated in three, the other in fifteen and both of them together in two sessions. The DPP setting was exploratively extended to three participants to test further benefits of DPP compared to co-located PP (as described by Salinger et al. (2010), the technical infrastructure deployed also allowed more than two participants). This was dropped by the developers not seeing any further advantage compared to the additional effort of a third developer involved. As one of the newbies was mainly involved in other tasks outside DPP sessions, he only participated a few times and later on information about the common project was transferred to him by the other newbie.

3.2 Research Method

The field study was conducted following principles from canonical action research (CAR) as described by Davison et al. (2004). Most important about this approach is that the researcher (in this case the process coach) and the participants are cooperating tightly and the direction of the collaboration can be influenced by either of these parties. This rather explorative research process is structured through several principles. It is based on an iterative process model (see Figure 1). With each iteration new insights are collected and interventions are planned to optimize the ongoing process. Process steps and iteration length can vary depending on individual demands of a particular research process.

One of the main goals of the researcher was to establish DPP according to the needs of the developers and overall goal of the project. This afforded to look at it from a researcher's perspective, i.e. the practicability of DPP in general as well as from the developer's point of view, i.e. the success of the planned development project. For initial background information about the developers involved, these completed a preliminary questionnaire about their experience with and expectations of DPP. Then the process started with a researcher-developer agreement, which continuously provided a general direction for the research process. In it the developers involved agreed upon project goals to be achieved, e.g. to create a prototype, and on the overall structure of the research process, e.g. when to meet for DPP sessions.

This initiated a cyclic process which contained three main steps (illustrated by the process model in Figure 1). The first step in each cycle was to determine the actual state of DPP usage with a focus on occurring problems, the project status regarding the overall project goals and the fulfillment of specific session goals. Hence before each upcoming session the developers completed a questionnaire stating their planned tasks, e.g. scope and duration, and what benefit or difficulties they expected. After each session they completed another questionnaire to evaluate the success of the session, i.e. whether their session goals were achieved and if expected benefits or difficulties emerged. Additionally, the process coach participated in each session and gathered information through observation and by analyzing log-files and videos recorded during the session post-hoc. Last, "reflection meetings" were held after each session to let developers discuss their experiences with DPP and to support the analysis with first-hand impressions.

In the second phase of each cycle (ideal state analysis) the previously identified problems or other phenomena such as last minute changes to planned tasks were analyzed to improve the use of DPP. This was done by the researchers using scientific literature and regular discussion. Afterwards the results were discussed with the developers in one of the following reflection meetings or sometimes in individual interviews, e.g. if only one developer was affected. Finally, new or adjusted process goals were evaluated and set based on the insights of the actual state to approach a more ideal state.

The third and last phase of each cycle consisted of planning and performing interventions according to the results of the ideal state analysis, e.g. changing audio settings for better quality.

At the end of the project kick-off the developers were asked to state their final impression of DPP, their experience with it, the overall project success, as well as anything else about DPP they found remarkable in a last individual interview.

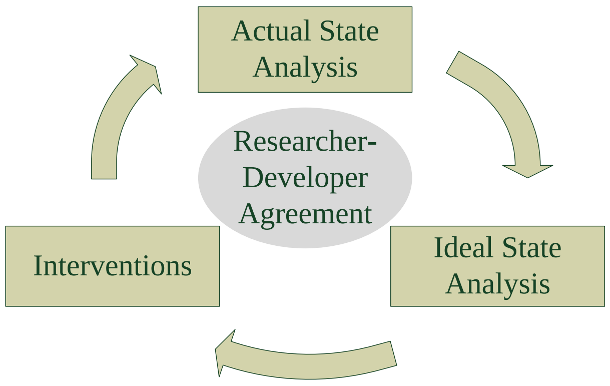


Figure 1. Cyclic research process model with three main phases based on a researcher-developer agreement

In summary, the following data sources were used to collect information about the DPP process:

- Observation: One of the researchers participated as the process coach in every DPP session.
- Preliminary questionnaire: One questionnaire was administered to the developers before the cooperation started to gather background information such as their level of experience with PP.
- Questionnaires before and after each DPP session: Each session was accompanied by a questionnaire about the session.

- Reflection meetings after each session or at least once a week: The developers and the process coach reflected together about the most recent session and discussed possible interventions for future sessions.
- Individual interviews: In total three individual interviews with the domain expert were conducted over the course of the collaboration.
- Final individual interviews: At the end of the project an interview with open questions about the overall success of establishing DPP was conducted.

3.3 Technical Infrastructure and Collaboration Tool

The different software packages necessary for conducting distributed pair programming were provided by the researchers and installed on a company server and on the computers of all participants. The packages consisted of the collaboration tool Saros[1] (Salinger et al., 2010), which integrates in the development environment Eclipse[2], the VoIP application Mumble and its server component Murmur[3], the instant messaging server OpenFire[4] and virtual private network client and server via OpenVPN[5].

The most central component in this setup is the collaboration tool Saros, which lets multiple developers work collaboratively in the development environment. Saros has been developed at Freie Universität Berlin by a team of students since 2006 (Salinger et al., 2010). To bring developers together for DPP, the software defines the concept of a session to which one participant in the role of the host can invite any number of participants as clients. The software then allows to share a software development project between all participants or synchronize existing copies to match the version provided by the host. During programming Saros closely models the roles of driver and observer known from PP by granting write access only to the driver and allowing the observer to follow the movement in files and package of a driver.

To increase awareness about the activities of the remote peers during a session, Saros also highlights the cursor, text selection, written text, visible viewport in a file, and the opened files in the project explorer. An annotated screenshot can be seen in Figure 2 presenting these options.

During the study all sessions were recorded on video both for scientific analysis and to improve Saros in combination with input from the participants and log-files generated by Saros.

4 Results and Lessons Learned

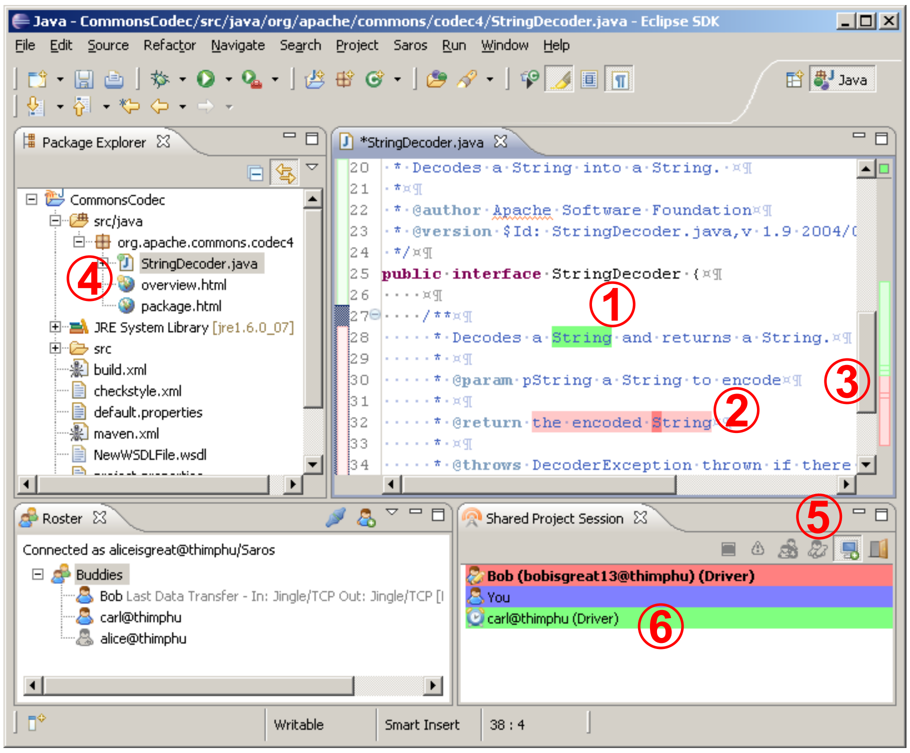
4.1 Session Overview

The first two of 16 sessions were used by the coach to explain the research process, introduce basic terms and processes of DPP and to show and explain the technical infrastructure to the developers. The following sessions mainly involved code walkthrough (sessions 3,7 and 9) and pair-developing including ad-hoc testing (sessions 4-6, 8 and 10-16).

In the first few research process cycles primarily a lot of adjustments in the technical infrastructure were needed, such as adjustments to audio settings and equipment to improve audio quality. Since Saros had never been deployed in an industrial scenario between continents before, it was necessary to find workarounds and software updates had to overcome several problems such as improving the synchronization of large projects.

After the fifth session a fixed starting time and duration of 90 minutes per session was set. Before, the expert had chosen a starting time and duration according to his planned task, which did not take into consideration any delays caused by developers not being on time, the synchronization of their large project, or answering general questions. Figure 3 gives an overview of the time spent in each session

[1] Available for download at `http://dpp.sourceforge.net/`.
[2] `http://www.eclipse.org`
[3] `http://mumble.sourceforge.net/`
[4] `http://www.igniterealtime.org/projects/openfire/`
[5] `http://openvpn.net/`


Java - CommonsCodec/src/java/org/apache/commons/codec4/StringDecoder.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Saros Run Window Help
Java
Package Explorer
CommonsCodec
src/java
org.apache.commons.codec4
StringDecoder.java
overview.html
package.html
JRE System Library [jre1.6.0_07]
src
build.xml
checkstyle.xml
default.properties
maven.xml
NewWSDLFile.wsdl
*StringDecoder.java
1
2
3
4
5
6
Roster
Connected as aliceisgreat@thimphu/Saros
Buddies
Bob Last Data Transfer - In: Jingle/TCP Out: Jingle/TCP [I
carl@thimphu
alice@thimphu
Shared Project Session
Bob (bobisgreat13@thimphu) (Driver)
You
carl@thimphu (Driver)
Writable
Smart Insert
38 : 4

```
 * Decodes a String into a String.
 *
 * @author Apache Software Foundation
 * @version $Id: StringDecoder.java,v 1.9 2004/0
 */
public interface StringDecoder {

    /**
     * Decodes a String and returns a String.
     *
     * @param pString a String to encode
     *
     * @return the encoded String
     *
     * @throws DecoderException thrown if there
```

Figure 2. Various awareness features used in Saros such as (1) selection, (2) text edits, and (3) viewports highlighted in each users' color, (4) opened and active files by current drivers, (5) button for following the viewport of a driver, and (6) information about Eclipse being the foreground window.

and shows that preparation time declined as the developers became more experienced (with the notable exception of session 14 in which technical problems prevented any development to be started, but still allowed the developers to discuss changed requirements and to plan ongoing work).

During later research process cycles the focus then could be shifted to improving the use of DPP (described in the following sections) and on optimizing the research method, e.g. finding an ideal time for reflection meetings, and improving communication between developers. Planning reflection meetings was demanding because it was necessary to get all participants together and not let too much time go by after the sessions to be reflected. Sometimes only a few days after a session the developers would not remember what had happened in their last session. Finally, performing one reflection meeting after the last session of the week showed to be most effective.

4.2 Benefits of DPP

In twelve of the thirteen post-session questionnaires the developers declared the session a success and stated that most of the times DPP had been helpful to achieve their session goals (the expert agreed for 85% of sessions, the newbies for 89%). In the final interview they confirmed that their project goals were achieved.

Over the course of the study the following three benefits appeared to most prominently support the use of DPP:

– First of all, communication between developers was enhanced noticeably throughout the collaboration. Before the DPP sessions started, communication was limited to chat or e-mail. Due to DPP sessions and reflection meetings the developers talked to each other at least once a week for more than one hour. Moreover, communication was enhanced due to newly introduced walkthroughs and pair-developing in DPP sessions which were supplementary to their local, usually rather loosely coupled, development process. The developers used the session time to ask questions or discuss open

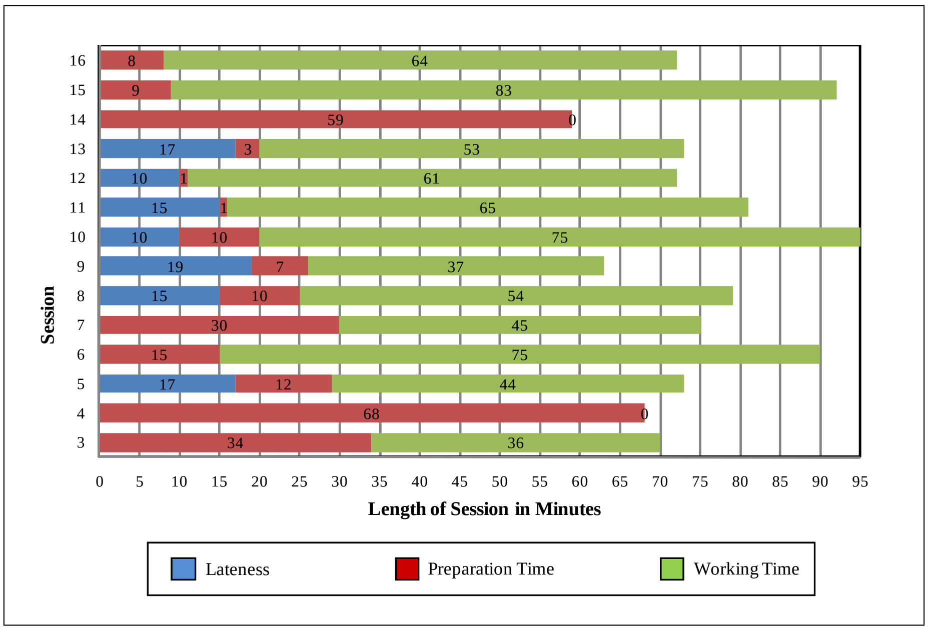


Figure 3. Length of DPP sessions as a breakdown of time lost due to developers coming late and having to prepare for the session, and time spent on productive work.

issues with the respective artifacts in plain view. In reflection meetings the developers also talked about the establishment of DPP in general, i.e. their experience and expectations, or discussed procedural strategies. In earlier sessions the driver, who was most of the times the expert, talked more than the observer. Nevertheless, between tasks or during project synchronization the developers used the pauses to reorder assignments or agree on general procedures. Later on in the collaboration the new developers integrated themselves more actively in sessions by making suggestions or discussing development issues.

– Second, the expert stated in eight of thirteen questionnaires after sessions that he transferred all important information to his co-developer. This was either in sessions where he was pair-developing or in walkthroughs of code previously created by one of the newbies outside the session giving feedback on requirements of the provisional specification. In most sessions the expert continued to write code simultaneously to explaining and teaching his partners, thereby showing a second benefit of DPP as combining teaching with productive work.
– Third, one of the newbies who presented his code in one of the walkthrough sessions stated in the questionnaire after the session that the task was accomplished faster than expected because there was no delay in the feedback. In the reflection meeting that followed he stated that feedback without DPP (through e-mail and chat) would have been much more complicated in comparison. Another benefit of DPP was observed in the sessions when errors of the driver could be avoided. In at least five of the pair-developing sessions the observer made the driver aware of errors and thereby avoid them, e.g. when the driver was about to write code in the wrong artifact or when the declaration of a variable he was about to add already existed.

4.3 Problems with Establishing DPP

Three of the problems that occurred in the establishment process could not be resolved even after interventions. These were (1) conflicts with role fulfillment, (2) ambiguity about session goals, and (3) missing awareness, which will be discussed in turn. The fact that development happened distributedly possibly influenced all of these problems. Since the developers did not have common office hours or did not have the possiblity to work co-locatedly might have had intensified these problems. Especially since the expert stated that he usually waits until common lunch breaks to discuss issues with colleagues, which did not happen in this project due to the distance.

Conflicts with Role Fulfillment A first problem arose in the context of assuming the roles of driver and observer, which in DPP—as in PP—stipulate the responsibilities of each developer during a joint session. For instance, XP recommends the driver to attend to details during programming and the observer to keep the top-level concerns in mind (Beck, 1999). Nevertheless role assignment in DPP as well as PP is not self-explanatory and hence can be challenging (Bryant et al., 2008).

The first indication of a problem with role assignment and fulfillment appeared in one of the first pair-developing sessions when the expert stated in the questionnaire after the session that he had the driver role for too long. As it is not unusual for an expert to assume the driver role and keep it over longer times than usually recommended by XP (Schümmer & Lukosch, 2008), this point was brought up in an interview with the expert to explore the reasons for his statement. The expert explained that he considered more regular role switching important to achieve the benefits of DPP. In particular he noted that he felt very exhausted from being driver most of the time and had had little opportunity to assess the knowledge gains of the new developers.

In the next pair-developing session the expert tried to hand over the driver role to one of the newbies. Yet, the newbies did not react to his request to assume the driver role and the expert continued being driver. The developers made no attempt to talk about this in the next reflection meeting. Hence the coach confronted the new developers with the issue. The newbies stated that from their point of view they did not have enough knowledge about the artifacts and did not feel comfortable being observed when taking over the driver role. Although the expert had expressed his wish for more role changes during sessions, the developers did not want to set up more formal rules for role fulfillment. Instead they agreed on finding spontaneous solutions on occasions in the session when necessary. In later pair-developing sessions it was observable that the newbies could integrate themselves more actively in the sessions by asking questions or making suggestions for code changes, although most of the times they still refused to take the driver role. Additionally, the final interview with one of the newbies revealed that he still had not understood how a second developer could integrate himself actively in a DPP session and that for him the observer role was mainly boring.

In a qualitative analysis based on the data collected through observations in the sessions, questionnaires, personal interviews, reflection meetings, and a discussion with the developers in the final interview it was attempted to build a conceptual model around the role behaviors following the paradigm model of Strauss & Corbin (1990) to distinguish causal conditions, intervening conditions and consequences.

Three primary causal conditions for the problems with role fulfillment could be identified from the data (see Figure 4): (1) The developers had *little experience with DPP and PP* and thus did not know what the distinct responsibilities of the roles were, when role changes would be best suitable and what benefits would arise from following the prescriptions of a process model such as XP on role segmentation or responsibilities. (2) The developers showed *little role consciousness* during the sessions, but often only mentioned problems when explicitly queried after the session. Thus instead of resolving conflicts during the sessions, the reflection meetings were necessary to raise the awareness of the developers on these issues. (3) *Diverging expectations* between the expert and the new developers caused them to require or reject different aspects of the roles. While the expert wanted to transfer the driver role to the new developers so they would practice writing code under his support, the new developers rather felt under observation and thus feared being caught making a mistake.

These causes are certainly dependent on each other—increased DPP experience in particular should both align expectations and raise consciousness about role mismatches—but sufficiently different in the way they could be addressed to be given separately here. Several other minor causes could be identified, but none of them had sufficient explanatory value.

As an intervening condition, pair-pressure (Williams, 2000) was identified. Usually such pressure is welcomed by the developers in a PP session because it increases concentration and helps developers push each other to complete their task (Williams, 2000). Unfortunately the developers from Bangalore could not benefit from this pressure, but rather their expectations diverged further and their consciousness about the development process gave way to feeling uncomfortable to be observed while coding in a domain in which they were not experienced.

The consequence of suboptimal role usage is primarily to be seen (1) in *less productive sessions*, in which less code was produced and less knowledge transferred, (2) *boredom* and (3) *exhaustion* on the side of the new developers and the expert respectively.

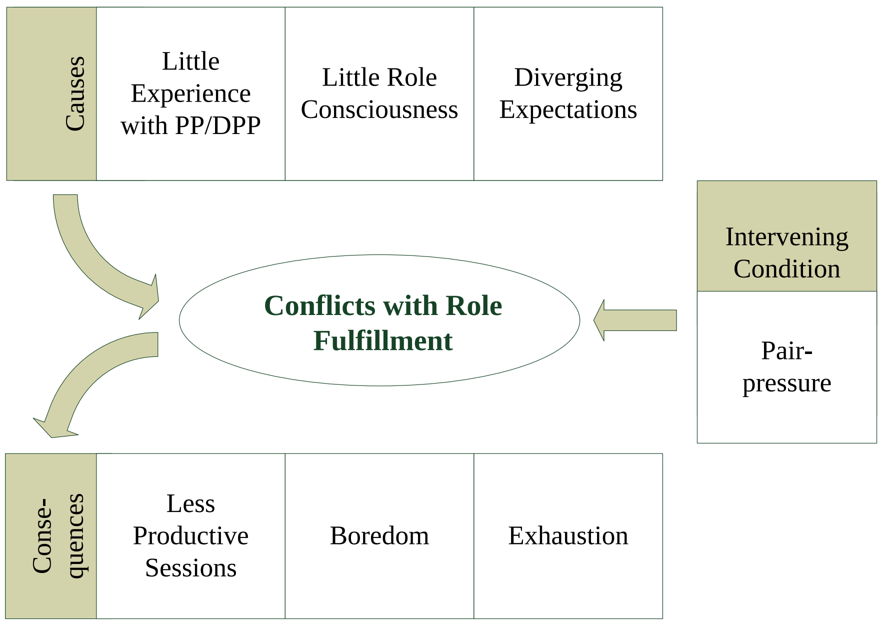


Figure 4. Possible causes, consequences and intervening conditions of conflicts in role fulfillment.

Lesson learned: Although problems in role fulfillment were identified in an early stage of the establishment of DPP and discussed in several reflection meetings, they could not be resolved. It seems as if it was not enough to know and talk about problems with role conflicts in reflection meetings, if the developers involved are not conscious enough about their conduct during the sessions. Evidently it happens that participants do not discover or experience the benefits of a new practice. Therefore it is suggested to try to improve knowledge about PP and DPP topics in particular before and during the process of establishing DPP. Moreover, it should not be ignored that the success of role fulfillment in DPP also depends on the fears and attitudes of the participating developers.

Ambiguity about Session Goals A second problematic phenomenon associated with DPP could be traced to the goals associated with individual sessions and is best explained with one particular session early on in the project: For this session the expert had scheduled a “code review” with one developer and his newly written code. In the questionnaire before the session the expert clarified his goal for the session as “code review presented by [name of newbie] followed by a discussion”, while the new developer stated as the goal “get the code reviewed by [name of expert] and discuss the open issues if any”. Thus by using the unqualified term “code review” when scheduling the meeting, the expert had inadvertently introduced ambiguity into the session goals, which led the new developer to believe that the expert would be primarily responsible for conducting the review. When the expert at the beginning of the session then asked the new developer to start presenting his code, the new developer silently concurred with the request, but obviously was not prepared for this task: First, he stated that he found it hard to find a good position in the code to start with. Then, a lot of times during the presentation he had to jump back and forth between different artifacts, his explanations were sometimes halting and several times he stated in the middle of an explanation that he forgot to mention some precondition.

After the session the expert stated in his questionnaire that the task was completed slower than expected. He attributed this to the large amount of code to be reviewed, ignoring the halting pace of the presentation, maybe not even aware that the newbie had been surprised by the expert's interpretation of the session goal and unprepared for a code presentation.

The next session designated as a code review then revealed how much time had been wasted by the ambiguous statement of performing a code review in the first session. Here the new developer had adjusted to the expert's interpretation of the goal and was excellently prepared to present the code.

Analyzing all DPP sessions which occurred during the project revealed that problems associated with ambiguous goals were more prevalent than this single example. Conceptualizing these incidents resulted in the following three causes to be identified for ambiguous session goals (see Figure 5):

The first cause of ambiguity in session goals, exemplified by the above example, was the *lack of precise communication*. In the above case it would have been sufficient to either describe in a single short sentence what a code review would entail or use the term walkthrough, which usually connotates the author to present the code (Freedman & Weinberg, 2000).

The second cause identified was the existence of *diverging project goals*. While the expert wanted to transfer much of his applicable domain expertise over the duration of the project, a goal to receive such domain knowledge was never mentioned by the newbies. The newbies' primary goal as stated both in the initial questionnaire and the final interview was rather to create code that was executable. This caused for instance a newbie to state after session 14, in which technical problems kept the developers from coding, that the session was not successful, although the expert stated it was a partial success because important questions were discussed and knowledge transfer had taken place.

The third reason why session goals often were ambiguous was session planning was conducted on *short notice*. In most cases the expert contacted the newbies only about one hour before the session to announce what the planned task would be. Since these announcements came in just before the newbies' lunch break (due to different time zones), this shortened their possible preparation time to zero. In an individual interview the expert stated that announcements were made on short notice after considering the latest status of the development process. He stated that he would not change this, ignoring the hint that this might allow the newbies more time for preparation. The newbies thus remained in a position in which the session goals were unknown to them until the very last moment.

Two intervening conditions could be identified to affect ambiguity in session goals: (1) The use of *instant messaging chat* aggravated the misunderstandings arising from the *lack of precise communication*, in particular because chat messages are more terse than voice or e-mail communication, are not persistently stored, and lack sufficient detail. (2) The emphasize of *flexibility over structure* within a session further amplified the ambiguity of session goals. For instance, the developers once performed lengthy ad-hoc testing of newly written code in a pair-coding session and thereby introducing a newly emergent goal of increasing quality to the existing goal of producing code for a certain feature.

The consequences of ambiguous goals ultimately were *wasted time and resources*—as the difference between the unprepared and slow walkthrough and the improved second session shows— and *inadequate session results*. Yet, more practically the lack of clarity about session goals ahead of time caused (1) *incorrect and insufficient preparation* for sessions by developers, and (2) led to an *invalidation of contributions* (the expert's lengthy explanations for instance seem a waste of time under the developers' assumption that the goal is primarily to produce code as fast as possible).

Lessons learned: Misunderstandings leading to ambiguity in session goals cannot be completely avoided. Nevertheless a short preparation sufficiently ahead of time to align the understanding of session goals, in particular who is in charge of what, can economize session time. Better aligned session goals are also more likely to increase satisfaction with a session as the contributions of all participants will integrate better and can be more easily valued by all parties.

Missing Awareness It is well known that shared awareness is a crucial element of distributed collaboration (Gutwin & Greenberg, 1999; Olson & Olson, 2000), where awareness describes the consciousness about one's own and the other participants' actions in the context of the collaborative environment

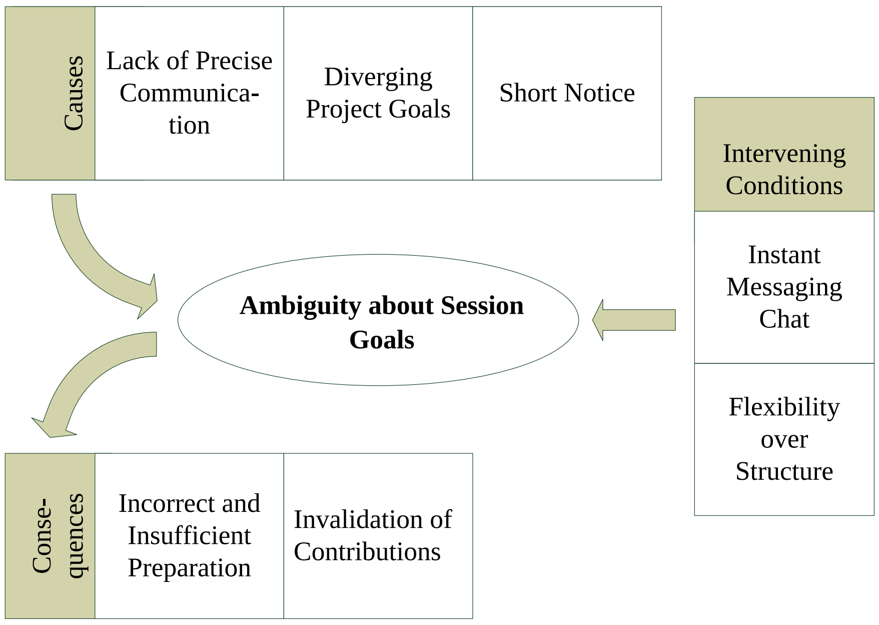

Causes
Lack of Precise Communica-tion
Diverging Project Goals
Short Notice
Intervening Conditions
Instant Messaging Chat
Flexibility over Structure
Ambiguity about Session Goals
Conse-quences
Incorrect and Insufficient Preparation
Invalidation of Contributions

Figure 5. Possible causes, consequences and intervening conditions of ambiguity in session goals

(Dourish & Bellotti, 1992). Awareness can be challenging to attain at the beginning of a project when co-developers have never worked together and there is no common understanding of development strategies yet. Enhancing awareness may have a positive effect on distributed collaboration (Gutwin et al., 1996).

In DPP sessions awareness can be provided by the technical infrastructure as well as by the co-developers. Gutwin et al. (1996) have made an attempt to divide awareness into different categories such as technical awareness, workspace awareness, or social awareness (Gutwin et al., 1996). Workspace awareness covers all types of awareness which help the developers to find the location and actions of other co-developers in the developing environment. Awareness of the categories social or group-structural awareness cover social interactions such as expectations and abilities, e.g. if developers are aware of what other developers expect them to do next. Identifying the latter can be difficult, because it depends on the motives of the developers involved which are not always known. In the analysis of problems with role fulfillment mentioned above the motives of the newbies to not take over the driver role were questionable in that matter. It was not verifiable whether the newbies did not know what was expected of them or if they just pretended not to know. In the first case it would be a matter of awareness, in the latter merely ignorance. Although other forms of awareness were an issue in the field study conducted, this subsection will focus on missing awareness in terms of workspace awareness.

Three causes for missing workspace awareness could be identified by analyzing the collected data (see Figure 6):

(1) *Weaknesses in the technical infrastructure*: The technical infrastructure deployed in the field study and in particular Saros had never been analyzed in commercial software development before and did not include a video connection showing the remote partner's face, following advice from other studies (e.g. Baheti et al., 2002) in which developers had stated that the video connection had not resulted in additional benefit.

Analyzing the events of the DPP sessions showed that a video connection would have been favorable. Missing non-verbal communication (in distributed collaboration made visible through a video connection) was one of the reasons why actions of the observer were not always detectable in sessions. In earlier pair-developing sessions during the establishment of DPP in which the expert was the driver throughout the whole session, long phases were observed in which the observer's actions were not detectable. Sometimes background noises in sessions suggested that the newbie was distracted, e.g. a mobile phone rang or laughter could be heard over his microphone. On rare occasions one of the newbies noticeably started talking to one of his co-located colleagues, even provoking the expert to comment on it.

The problem of possible distraction of the observer was discussed in the following reflection meeting. The newbie stated that he could not remember these episodes of the session. To get more information about the newbie's behavior during pair-developing sessions, without breaking the limits of bandwidth, it was planned to install an additional desktop sharing application between him and the coach. For different reasons the desktop sharing application was never installed. Motivated by the problem, the Saros developers then improved the collaboration tool to show whether the development environment was the foreground window (in contrast to a browser for instance). The expert did not comment on any behavior of the newbies concerning their distractions during sessions. However in later sessions he demanded more frequently the observer's attention. This was noticeable through the amount of questions he asked the observer, e.g. about the location of the observer in the workspace or "what would you do in this case?".

(2) The second cause identified for missing awareness was the plain *non-use of awareness functionalities* provided by the collaboration tool. This was noticeable in sessions when the observer asked the driver at what location the former was, instead of using one of the Saros options, e.g. activating the "automatic follow mode" or double-clicking on the remote partner's name in Saros. Even when the developers used the follow mode, they used line numbers for orientation in the code, e.g. when asking questions about the code: "here in line number 500 we have an event", instead of marking the code with their cursor and thus highlighting it for the co-developer. Neither regular updates about existing and new Saros functionalities nor pointing out their advantages, e.g. preventing delays in the session by using the automatic follow mode and hence not having to ask for positions of the co-developers, could convince the developers of using the awareness functionalities provided by Saros.

(3) A third reason for missing awareness was identified as *lack of talk-aloud*, which was identified in one of the rare occasion when one of the newbies became driver in a session: The code he was typing in this situation was not being displayed in the observer's view because of a technical problem. Additionally the driver had not verbalized his actions and hence made it impossible for the observer to be aware that the driver was about to start writing code at a particular position. Taken together, the technical problem and the *lack of talk-aloud* led to the consequence that it took five minutes until the technical problem was noticed. When the expert as the host of the session tried to remedy the technical problem, he inadvertently overwrote the text written by the new developer, unaware of the work the new developer had already invested.

One important intervening condition which increased the problem of missing awareness was the *lack of role changes* during the sessions. Since the new developers rarely received the driver role (only in half of the pair-developing sessions, but never for more than five minutes), they also had little experience with managing awareness from both perspectives. More troubling, they in general did not expect role changes, as one of them stated in the final interview, and therefore did pay less attention to the action of the driver in the workspace, since they felt sure that only a passive role would be required from them. The fact that there was some evidence of the observer being distracted from sessions and that the observer had to ask for the position of the driver several times during sessions also led to this conclusion.

Figure 6 shows the discovered conceptual relationships and several consequences of missing awareness in the DPP sessions.

Lessons learned: Any comment from driver as well as observer can enhance awareness in DPP sessions. As stated by Stotts et al. (2003), constant exchange of information about what developers can see and commenting on actions keeps attention higher and avoids missing awareness. As later sessions showed, this can be achieved by the driver demanding frequent feedback through questions from the observer and should be enhanced by frequent role changes. Additionally, desktop sharing and/or a video connection should be integrated in the technical infrastructure to counteract missing awareness and to achieve more detailed and faster feedback between co-developers (Schümmer & Lukosch, 2008). Motivated by the requirements of additional awareness functionalities in the collaboration tool, the implementation of a desktop sharing functionality for Saros was started shortly after the project kick-off (Salinger et al., 2010).

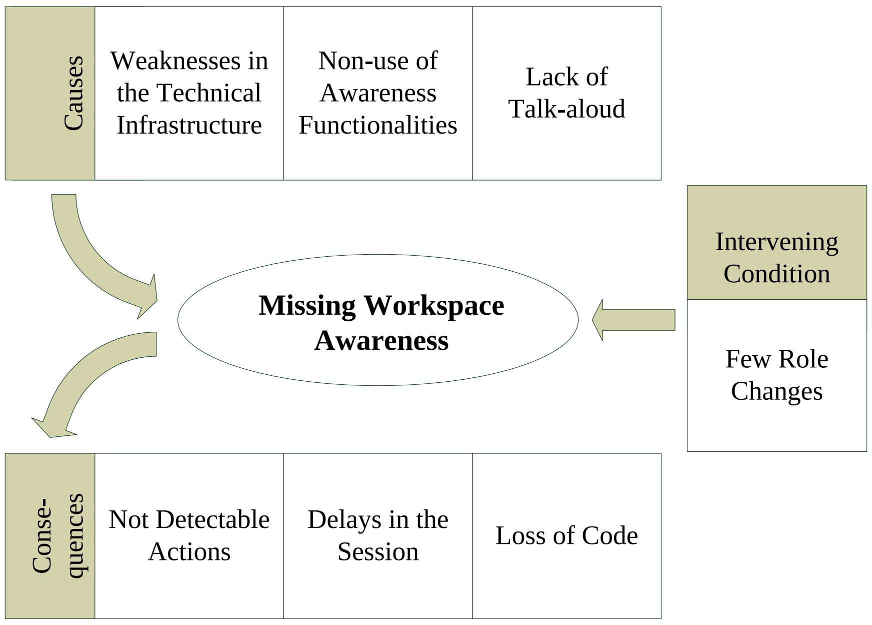


Figure 6. Possible causes, consequences and intervening conditions of missing workspace awareness

5 Conclusion

After sixteen DPP sessions the goal of the kick-off to develop a prototype was successfully accomplished within the given deadline and the development responsibilities were given exclusively to the Indian development team.

The field study conducted showed that DPP can be established and integrated into an existing development processes to support distributed collaboration in a project kick-off. The research method made it possible to constantly improve distributed collaboration and at the same time incorporate the developers' requirements. Frequent questionnaires and reflection meetings in combination with observations were essential sources of data collection and analysis. Constant improvement of the technical infrastructure according to the requirements of the development process, e.g. adjusting audio quality or introducing new awareness features of the collaboration tool, had a positive effect on the collaboration.

The analysis of the data collected confirmed already known insights, but also uncovered new problems about the establishment of DPP. Benefits such as a high level of communication, combining knowledge transfer and productive work as well as reduced delay in feedback supported the distributed collaboration and hence the project kick-off. Some of the lessons learned were that solving problems such as conflicts in role fulfillment, ambiguity in session goals and missing awareness can be challenging, if solving them is against the developers priorities.

For future work we are looking to replicate the study in other companies and distributed development settings to follow up first indications and other aspects such as inter-cultural influences about distributed collaborations.

Bibliography

Aveling, B. (2004). XP Lite considered harmful? In *Proceedings of the International Conference on Extreme programming and Agile Processes in Software Engineering (XP 2004), Garmisch-Partenkirchen*, volume 3092/2004 of *Lecture Notes in Computer Science*, (pp. 94–103)., Berlin / Heidelberg. Springer.

Baheti, P., Gehringer, E., & Stotts, D. (2002). Exploring the efficacy of Distributed Pair Programming. In *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, volume 2418/2002 of *Lecture Notes in Computer Science* (pp. 387–410). Berlin / Heidelberg: Springer.

Baheti, P., Williams, D. L., Gehringer, E., & Stotts, D. (2002). Exploring pair programming in distributed object-oriented team projects. In *OOPSLA Educator's Symposium, Seattle, WA*.

Bass, M., Herbsleb, J. D., & Lescher, C. (2007). Collaboration in global software projects at siemens: An experience report. In *Proceedings of the International Conference on Global Software Engineering (ICGSE 2007)*, (pp. 33–39)., Washington, DC, USA. IEEE Computer Society.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

Bipp, T., Lepper, A., & Schmedding, D. (2008). Pair programming in software development teams - an empirical study of its benefits. *Information and Software Technology*, *50*(3), 231–240.

Braithwaite, K. & Joyce, T. (2005). Xp expanded: Distributed extreme programming. In *Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005)*, (pp. 180–188)., Berlin / Heidelberg. Springer.

Bryant, S., Romero, P., & du Boulay, B. (2008). Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, *66*(7), 519–529.

Canfora, G., Cimitile, A., & Visaggio, C. A. (2003). Lessons learned about distributed pair programming: what are the knowledge needs to address? In *Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, (pp. 314–319)., Los Alamitos, CA, USA. IEEE Computer Society.

Cohn, M. & Ford, D. (2003). Introducing an agile process to an organization. *Computer*, *36*(6), 74–78.

Damian, D. & Lanubile, F. (2004). The 3rd international workshop on global software development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, (pp. 756–757)., Washington, DC, USA. IEEE Computer Society.

Davison, R., Martinsons, M. G., & Kock, N. (2004). Principles of canonical action research. *Information Systems Journal*, *14*(1), 65–86.

Dourish, P. & Bellotti, V. (1992). Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, (pp. 107–114)., New York, NY, USA. ACM.

Freedman, D. P. & Weinberg, G. M. (2000). *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. New York, NY, USA: Dorset House Publishing Co., Inc.

Gutwin, C. & Greenberg, S. (1999). The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Trans. Comput.-Hum. Interact.*, *6*(3), 243–281.

Gutwin, C., Greenberg, S., & Roseman, M. (1996). Workspace awareness in real-time distributed groupware: Framework, widgets, and evaluation. In *People and Computers XI*, (pp. 281–298)., Berlin / Heidelberg. Springer-Verlag.

Hanks, B. (2008). Empirical evaluation of distributed pair programming. *International Journal of Human-Computer Studies*, *66*(7), 530–544.

Hannay, J., Dybå, T., Arisholm, E., & Sjøberg, D. (2009). The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, *51*(7), 1110–1122.

Herbsleb, J. D. & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, *29*(6), 481–494.

Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM*, *41*(3), 105–108.

Olson, G. M. & Olson, J. S. (2000). Distance matters. *Human-Computer Interaction*, *15*(2), 139–178.

Poole, C. (2004). Distributed product development using extreme programming. In *Extreme Programming and Agile Processes in Software Engineering*, (pp. 60–67).

Salinger, S., Oezbek, C., Beecher, K., & Schenk, J. (2010). Saros: An Eclipse plug-in for distributed party programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. ACM. To appear.

Schümmer, T. & Lukosch, S. (2008). Supporting the social practices of Distributed Pair Programming. In *Groupware: Design, Implementation, and Use*, volume 5411/2008 of *Lecture Notes in Computer Science* (pp. 83–98). Berlin / Heidelberg: Springer.

Stotts, D., Williams, L., Nagappan, N., Baheti, P., Jen, D., & Jackson, A. (2003). Virtual teaming: Experiments and experiences with Distributed Pair Programming. In *Extreme Programming and Agile Methods — XP/Agile Universe 2003*, volume 2753/2003 of *Lecture Notes in Computer Science* (pp. 129–141). Berlin / Heidelberg: Springer.

Strauss, A. L. & Corbin, J. M. (1990). *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE.

Williams, L., Kessler, R. R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software*, *17*(4), 19–25.

Williams, L. A. (2000). *The collaborative software process*^{SM}. PhD thesis, The University of Utah. Adviser-Kessler, Robert R.

Williams, L. A. & Kessler, R. R. (2000). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM*, *43*(5), 108–114.

Yap, M. (2005). Follow the sun: Distributed extreme programming development. *Agile Development Conference/Australasian Database Conference*, *0*, 218–224.

The use of MBTI in Software Engineering

Rien Sach
Maths and Computing Faculty
Open University
Milton Keynes, MK7 6AA
United Kingdom
r.j.sach@open.ac.uk

Marian Petre
Maths and Computing Faculty
Open University
Milton Keynes, MK7 6AA
United Kingdom
m.petre@open.ac.uk

Helen Sharp
Maths and Computing Faculty
Open University
Milton Keynes, MK7 6AA
United Kingdom
h.c.sharp@open.ac.uk

Keywords: POP-I.A. Group Dynamics, POP-II.A. Personality, POP-VI.F. Exploratory

Abstract

In this paper we evaluate the use of Carl Jung's theories of Psychological Type assessed using the Myer-Briggs Type Indicator in the Software Engineering field. The current level of implementation and its quality is established and the results discussed to provide insight into what we currently know, and suggestions on what could be important to investigate for the future.

Upon gathering MBTI data from a range of sources it is apparent that there is agreement on the types of personalities often discovered inside software engineering. *Thinking* and *judging* personality preferences are commonly found, while *feeling* and *perceiving* is far less common. This differs substantially from results representative of the American population, and supports the belief that software engineers are more commonly represented by specific *types* of people.

However, there is discrepancy between four of the 16 types identified in the MBTI, suggesting that there is still some understanding to be gained about personality in software engineering, and we do not by any means know the exact breakdown of types present within the industry.

1. INTRODUCTION

Software Development has been an expanding market for over 40 years, and it is estimated that the global software market grew by 6.5% in 2008 and is now valued at $303.8 billion [1]. It is also predicted that by 2013 the global software market will be valued at $457 billion [1].

Personality is a term used to describe the behaviour, traits and character of an individual, and can be used to suggest how different individuals process situations and events [5]. Each individual's different personality relates to how they prefer to use their mind, and this can explain apparently random behaviour and differences [3].

The Myers-Briggs Type Indicator has been used for over 50 years to identify the personality type of an individual and their personality preference, making the theories of Jung useful and applicable to everyday life.

In this paper we present the compiled results of 5 MBTI assessments on software engineering practitioners, and what this tells us about the personality of a software engineer. We then proceed to compare the compiled results with previously published results to draw conclusions and comparisons.

The purpose of this paper is to identify the current level of published data on MBTI assessments administered specifically to software engineers, their quality, validity, and what they tell us and suggest about the personalities of software engineering practitioners.

1.1 Background

The Myers-Briggs type indicator was developed by Katherine Briggs and her daughter Isabel Myers from the theories first published in 1921 by Carl Jung [6]. The MBTI (Myers-Briggs Type Indicator) was first published in 1962 [7], and has become a widely accessible and used tool in assessing a person's personality type.

The Myers-Briggs type indicator has become the most widely used personality inventory, with over 3.5 million assessments administered worldwide each year [7, 8].

ISTJ	ISFJ	INFJ	INTJ
ISTP	ISFP	INFP	INTP
ESTP	ESFP	ENFP	ENTP
ESTJ	ESFJ	ENFJ	ENTJ

Table 1. Possible MBTI Types

Carl Jung's theory on psychological type states that there are two *worlds* in which we can focus our minds (**Extraversion**, **Introversion**), when we are *using* our mind we are either taking in information (**Perceiving**) or processing this information and drawing conclusions from it (**Judging**).

Additionally we can *perceive* in two ways, by living in the present and focusing on what is real and actual (**Sensing**), or looking towards the future and the possibilities (**iNtuition**). And when we are *judging* this can also be done in two ways, by looking at the logical consequences and being analytical (**Thinking**), or by looking at what is important to ourselves and others and assessing the impact on people (**Feeling**).

These four pairs of scales produce the 16 possible Myers-Briggs types indicated in Table 1, and Figure 1 represents the four dichotomies and the two different preferences for each one.

According to the theory everyone has a preference to one of each of the paired scales [7], and this leads to your type category. For example a personality of INFP is someone with preferences to *Introversion*, *Intuition*, *Feeling* and *Perceiving*. Additionally, everybody has a favourite process which is used primarily in their favourite world.

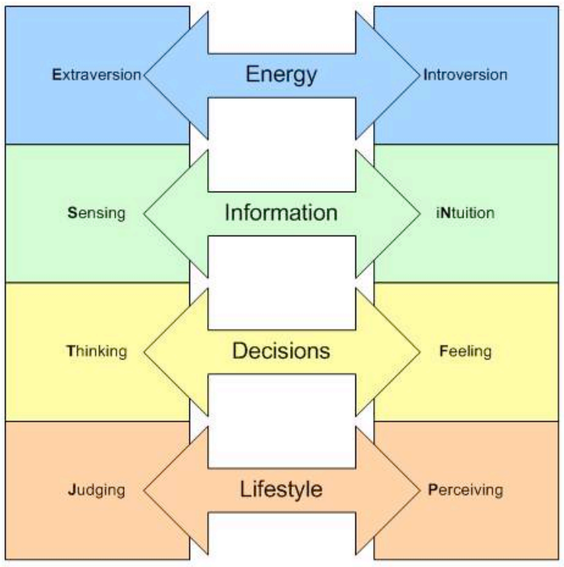


Figure 1. MBTI Dichotomies

Your favourite process is one of the two middle letters, and is most often used in your favourite *world*. To balance this, your second favourite process is used most often in the other *world*. For example, an INTJ would be described as favouring *introverted intuition* and *extroverted thinking*.

A lot of significance is put into your personality type as it suggests how you process and gather information, how you may act in situations, and your preferences in career choice [3]. It has been reported that the personality of a software engineer and the entire team is an important factor relating to project success and team cohesion [4].

There has been some widely publicised criticism of the MBTI assessment, stating that:

> *There was no support for the view that the MBTI measures truly dichotomous preferences or qualitatively distinct types, instead, the instrument measures four relatively independent dimensions.*
> *(R. McCrae and P. Costa, 1989)*

This claim and other claims regarding the MBTI assessment, such as a lack of independent evidence, and no evidence that *MBTI measures truly dichotomous preferences or qualitatively distinct types*, among other criticisms, were published in 1989 [22].

Additionally, reported MBTI results in software engineering have not been without their problems. S. McDonald and H. M Edwards (2007) reported identifying an article which had claimed to be reporting MBTI assessment results was not actually using a MBTI assessment [10], as later admitted by the initial authors in the technical report [21].

1.2 Literature Review

The literature review consisted of searching through the major online databases of journals and papers (including IEEE, ACM, PsycINFO) specifically searching for *MBTI, Myers-Briggs personality,* and *software engineering*.

The results displayed a vast amount of papers discussing personality and its usage [10] and potential effect [9, 11, 12] on a range of aspects of software engineering, such as educating engineers [13], and practitioner preferences [14]. However there is a significant lack of published complete MBTI data specific to software engineers, as many of the published works only print their conclusions and what the data informs them of, and not the actual MBTI preference breakdowns.

Some of the papers focused on alternative approaches to identifying and describing personality, such as the Five Factor Model [19, 15], as well as some other papers focusing on specific practices or roles in software engineering such as pair programming [15, 16, 18], software testing [17], and software team cohesion [19].

The literature review identified 12 papers reporting tables of Myers-Briggs Type Indicator data. The following section discusses these papers.

1.3 Collected MBTI Data

Of the 12 identified published data collections, there were 10 useable collections. One collection of data was removed as it was not possible to access enough details about how the data was collected and when it was collected. Of the remaining 10 collections of data there is 5 that collect their data primarily from practitioners, and 5 that collect their data primarily from students.

It was decided to focus on the papers that specifically collected their data from active working practitioners and to exclude data collected from predominantly student samples. This was to ensure that the data was representative of working software practitioners, and not people who were only potential software engineers. The paper will now continue to discuss these 5 data samples.

The 5 pieces of data are predominantly published in a span of 5 years, with the first 4 all ranging from 1985 to 1990 and the 6th data being published in 2003. All of them were collected from western companies, primarily from America. We'll now look at the information provided by each source separately in chronological order.

The first source is an article published in a computing magazine (Datamation) in 1985 by M.L. Lyons. The data consisted of 1,229 computer professionals, of whom 213 of them were based in the UK and Australia; the other 1,016 were based in America. The data consisted of 73% males and 27% females, and the median age for males was 34 and for females it was 31.

The surveyed members had a median of seven years experience in computing, with 30% having worked in computing for 5 years or less. Twenty percent of the surveyed members were in management positions, with an additional 20% working as project managers and team leaders. No clear information is given about the gathering of the data.

The second source was an article published in the Journal of Psychological Type in 1988 by E. A. Buie. In this study the MBTI results of 47 scientific computer professionals were presented, with 57.6% of them being male. The MBTI method used to gather the data from this source is described as being from a questionnaire "developed specifically for this study".

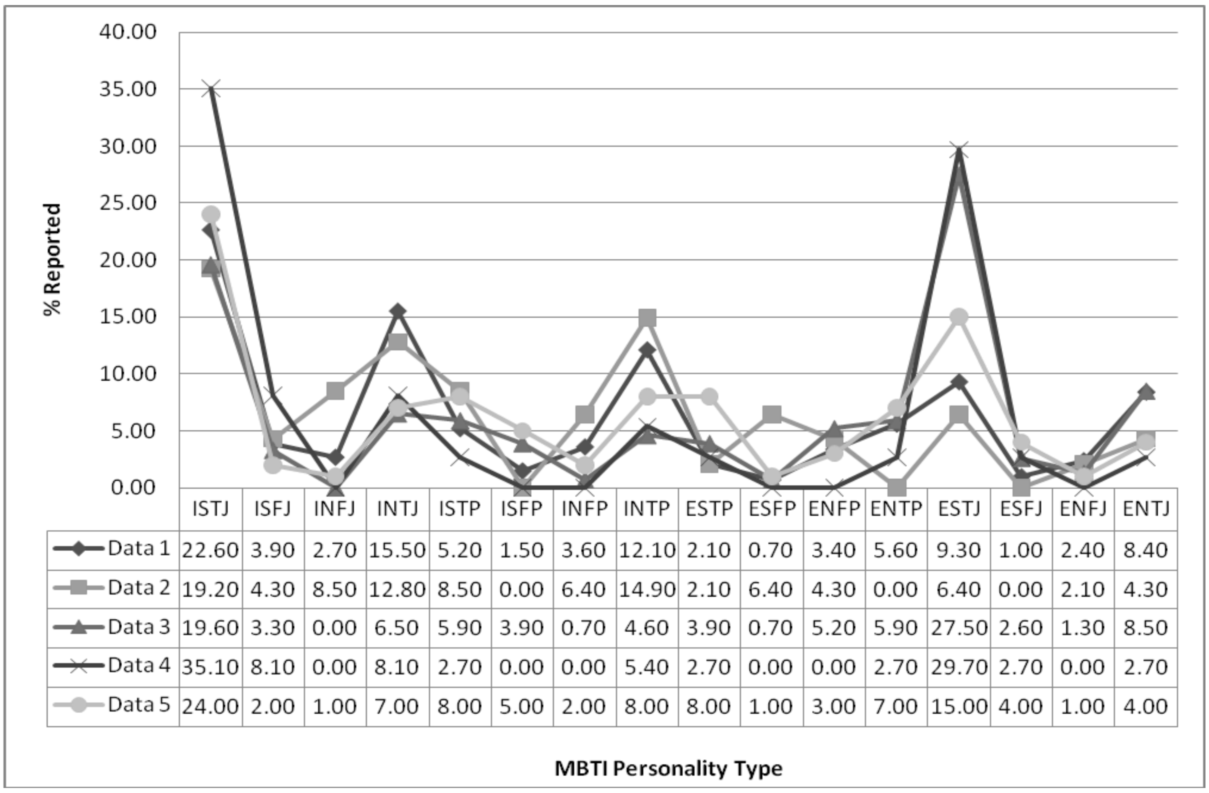
The third source was published in the Journal of Psychological Type in 1988 by P. Westbrook. The results presented were from a group of 153 professionals described as "Field Engineers". The results were said to be gathered from a "Fortune 400 computer company", and describes the method of gathering the information as a "self-scoring short form".

The fourth source is an article published in the ACM SIGCPR journal in 1989 by D. C. Smith. The data presented was gathered from 37 systems analysts working at a large insurance company. The method described for gathering the results is the "shorter version of the MBTI" and also states the questionnaire was administered by a psychologist.

The fifth source is an article published in the International Journal of Human-Computer Studies in 2003 by L. F. Capretz. This data collection contained 100 software engineers of which 80% were male and 20% female, and states that they were either working for the government, working for software companies, or were studying in public or private universities. The published paper also states that MBTI assessment was administered using Form G, which is an older and less reliable form of administering the MBTI assessment.

2. FINDINGS

This section discusses the 5 papers implementing a MBTI assessment on a group of software engineering practitioners. The data will be presented in this section, and discussed in subsequent sections.



	ISTJ	ISFJ	INFJ	INTJ	ISTP	ISFP	INFP	INTP	ESTP	ESFP	ENFP	ENTP	ESTJ	ESFJ	ENFJ	ENTJ
◆ Data 1	22.60	3.90	2.70	15.50	5.20	1.50	3.60	12.10	2.10	0.70	3.40	5.60	9.30	1.00	2.40	8.40
■ Data 2	19.20	4.30	8.50	12.80	8.50	0.00	6.40	14.90	2.10	6.40	4.30	0.00	6.40	0.00	2.10	4.30
▲ Data 3	19.60	3.30	0.00	6.50	5.90	3.90	0.70	4.60	3.90	0.70	5.20	5.90	27.50	2.60	1.30	8.50
✕ Data 4	35.10	8.10	0.00	8.10	2.70	0.00	0.00	5.40	2.70	0.00	0.00	2.70	29.70	2.70	0.00	2.70
● Data 5	24.00	2.00	1.00	7.00	8.00	5.00	2.00	8.00	8.00	1.00	3.00	7.00	15.00	4.00	1.00	4.00

Figure 2. Total MBTI Results

2.1 Compiled Results

The purpose of these studies was to identify the MBTI preferences of the participants, and these combined results from the 5 collections of data are expressed as percentages of the 16 different types.

The results gathered from the 5 sources of MBTI data are displayed in figure 2. It is apparent that widespread conformity can be noted on the majority of types, with the most variation in reported levels being represented by ISTJ, INTJ, INTP, and ESTJ.

ISTJ	ISFJ	INFJ	INTJ
6.47	2.29	3.56	3.97
ISTP	ISFP	INFP	INTP
2.33	2.28	2.56	4.41
ESTP	ESFP	ENFP	ENTP
2.48	2.62	1.97	2.85
ESTJ	ESFJ	ENFJ	ENTJ
10.55	1.57	0.95	2.69

Table 2. MBTI Standard Deviation

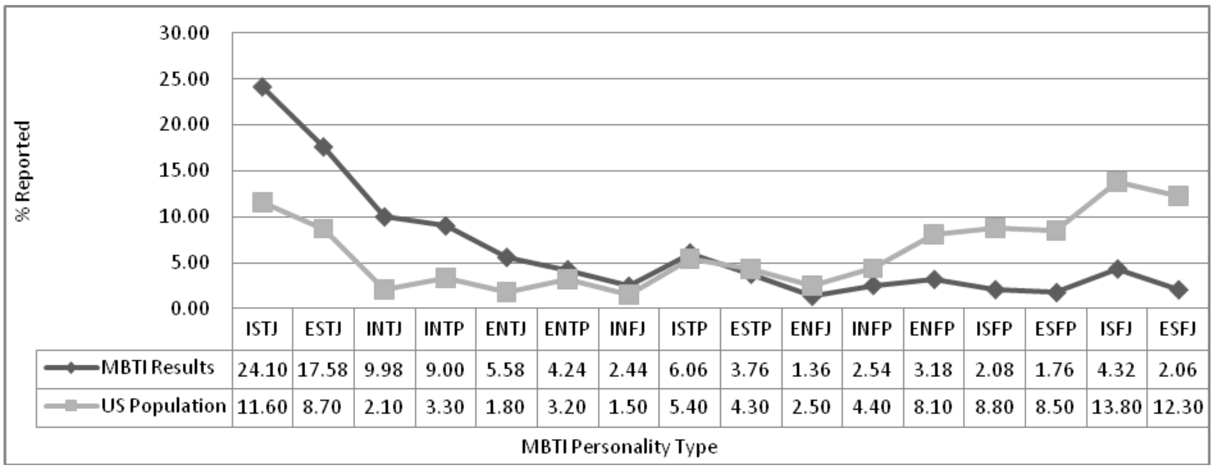
Table 2 further supports the level of agreement in commonly reported results by presenting the standard deviation value for each of the 16 MBTI types when all of the different data collection results are combined.

Figure 3 compared the averages of all of the MBTI types based on the results gathered from the 5 sources to the average preferences of the US population as reported by CAPT (Centre for Applications of Psychological Type) [2].

The comparison makes it clear that the common preferences of the US population are not reflected inside software engineering. In different categories the preferences are over or under represented, but there is an emerging pattern that the *thinking* preference is consistently over-represented in the reported MBTI results.

ISTJ, INTJ, ISTP, INTP, ENTP, ESTJ, and ENTJ all display an average higher than the general US population, with the only exception being the marginal difference on ESTP from 4.30 to 3.76.

Table 3 shows the representation of each pair in the results, as well as the level of standard deviation and the mean. These figures were generated from the data represented in Figure 2. There is a higher level of variation when the data is constructed like this and thus the means are less reliable.



	ISTJ	ESTJ	INTJ	INTP	ENTJ	ENTP	INFJ	ISTP	ESTP	ENFJ	INFP	ENFP	ISFP	ESFP	ISFJ	ESFJ
MBTI Results	24.10	17.58	9.98	9.00	5.58	4.24	2.44	6.06	3.76	1.36	2.54	3.18	2.08	1.76	4.32	2.06
US Population	11.60	8.70	2.10	3.30	1.80	3.20	1.50	5.40	4.30	2.50	4.40	8.10	8.80	8.50	13.80	12.30

Figure 3. MBTI Results and US Population Averages Comparison

As you can see from Table 3, five of the highlighted pairings are pairings including thinking, and four of the pairings including judging. Sensing is also present four times, and extroversion and introversion appear three times each. However, feeling does not appear once, neither does perceiving, showing a clear lack of a common representation to either of these preferences.

	Data 1	Data 2	Data 3	Data 4	Data 5	Standard Deviation	Mean
TJ	55.8	42.7	62.1	75.6	50	12.52	57.24
ST	39.2	36.2	56.9	70.2	55	13.93	51.5
IT	55.4	55.4	36.6	51.3	47	7.82	49.14
SJ	36.8	29.9	53	75.6	45	17.67	48.06
IJ	44.7	44.8	29.4	51.3	34	8.91	40.84
IS	33.2	32	32.7	45.9	39	5.92	36.56
ET	25.4	12.8	45.8	37.8	34	12.62	31.16
NT	41.6	32	25.5	18.9	26	8.53	28.8
EJ	21.1	12.8	39.9	35.1	24	10.92	26.58
ES	13.1	14.9	34.7	35.1	28	10.59	25.16
IN	33.9	42.6	11.8	13.5	18	13.59	23.96
TP	25	25.5	20.3	13.5	31	6.55	23.06
IP	22.4	29.8	15.1	8.1	23	8.31	19.68
NJ	29	27.7	16.3	10.8	13	8.45	19.36
NP	24.7	25.6	16.4	8.1	20	7.12	18.96
EN	19.8	10.7	20.9	5.4	15	6.45	14.36
SP	9.5	17	14.4	5.4	22	6.46	13.66
EP	11.8	12.8	15.7	5.4	19	5.06	12.94
IF	11.7	19.2	7.9	8.1	10	4.64	11.38
SF	7.1	10.7	10.5	10.8	12	1.84	10.22
FJ	10	14.9	7.2	10.8	8	3.01	10.18
FP	9.2	17.1	10.5	0	11	6.15	9.56
NF	12.1	21.3	7.2	0	7	7.87	9.52
EF	7.5	12.8	9.8	2.7	9	3.71	8.36

Table 3. MBTI Paired Preferences

The level of representation of *thinkers* is very apparent here with multiple pairings including *thinking* surpassing an average representation of 50%.

3. RESULTS SUMMARY

For the most part the MBTI results show agreement with each other. There are only two different types with a standard deviation above 5.00 (ESTJ with 6.47 and ISTJ with 10.55).

Thinkers are present significantly more within the results than *feelers.* This can be seen within the results where the types of *feelers* are commonly scoring lower than the types of *thinkers*.

4. DISCUSSION

4.1 Results Discussion

The results presented here, although from a range of sources and publications, present a common view on the MBTI personalities present inside software engineering. Although four of the five collections of data were during 1985-1990, it is interesting that the one collection of data from 2003 does not show much if any of a change in the reported types.

The data suggests a relationship between psychological type and software engineers, but this does not imply that there is a singular fix for a software engineer nor that one type is more *useful* than any other, it simply suggests that there are some types more prominently found inside software engineering.

Specifically *thinkers* have been established by the data as being commonly represented as a preference by the majority of the reported results, with preferential *thinkers* representing an average (mean) of 80.3% of the reported results, compared to 19.7% being *feelers*.

There is also a 60/40 split in favour of *introverted* preference than the *extroverted* preference, which is different to the near 50/50 split suggested to be present among the US population. There is also a 67% representation of the *judgement* preference being present over the *perceiving* preference.

It is also worth noting that the pairing of *thinking* and *judging* is substantially higher than the US population (24.20%) with an average of 57.24%, while the pairing of *feeling* and *perceiving* is much lower than the US population (29.80%) with an average reported result of 9.56%.

All of this information leads to suggesting that *thinkers* and *judgers* are more attracted to software engineering and *feelers* and *perceivers* are less attracted to software engineering jobs, based on the data combined from 5 sources of MBTI assessment.

4.1 Limitations

The information presented here has inherent limitations on how the original collections of data have been published. There is not enough information published in most of the papers detailing how the assessments were administered, if a psychologist was present to administer the MBTI assessments, or what form was used. This makes it impossible to ensure the data is comparable across studies.

The details of the respondents also vary, with some of the papers stating the exact breakdown of age, gender and experience, with others offering either incomplete breakdowns or no information at all. The physical number of respondents also ranges from over a thousand to under fifty, as well as the type of people the results report about.

It would be acceptable to classify the 2nd through to the 5th source as valid as software engineering personnel, but the first source is simply too generic to be classified as specifically about software engineers. This means the type of data being compared cannot be described as being exclusively focused on identical groups of software engineers in a range of studies.

5. CONCLUSIONS

The results gathered here represent a common picture on the majority of the MBTI preferences present inside software engineering, and show a large preference towards *thinkers* and *judgers*.

On average there are 57.24% of the respondents with a *thinking judging* preference, 51.50% with a *sensing thinking* preference, and 49.14% showing an *introverted thinking* preference, further establishing the strong preference towards *thinking* in the results presented here.

The combined results of these 5 MBTI data collections compare well with other publications where full MBTI data was not available but the conclusions were presented. Bush and Schkade (1985) [23] also identified *thinkers* as being a significantly represented preference in their results of 40 programmer analysts with 73% of the respondents reported as being *thinkers*. They also identified that 70% of their respondents preferred *judging* as opposed to *perceiving*.

Thomsett (1990) [24] identified a high representation of *thinkers* (79%) in a group of 656 computer professionals. It was identified that there was a far higher representation of *judgers* than there was *perceivers*, with 92.3% of the respondents showing a preference towards *judging*.

However it is clear that there has been very little work done on the types of personalities in software engineering practitioners recently using the MBTI assessment tool, with only one of the results being published in the past 20 years (2003). This is important, as it's very likely at the types of roles, and software engineering has changed in those 20 years, along with the years since that study, with the growth of agile development practices.

If we are to accept and believe what we are told about the implications of personality and MBTI, it is extremely important that further work is done to understand the types inside software engineering. A better understanding could lead to a better workplace and task fit to specific people, specifically a greater understanding into the effects of personality on teams. It is not unreasonable to consider the possibility that the dynamics of a team and the personalities present could drastically affect performance and productivity.

The results presented here are old and generic, one collection of data is even too generic to really be considered about software engineers. Future work on personality, and specifically MBTI, should focus on establishing up-to-date classifications of what *software engineering* really means, and specifically the different roles identifiable inside this area. It's quite possible that the role of somebody considered a software engineer could also affect the type of personality drawn to this role, and potentially explain the variations reported in this paper. We do not yet have any data to suggest that the skills and personality required to be a software designer are equal to that of the skills required to be a software programmer, and Capretz (2010) [25] recently reports on the potential varying personality types required under the umbrella of software engineering.

6. REFERENCES

1. Datamonitor (2006) *Software: Global Industry Guide*. Available from: http://www.infoedge.com/product_type.asp?product=DO-4959, accessed 01/02/2010.
2. CAPT (Centre for Applications of Psychological Type) (2010). *Estimated Frequencies of the Types in the United States Population*. Available from: http://www.capt.org/mbti-assessment/estimated-frequencies.htm, accessed 25/01/2010.
3. Myers, I.B. 1998. Introduction to type : a guide to understanding your results on the Myers-Briggs Type Indicator.5th Edition. Oxford : Oxford Psychologists Press, 1998.
4. J. Karn and T. Cowling, "A follow up study of the effect of personality on the performance of software engineering teams," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, 241.
5. Sidney J. Blatt, "Where Have We Been and Where Are We Going? Reflections on 50 Years of Personality Assessment," *Journal of Personality Assessment* 50, no. 3 (1986): 343.
6. C.G. Jung, *Psychological Types*, 1st ed. (Routledge, 1992).
7. Isabel Briggs Myers, An Introduction to Type: A Guide to Understanding Your Results on the Myers-Briggs Type Indicator: European English Version, 5th ed. (Oxford Psychologists Press, 198).
8. Catherine Bishop-Clark and Daniel D. Wheeler, "The Myers-Briggs personality type and its relationship to computer programming.," *Journal of Research on Computing in Education* 26, no. 3 (Spring94 1994): 358.
9. Alessandra Devito Da Cunha and David Greathead, "DOES PERSONALITY MATTER? AN ANALYSIS OF CODE-REVIEW ABILITY.," *Communications of the ACM* 50, no. 5 (May 2007): 109-112.
10. S. McDonald and H. M Edwards, "Who should test whom?," *Communications of the ACM* 50, no. 1 (2007): 71.
11. Robert Feldt et al., "Links between the personalities, views and attitudes of software engineers," *Information and Software Technology* In Press, Accepted Manuscript.

12. J. S. Karn et al., “A study into the effects of personality type and methodology on cohesion in software engineering teams.,” *Behaviour & Information Technology* 26, no. 2 (March 2007): 99-111.
13. Lucas Layman, Travis Cornwell, and Laurie Williams, “Personality types, learning styles, and an agile approach to software engineering education,” *ACM SIGCSE Bulletin* 38, no. 1 (3, 2006): 428.
14. L Capretz, “Personality types in software engineering,” *International Journal of Human-Computer Studies* 58, no. 2 (2, 2003): 207-214.
15. N. Salleh et al., “An empirical study of the effects of personality in pair programming using the five-factor model,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, 214–225.
16. J. E Hannay et al., “Effects of Personality on Pair Programming.”
17. Shoaib, L.; Nadeem, A.; Akbar, A.; , "An empirical evaluation of the influence of human personality on exploratory software testing," *Multitopic Conference, 2009. INMIC 2009. IEEE 13th International* , vol., no., pp.1-6, 14-15 Dec. 2009
18. T. Walle and J. E Hannay, “Personality and the nature of collaboration in pair programming,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, 203–213.
19. Rebecca H. Rutherfoord, “Using personality inventories to help form teams for software engineering class projects,” *ACM SIGCSE Bulletin* 33, no. 3 (9, 2001): 73-76.
20. R. R McCrae and O. P John, “An introduction to the five-factor model and its applications,” *Personality: critical concepts in psychology* 60 (1998): 295.
21. Karn, J.S. and Cowling, A.J. A study into the effect of disruptions on the performance of software engineering teams. Research Memorandum CS-04-07, Department of Computer Science, Sheffield University, 2004.
22. Robert R. McCrae and Paul T. Costa Jr., “Reinterpreting the Myers-Briggs Type Indicator From the Perspective of the Five-Factor Model of Personality.,” *Journal of Personality* 57, no. 1 (March 1989): 17-40.
23. Chandler M Bush and Lawrence L Schkade, “In search of the perfect programmer,” *Datamation* 31, no. 6 (1985): 128-132.
24. R. Thomsett, "Building Effective Project Teams", American Programmer, Summer 1990
25. Luiz Fernando Capretz and Faheem Ahmed, “Making Sense of Software Development and Personality Types,” *IT Professional*, 2010.

7. MBTI DATA SOURCES

1. Michael L. Lyons, “THE DP PSYCHE.,” *Datamation* 31, no. 16 (1985): 103-105, 108, 110.
2. Buie, E. A. (1988). Psychological type and job satisfaction in scientific computer professionals. Journal of Psychological Type, 15, 50-53.
3. Westbrook, P. (1988) Frequencies of MBTI Types Among Computer Technicians. Journal of Psychological Type, 15, 49.
4. D. C. Smith, “The personality of the systems analyst: an investigation,” *SIGCPR Comput. Pers.* 12, no. 2 (1989): 12-14.
5. L Capretz, “Personality types in software engineering,” *International Journal of Human-Computer Studies* 58, no. 2 (2, 2003): 207-214.
6. R. Thomsett, "Building Effective Project Teams", American Programmer, Summer 1990.

7. Jo Ann C. Carland and James W. Carland, “Cognitive styles and the education of computer information systems students.,” *Journal of Research on Computing in Education* 23, no. 1 (Fall90 1990): 114.

8. Eugene Kaluzniacky, *Managing psychological factors in information systems work* (Idea Group Inc (IGI), 2004).

9. L. F Capretz, “Clues on Software Engineers' Learning Styles,” *International Journal of Computing & Information Sciences* 4, no. 1 (2006): 46-49.

10. J. S. Karn et al., “A study into the effects of personality type and methodology on cohesion in software engineering teams,” *Behaviour & Information Technology* 26, no. 2 (3, 2007): 99-111.

11. S. Holmes and P. L Kerr, (2007). The IT crowd. *Australian Pyschological Type Review* 9, no. 1 (4, 2007): 31-38.

Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size, Experience and Reasoning Skills

Gul Calikli
Department of Computer Engineering, Software Research Laboratory, Bogazici University, Turkey
gul.calikli@boun.edu.tr

Berna Arslan
Department of Computer Engineering, Software Research Laboratory, Bogazici University, Turkey
berna.arslan@boun.edu.tr

Ayse Bener
Department of Computer Engineering, Software Research Laboratory, Bogazici University, Turkey
bener@boun.edu.tr

Keywords: POP-II.B. problem comprehension, testing; POP-V.A. confirmation bias, cognitive bias

Abstract

During all levels of software testing, the goal should be to fail the code to discover software defects and hence to increase software quality. However, software developers and testers are more likely to choose positive tests rather than negative ones. This is due to the phenomenon called confirmation bias which is defined as the tendency to verify one's own hypotheses rather than trying to refute them. In this work, we aimed at identifying the factors that may affect confirmation bias levels of software developers and testers. We have investigated the effects of company size, experience and reasoning skills on bias levels. We prepared pen-and-paper and interactive tests based on two tasks from cognitive psychology literature. During pen-and-paper test, subjects had to test given hypotheses, whereas interactive test required both hypotheses generation and testing. These tests were conducted on employees of one large scale telecommunications company, three small and medium scale software companies and graduate computer engineering students resulting in a total of eighty-eight subjects. Results showed regardless of experience and company size, abilities such as logical reasoning and strategic hypotheses testing are differentiating factors in low confirmation bias levels. Therefore, education and/or training programs that emphasize mathematical reasoning techniques are useful towards production of high quality software. Moreover, in order to investigate the relationship between code defect density and confirmation bias of software developers, we performed an analysis among developers who are involved with a software project in a large scale telecommunications company. We also analyzed the effect of confirmation bias during software testing phase. Our results showed that there is a direct correlation between confirmation bias and defect proneness of the code.

1. Introduction

Human aspects are one of the basic components of software development and testing. Cognitive biases belong to these aspects and they are defined as the deviation of the human mind from the laws of logic and accuracy (Stacy & MacMillan, 1993). The notion of cognitive biases was first introduced by Tversky and Kahneman (1971) and further elaborated to comprise various bias categories (Kahneman, Slovic, & Tversky, 1982**).** Availability, anchoring and representativeness are examples of cognitive biases.

As far as we know, Stacy and MacMillan (1993) are the two pioneers who recognized the possible effects of cognitive biases on software engineering. In this study, we have investigated the confirmation bias, which is defined as the tendency of people to seek for evidence that could verify their hypotheses rather than refuting them. The term confirmation bias was first used by P.C. Wason in his rule discovery experiment (Wason, 1960).

The importance of confirmation bias in software engineering comes from the fact that, most of the defects are overlooked unless the goal is to fail the code during all levels of software testing. In other words, high confirmation bias levels of software developers and testers lead to an increase in software defect density which in turn results in a decrease in software quality. Empirical evidence shows that testers are more likely to choose positive tests rather than the negative ones (Teasley, Leventhal, & Rohlman, 1993). However, during all levels of software testing the attempt should be to fail the code. This study is concentrated on factors affecting confirmation bias, since circumvention of the effects of confirmation bias requires that we know about these factors. In this study, we also perform a small scale empirical analysis about the effects of confirmation bias on software development and testing.

We propose a method based on Wason's work to quantify confirmation bias levels. Quantification of confirmation bias helps us to analyze the effect of confirmation bias on software defect density as well as analyzing potential factors that may affect confirmation bias. In the following section, we will give more detailed information about confirmation bias. Then we will explain our methodology in detail. Finally, results will be presented, discussed together with threats to validity and potential future directions will be given.

2. Confirmation Bias

The term confirmation bias was first used by Peter C. Wason in his rule discovery experiment, where the subject must try to refute his/her hypotheses to arrive at a correct solution (Wason, 1960). Wason also explained the results of his selection task experiment using facts based on confirmation bias (Wason, 1968). This section explains these two experiments.

2.1. Wason's Rule Discovery Task

In this experiment, Wason asked his subjects to discover a simple rule about triples of numbers. The experimental procedure can be explained as follows: Initially, subjects are given a record sheet on which the triple "2 4 6" is written. The subject is told that "2 4 6" conforms to a simple rule which is only known by the experimenter at the beginning of the experiment. In order to discover the rule, the experimenter asks the subject to write down triples together with the reasons of his/her choice on the record sheet. After each instance, the experimenter tells whether the instance conforms to the rule or not. The subject can announce the rule only when he/she is highly confident. If the subject cannot discover the rule, he/she can continue giving instances together with reasons for his/her choice. This procedure continues iteratively until either the subject discovers the rule or he/she wishes to give up. If the subject cannot discover the rule in 45 minutes, the experimenter aborts the test.

2.2. Wason's Selection Task

In Wason's original task, the subject is presented with four cards, which have a letter on one side and a number on the other. These cards are placed on a table showing D, K, 3 and 7 respectively. The subject is then given the hypothesis "If a *card has a D on one side, then it has a 3 on the other side*" and he/she is asked which card(s) he/she should turn over to test whether the given hypothesis is true or false regarding the four cards presented to him/her. The hypothesis can be considered as a "*If P, then Q*" sentence. The correct way to test this hypothesis would be to select the *P* and *not Q* cards, which corresponds to selecting *D* and *7* respectively.

3. Methodology

Our methodology aims to quantify confirmation bias levels in order to make empirical analyses to investigate the factors affecting confirmation bias.

3.1. Preparation of the Tests

Pen-and-Paper Test. We prepared the pen-and-paper test based on Wason's Selection Task (Wason, 1968). Our test consisted of two parts including twenty-two questions of three different types. There were eight abstract, six thematic questions in the first part, whereas the second part contained eight questions about software domain.

Abstract questions where subject is asked to check the validity of hypothesis of the form "*If P then Q*", can be answered correctly by pure logical reasoning. However, a subject might select the cards *D* and *3*, when he/she is asked to check the validity of the hypothesis "*If a card has a D on one side, then it has a 3 on the other side.*" Regarding the four cards presented to him/her. Such behavior of the subject can be explained by one of the following reasons:

1. *Verifying:* Subject's aim might be to *verify* the given hypothesis.
2. *Matching:* Subject might select the cards just by *matching* the letter *D* and number *3*. In other words, any reasoning strategy is not employed by the subject. This phenomenon, which was first defined by Evans (1972), is called *matching bias.*

One of the abstract questions was the question proposed by Wason in his selection task. Table 1 shows all versions of the original question with their possible answers categorized as true/false antecedent and true/false consequent. In order to analyze response strategies of subjects to these questions, all three negated versions of this original question were included in the test. Three of the remaining abstract questions had slight variations from the original Wason's Selection Task question.

Rule	TA	FA	TC	FC
(If P, then Q) If there is a D on one side, then there is a 3 on the other side	D	K	3	7
(If P, then not Q) If there is a D on one side, then there is not a 3 on the other side	D	K	7	3
(If not P, then Q) If there is not a D on one side, then there is a 3 on the other side	K	D	3	7
(If not P, then not Q) If there is not a D on one side, then there is not a 3 on the other side	K	D	7	3

Table 1 – The four logical choices in Wason's selection task with negated components permuted, TA: True Antecedent, FA: False Antecedent, TC: True Consequent, FC: False Consequent

Each thematic question in the test represented a probable real-life situation, so that they could be answered correctly using the cues produced by memory. Finally, domain-specific questions required participants to analyze a software problem which is independent of programming tools and environment.

Interactive Test. This test was a replication of Wason's Rule Discovery Experiment. Hence, similar to the original experiment, the experimenter performs the test face-to-face with each subject one at a time. Prior to the test, the experimenter explains to each subject the experimental procedure and the subject is also told that there is no time constraint.

3.2. Evaluating Pen-and-Paper Test

Falsifier/Verifier/Matcher Classification. Given the conditional rule of the form *if P, then Q*, the subject who selects *P, Q* as the answer can either be a verifier or matcher. Similarly, the same answer for the rule *if P, then not-Q*, means that the subject can be a falsifier or a matcher. In order, to overcome this fuzziness, we employed the method of Reich and Ruth (1982), which is explained below as follows:

- choice of not-Q in the rule "If P, then Q" = falsifying
- choice of not-Q in the rule "If P, then not Q" = verifying
- choice of P in the rule "If not P, then Q" = matching
- choice of not-Q in the rule "If not-P, then Q" = falsifying
- choice of P in the rule "If not P, then not Q" = matching

- choice of not-Q in the rule "If not P, then not Q" = verifying

According to the responses given to the four types of an abstract question, the subject was categorized as a falsifier, verifier or a matcher. This categorization decision was given according to the majority of the response tendencies, i.e. if a subject usually responded in a falsifying strategy, then he/she was categorized as a falsifier. There was also the possibility that the subject could not be categorized. This categorization neglects a large proportion of data provided by the subjects. On the other hand, it gives a general view about the subjects' responses. For these reasons, we used this method and labelled subjects, whom we could not classify, as *None*.

Insights. As an additional measure of the test, we took insights of the subjects into account. Johnson-Laird and Wason (as cited in Evans, Newstead & Bryne, 1993) proposed that subjects were in one of three states of insight as follows:

No insight. Subjects attempted to verify and chose *p* alone or the combination of *p* and *q* in the original question.

Partial insight. Subjects attempted to both verify and falsify the rule and hence chose *p*, *q* and *not-q* cards in the original question.

Complete insight. Subjects only attempted to falsify the rule and chose *p* and *not-q* cards in the original question.

Since we had more than one question in the test, we devised a method to classify the subjects according to this concept. Subjects who answered all of the abstract questions correctly were classified to have *complete insight*. Those who chose true antecedent alone or the combination of true antecedent and true consequent in more than or equal to half of the abstract questions were classified as having *no insight*. Finally, subjects who chose the combination of true antecedent, true consequent and false consequent in more than or equal to the half of the questions were classified as having *partial insight*.

3.3 Evaluating Interactive Test

Eliminative/Enumerative Index. This index aims to give an idea about how a subject thinks while he/she is going forward in the interactive test. A subject may think more eliminative and test more hypotheses or may think more enumerative and test similar hypotheses with different instances. The calculation of this index takes into account the nature of the instances in relation to their reasons for choice. The index is calculated as a ratio of the number of subsequent instances incompatible with each reason of choice to the number of compatible instances, summed over all proposed reasons. It is desirable to have this index to be greater than 1. Wason indicates that when this value is greater than 1 (the higher the better), the less confirmation bias of the subject is.

Test Severity. Severe testing consists of testing observations that have a high probability of being true in focal hypothesis and a low probability under all possible hypotheses (Poletiek, 2001). The severity of a test can be thought of as the power to eliminate alternative hypotheses. Poletiek's rule discovery experiment (2001) presented the subjects with the triple "2 7 6" and asked them to discover the rule that this triple conformed to. Test severity was calculated for each subject as follows:

$$S(x,H,b) = \frac{P(x \mid H,b)}{P(x \mid b)} \qquad (1)$$

This formula was defined by Popper (Poletiek, 2001), and *x* represents the severity of a test, *H* represents the hypothesis and *b* stands for the background knowledge. The severity of a test *x* is interpreted as the supporting evidence of the theory *H* given the background knowledge *b*. A test is more severe when the chance of the supporting observation occurring under the assumption of the hypothesis *H* exceeds the chance of its occurring without the assumption of the *H* (i.e. with the assumption of the background knowledge *b* only). The higher this ratio is (exceeds 1), the higher the severity of the test is.

To carry out a similar calculation, we have defined a set of possible alternative hypotheses of the interactive test as shown in Table 2. Instances given by the subjects are categorized as positive or negative tests according to their compliance with the experimenter's rule. Then, a positive test is more severe if it *excludes* more alternative hypotheses shown in Table 2 and a negative test is more severe if it *includes* more alternative hypotheses. For each instance given by the subject, a severity score was calculated. Finally, a mean severity score for each subject was calculated over all of his/her instances. This mean score could be between 0 and 27, since there were twenty-seven alternatives defined.

Poletiek (2001) discussed that maximizing the severity of tests may be a successful strategy if the subject has enough confidence in his/her hypothesis. Starting with a severe test may not result in more knowledge, so that it may be a better strategy to start with a mild test, increasing its severity when there is more evidence and then decreasing its severity again when one becomes quite sure about the validity of the hypothesis.

1	Integers ascending with increments of 2	15	Sum of the first and second integer is the third integer
2	Integers ascending with increments of k, where k = 1,2, ...	16	The triples of the form (2n 4n 6n), where n = 1,2,3, …
3	Three integers in ascending order such that the average of the first and third integer is the second integer.	17	The triples of the form (n 2n 3n), where n = 1,2,3, …
4	The average of the first and third integer is the second integer	18	Second integer is greater than the first one
5	Even integers ascending with increments of 2	19	Third integer is greater than the first integer
6	Integers ascending with increments of m = 2k, where k = 1,2,3, …	20	Difference between the third and the first integer is even
7	Integers ascending or descending with increments of m = 2k, where k = 1,2,3, …	21	Greatest common divisor (GCD) of the integers is 2
8	Even integers in ascending order	22	Ascending integers such that each integer is 1 less than a prime number
9	Positive even integers in ascending order	23	Any three rational numbers
10	Three even integers in any order	24	Positive real numbers in increasing order
11	Three integers in any order, none of them are identical	25	Positive integers in increasing order
12	Three integers in any order, two or three of them are identical	26	Three integers whose sum is even
13	Three integers in ascending order such that difference between third and first number is even	27	Three even integers greater than zero
14	Integers ascending or descending with increments of k, where k = 1, 2, 3, …		

Table 2 – The Set of Plausible Alternative Hypotheses

4. Experiment

4.1. Participants

Participants were employees of four companies and graduate computer engineering students. One of the companies was a large scale telecommunications company C_1 (11 females, 23 males, mean age = 29.06 years). There were three other small and medium scale software companies C_2 (6 males, mean age = 24 years), C_3 (1 female, 7 males, mean age = 29.63 years), and C_4 (1 female, 11 males, mean age = 27.17 years). In addition, twenty-eight graduate computer engineering students participated in the study (7 females, 21 males, mean age = 27.96 years).

Most of our subjects had a bachelor degree in computer science, mathematics or other engineering fields.

4.2. Procedure

All participants from companies were tested in their work environment. Students were tested at the university. First, it was explained that the tests do not aim at measuring IQ or similar capabilities. It has been told that the goal of the tests was identifying how people think. Subjects were asked to fill in the form about their personal information. Information about age, gender, B.S. field and university, M.S./M.A. field and university (if they exist), software development experience (in years), software testing experience (in years) was taken from each participant.

In each company, pen-and-paper test was applied to the employees as a group after the explanation of the test. Later, subjects participated one by one in the interactive test. Durations of both tests were recorded.

5. Results

In the following subsections, results of the analyses of the effects of company size, experience and reasoning skills are presented.

5.1. Effects of Company Size, Experience and Reasoning Skills on Confirmation Bias

The Analysis of Company Size. In this part, results of the large scale telecommunication company are compared with the results of three small/medium scale software companies in terms of both test performances.

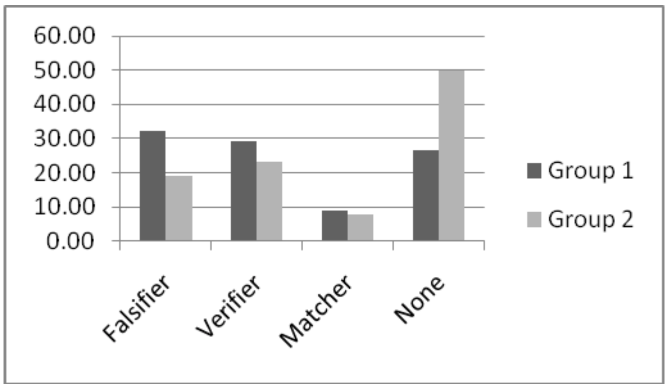


Figure 1 – The Distribution of Falsifiers, Verifiers and Matchers within Group 1(large scale company) and Group 2(small and medium scale companies)

In Figure 1, the percentages of falsifiers, verifiers and matchers for both groups can be seen. The large scale company, which is denoted as Group 1 in Figure 1, has a higher percentage of falsifiers, verifiers and matchers. Since this distribution alone is not very explanatory due to the high percentage of subjects not categorized, we examined another aspect, namely 'insights'. The distribution of insights is shown in the Figure 2. According to Figure 2, the large company had a higher percentage of subjects with complete insight, slightly smaller percentage of subjects with partial insight and smaller percentage of subjects with no insight. But this difference is not significant statistically. Also, no significant difference was observed in the scores of the abstract questions indicating that no difference in logical reasoning skills was observed among the two groups.

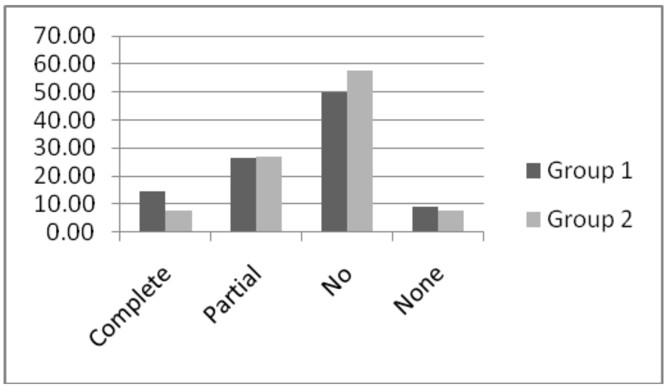


Figure 2 – The Distribution of Subjects according to their Insights within Group 1(large scale company) and Group 2(small and medium scale companies)

However, analyzing the interactive tests yielded differences between two groups. After excluding outliers from both groups, Mann-Whitney U-test was used to compare the ranks for eliminative/enumerative index values. The results of the test were significant, $z = -2.76$, $p < .01$. Group 1 ($n=34$) had an average rank of 35.47, while Group 2 ($n=26$) had an average rank of 23.06. This indicated that employees of the large scale company performed better in terms of elimination of their hypotheses.

Another Mann-Whitney U-test was used to compare the ranks for test severity values among Group 1 and Group 2. The results of the test were significant, $z = -2.26$, $p < .05$. Group 1 had an average rank of 34.96, while Group 2 had an average rank of 24.67. This indicated that, Group 1 had higher test severity values indicating a strategy that employed high severe testing. But, as mentioned before, it is a successful strategy to start with a mild test, continue with a more severe test and then end with a mild test again. In order to compare these severity strategies among two groups, we have made use of Vincent curves (as cited in Hilgard, 1938). These curves can be used to visualize the change in test severity of a group of subjects until the discovery of the correct rule. We have used the original method proposed as follows:

A number *N* was taken to be the bin size and total number of instances given by each subject was divided into fractions according to this number. Within each fraction, the average of the test severities of instances in that fraction was calculated for each subject. Then, all severity values in each fraction were averaged to give the mean severity value of all instances given by subjects in that fraction. As N, we have used the smallest number of instances given within the group of subjects. For instance, the division of 13 instances into N=4 fractions would be 4, 3, 3, 3. Additional instances that were left over from the division were added to the beginning according to the original procedure. Then, in the first fraction the average severity value of the four instances that fell in that fraction was calculated. This procedure was repeated for all subjects until mean severity values for N bins were obtained. A Vincent curve depicts these N data points, and it can be used to interpret the severity strategy employed by the subject.

Figure 3 shows the Vincent curve for the test severities of the three groups of subjects. The bin size was taken to be three, equal to the minimum number of instances given by a subject.

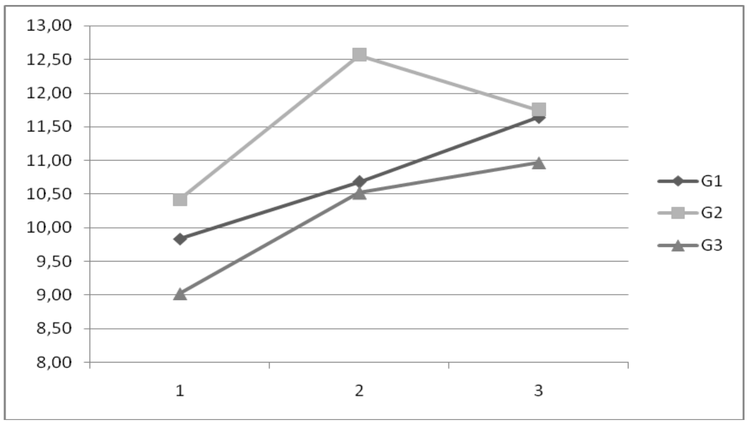


Figure 3 – Vincent Curve showing Test Severity Strategy of three groups, G1 being the group of small and medium scale companies, G2 being the large scale company and G3 being the group of graduate students

Examining the curve for the large scale company, denoted as G2 in the figure, shows that testing started with a mild test and became more severe and then ended in a milder test. This strategy was mentioned to be a successful strategy for hypothesis testing. The curve denoted with G1 in the figure shows that the overall strategy of the small and medium scale companies was to begin with a mild test and gradually enhance the severity. By comparing these two curves, we can say that the strategy of the large scale company shows a more sensible hypothesis testing strategy.

However, Poletiek (2001) also mentioned that selecting severe rather than weak tests does not necessarily reflect a motivation to falsify. In fact, the purpose of such a strategy might be to give a strong proof for one's theory. To investigate that, we have analyzed the percentages of subjects in both groups, who have showed strong confirmation tendencies by repeating or reformulating their reasons for choice or their rules and who have immediately announced new rules without giving an instance, i.e. without testing their hypotheses. We have determined the percentages of these subjects within all groups as shown in Table 3. A subject who may have repeated a reason may also have made an immediate rule announcement, so that it is not the case that a subject can only be found in one cell of Table 3.

In order to compare these statistics between groups, we have merged the columns of Table 3 and compared the number of subjects who behaved in a confirmative way between groups, i.e. subjects who engaged in at least one activity defined in the columns. We have found no significant difference between the groups of companies according to company size, ($\chi^2(1) = .09$, $p > .7$). Hence, we can say that large scale company engaged in a better hypothesis testing strategy by looking at eliminative/enumerative index, test severity values and Vincent curves.

The Analysis of Experience. We have investigated whether there was a significant correlation between software testing experience and test severity. All subjects who had software testing experience were included in the test (n=27), and no significant result was obtained. Also, no significant correlation was found between software testing experience and eliminative/enumerative index.

A similar analysis was conducted for software development experience. All subjects who had software development experience were included in the test (n=50). No significant correlation was found between software development experience and eliminative/enumerative index as well as test severity.

Hence, results showed no effect of software testing or development experience on the ability of hypothesis testing.

Another analysis was conducted to compare test results among subjects taken from graduate students and employees of all companies, where the subject selection criterion was having the combination of software development and testing experience greater than the average number of years of experience found among graduate students. The average experience value was taken from the students' group to make sure that there will be enough subjects from this group in the analysis.

Mann-Whitney U-test was used to compare the ranks for test severity, eliminative/enumerative index and test durations among employees (n=43) and students (n=13). The only significant result of the tests showed that interactive test duration of the student group was lower, $z = -1.39$, $p < .05$. First group had an average rank of 30.15, while second group had an average rank of 23.04. Hence, we concluded that the group of students reached the end of the interactive test more quickly.

	Reason Repetition / Reformulation	Rule Repetition / Reformulation	Immediate Rule Announcement
Small and medium scale companies	73%	7.6%	23%
Large scale company	64.7%	26.4%	32.3%
Graduate students	57.1%	10.7%	17.8%

Table 3 – The Percentages of Subjects within three Groups according to their Confirmation Tendencies

Our hypothesis for these two groups was that they would differ in terms of logical reasoning, and hence in their scores of the abstract questions found in pen-and-paper test. This difference was confirmed significantly in favor of the graduate students, when the number of subjects below and above median score were compared statistically ($\chi^2(1) = 8.114$, $p < .005$). Another interesting result was that the student group also performed better in questions with a software theme ($\chi^2(1) = 7.085$, $p < .01$).

Similar analyses were conducted among three more groups. All members who had software development and testing experience equal to or more than the average experience were included in the analyses. For the sake of simplicity, let us refer to the large scale company as C_1, the group of small and medium scaled companies as C_2 and graduate students as S. We first analyzed whether test results differed significantly among C_1 and C_2. Significant results were obtained with Mann-Whitney U-test for test severity, $z = -2.48$, $p < .05$ and eliminative/enumerative index, $z = -3.82$, $p < .001$. For test severity, the mean ranks were 23.36 for C_1 and 19.18 for C_2. For eliminative/enumerative index, the mean ranks were 27.07 for C_1 and 11.5 for C_2. These results indicated that, employees of the large scale company performed better in the interactive test compared to the employees of small and medium scaled companies when only subjects with an experience level higher than the average was taken into consideration.

Further analyses were conducted for the student group versus C_1 and C_2. It has been observed that the student group differed significantly from C_1 in terms of their scores in abstract questions ($\chi^2(1) = 6.773$, $p < .01$) and in software-domain questions ($\chi^2(1) = 7.38$, $p < .01$). A similar pattern of results was observed for the student group and C_2. Again, significant differences were found in the scores of the abstract questions ($\chi^2(1) = 6.677$, $p < .01$) and software-domain questions ($\chi^2(1) = 4.34$, $p < .05$). These results indicated that it was easier for the student group with the experience level equal to or more than the average experience to employ logical reasoning and use it effectively than software developers and testers of the companies.

The Analysis of Reasoning Skills. When we compared the strategies employed by the subjects in the pen-and-paper test, we observed that the percentage of falsifiers was higher in the group of students, while the percentage of verifiers and matchers was lower compared to the other group of software developers and testers of the four companies. This distribution is shown in Figure 4. The significant difference among the falsifiers, verifiers and matchers in both groups was confirmed statistically ($\chi^2(2) = 6.922$, $p < .05$).

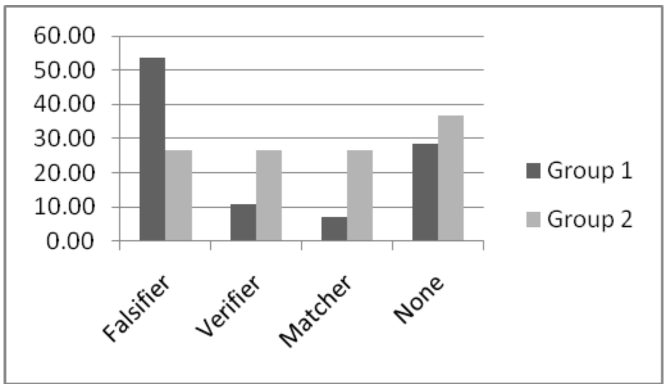


Figure 4 – The distribution of subjects according to Reich and Ruth's method among two groups, Group 1 being students and Group 2 being all employees

Since a high percentage of subjects could not be categorized according to this method, as an additional measure we have performed an analysis of 'insights'. Outcomes are shown in Figure 5, which seems to confirm the hypothesis that the percentage of people with complete insight is higher among students, whereas people with partial or no insight are less frequent in this group. These results were also confirmed statistically ($\chi^2(2) = 9.620$, $p < .01$). Hence, we can conclude that students performed better in pen-and-paper test when compared to all other subjects.

Further analyses were conducted to compare interactive test results. This time, S, the group of students, was compared to two groups, first being the large scale company C_1 and the second being the group of small and medium scaled companies C_2.

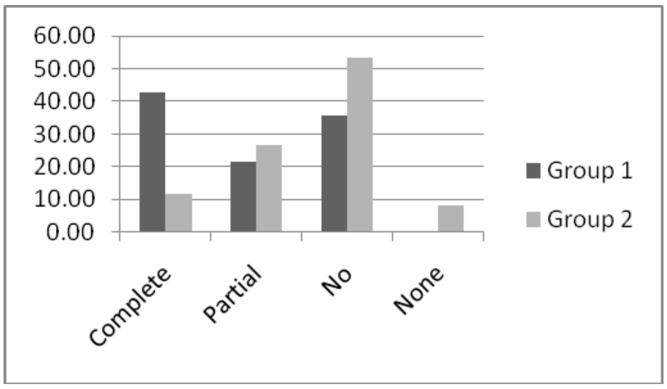


Figure 5 – The distribution of subjects according to insights among two groups, Group 1 being students and Group 2 being all employees

It has been observed that the time to finish the interactive test was significantly lower for S than C_1, $z = -3.57$, $p < .001$ and C_2, $z = -1.97$, $p < .05$. No significant difference was observed for eliminative/enumerative index.

When we compared test severities with Mann Whitney U-test, we have found a significant difference between S and C_1, $z = -2.46$, $p < .05$. S had an average rank of 24.72, while C_1 had an average rank of 35.99. This could mean that C_1 employed a better hypothesis testing strategy or exact the opposite, that they had strong confirmations. Looking again at Figure 3, we observed that the group of graduate students (G3 in the figure) also performed a desired hypothesis testing strategy starting with a mild test, increasing the severity and then again decreasing it. In order to be able to compare these two groups, we analyzed whether there was a significant difference between two groups in terms of confirmative behavior. In order to accomplish this, we took into account the subjects within groups who engaged in an activity such as repeating or reformulating a reason or a rule; or immediately announcing a rule. A significant difference was found between these groups in terms of the number of people engaging in a confirmative behavior ($\chi^2(2) = 5.939$, $p < .05$). We concluded that the group of graduate students was better at hypothesis testing. This was also supported by the fact that 27 out of 28 students found the correct rule at the end of the interactive test, whereas 29 out of 34 employees of the large scale company was able to find the rule. No significant difference was observed for test severity between S and C_2.

In order to observe more possible differences between the group of students and the large scale company, results of the pen-and-paper test are compared. Figure 6 shows the distribution of falsifiers, verifiers and matchers in both groups. This figure shows that the percentage of falsifiers among students is higher and the percentage of verifiers and matchers is lower.

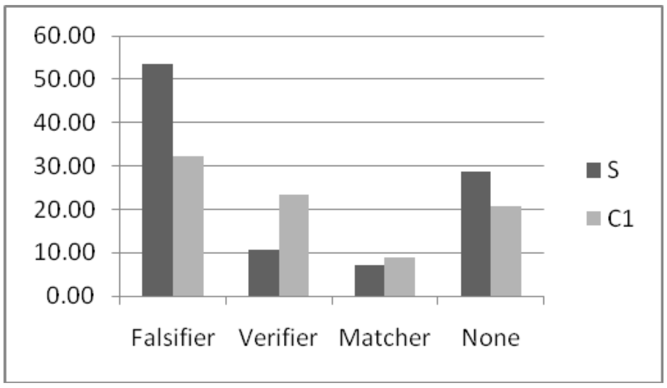


Figure 6 – The distribution of subjects according to Reich and Ruth's method among two groups, S being students and C1 being employees of the large scale company

It has been found that both groups differed significantly in terms of the percentages of falsifiers and verifiers, thus confirming the claim that there is a significant difference in both test performances between these groups ($\chi^2(1) = 4.835$, $p < .05$). The high percentage of subjects of the large scale company with no insight as shown in Figure 7 also confirms this fact.

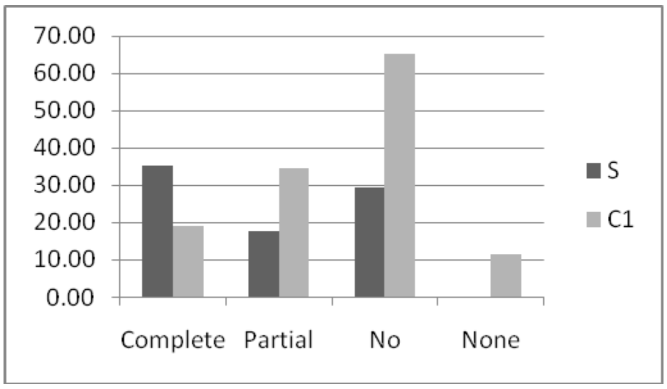


Figure 7 – The distribution of subjects according to insights among two groups, S being students and C1 being employees of the large scale company

5.2. Effects of Confirmation Bias on Software Defect Density

Analysis of the Effect of Confirmation Bias on Software Developer Performance. As a result of the lack of tendency to try to fail code during unit tests, a software developer is likely to introduce defects to his/her code. In order to analyze this phenomenon empirically, we analyzed the last ten releases of customer services and channel management software developed in the large scale telecommunications company C_1. There were 12 developers and 16 testers assigned to this software project. However, we could only perform our tests to five developers whose records appear in churn data. The rest of the development team were new to the project due to sudden change in the organizational structure.

During our analyses, we took into account only Java source codes, as churn data contained information about only Java source files. The files that are created in a release before the release of interest *R*, but just modified within release *R* are not taken into account. Our analyses include only Java source files that are created within each release. The owner of each file is determined from churn data and related defect information is obtained from the defect log of the corresponding release. As shown in Table 4, developer who gave up the interactive test (Developer1) has the highest defect ratio which is the ratio of the number of defected files to the total number of files implemented by that developer. Moreover, it took much longer for Developer1 to solve both parts of the pen-and-pencil test compared to the rest of the developers. On the other hand, no significant difference in elimination/enumeration index of Developer1 from the indices of the remaining developers was observed.

	Defect Ratio	Eliminative/ Enumerative Index	Interactive Test Duration (minutes)	Abstract & Thematic Test Duration (minutes)	Software Test Duration (minutes)
Developer1	0.86	0.83	ABORT	20	26
Developer2	0.38	1.83	12	12	10
Developer3	0.20	1.82	14	10	10
Developer4	0.00	0.75	22	9	13
Developer5	0.11	0.50	8	15	10

Table 4 – Defect Ratio versus Some Confirmation Bias Results of Developers of a Software Project of the Large Scale Telecommunications company (C_1).

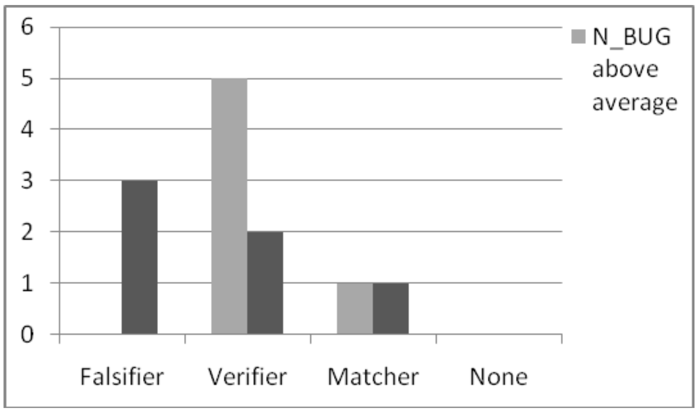


Figure 8- Distribution of falsifiers, verifiers, and matchers among testers who report bugs above and below average amount, according to Reich and Ruth's method.

Analysis of the Effect of Confirmation Bias on Software Tester Performance. In this part of our work, to analyze the effect of confirmation bias on tester performance, we inherited two tester performance metrics from tester competence reports of the large scale telecommunications company C_1. In this study we analyzed the testers of the same project group we performed the analyses about developer performance. Out of 16 testers, performance related data of 12 testers were in the tester competence reports. The remaining 4 testers had recently joined the project group due to the sudden change in the organizational structure. These metrics are the number of bugs reported (N_{BUG}) and the number of production defects caused (N_{PROD_DEF}) by each tester respectively. We grouped members of C_1 based on the values of N_{BUG} and N_{PROD_DEF}. Figure 8 shows that there are no falsifiers among testers who detect bugs above average N_{BUG} value. This group of testers also contains more verifiers compared to the tester group detecting bugs below average N_{BUG} value .

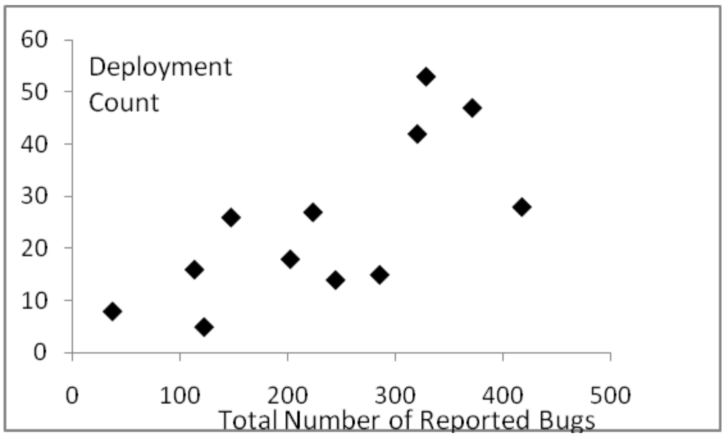


Figure 9- High Correlation between production defect and total number of reported bugs (spearman rank correlation: 0.8234)

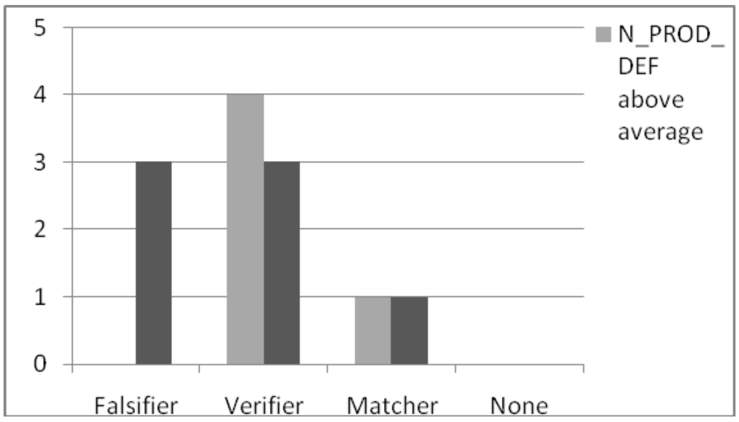


Figure 10- Distribution of falsifiers, verifiers, and matchers among testers who cause production defects above and below average amount, according to Reich and Ruth's method.

As shown in Figure 9, high correlation between total number of reported bugs and production defect count may indicate another phenomenon, namely, testers who report more bugs might be assigned codes with very high defect density requiring immense testing effort. However, for each tester there is also a time pressure to end the testing procedure and this may result in the deployment of the defected codes. Another explanation for the outcome shown in Figure 9, is that bugs are not classified according to their severities. Hence, large number of reported bugs does not necessarily mean that a significant portion of severe bugs has been reported. As a result, testers with low confirmation bias levels seem to detect more bugs. However they are more likely to overlook most of the defects which leads to an increase in production defects. In Figure 10, among testers who introduce production defects above average there are no falsifiers and this result is in line with our latter explanation.

7. Threats to Validity

In terms of internal validity, our quasi-independent variables are company size, experience and reasoning skills. In order to obtain measures for these variables, we performed both interactive and pen-and-paper tests to development and testing team working on a software project in the large scale telecommunication company (C_1) within a week. Tests were completed among graduate computer engineering students (S) also in less than a week; whereas the completion of the tests took 1 day for each of the small scale companies (C_2). Moreover, within any of the groups there was no event in between the confirmation bias tests that can affect subjects' performance.

However, problem may arise due to different experimental conditions. For instance, compared to graduate computer engineering students, stress factor of company workers due to the fact that they always have to rush the next release may have biased the results. In order to avoid mono-operation bias as a construct validity threat, we used more than a single dependent variable. We used Wason's elimination/enumeration index (Wason 1960), test severity in addition to interactive and pen-and-pencil test durations. As a result, we have avoided under-representing the construct and got rid of irrelevancies.

We have used three datasets to externally validate our results. We will continue expanding the size and variety of our dataset going forward. However, during our analysis to investigate the effect of confirmation bias on software defect density, we used data belonging to *only* five developers. This was partly because of the rapid and frequent changes in the development team. Out of 12 developers, only 5 of them were actively working on the project; while the rest were the newcomers and hence they have not started contributing to the project yet. We could not find the previous developers whose code commitment records were on the churn data, as most of them had left the company. In general, it is difficult to extract data that is informative about the defects introduced by a developer. Usually small and medium sized companies do not keep file-level defect information. Moreover, most companies do not classify defects according to their severity either. The data about the developers who contributed to a specific file, the dates of this contribution and defects related to the file differ from one company to another.

In order to statistically validate our results, we used Mann-Whitney U Test for continuous variables (e.g. test severity, eliminative/enumerative index, test durations). We used Mann-Whitney U Test, since we do not have any prior knowledge of the distribution of these values. For categorical variables such as number of falsifiers, verifiers, matchers, we used Chi Square test . Chi Square test was also used to evaluate the significance of the distribution of the subjects according to insights by Johnson-Laird and Wason.

8. Conclusion and Future Work

We have shown that there is no significant relationship between software development or testing experience and hypothesis testing skills. We concluded that experience did not play a role even in familiar situations such as problems about software domain.

The most striking difference was found between the group of graduate students and software developers and testers of the companies in terms of abstract reasoning skills. The fact that students scored better in software-domain questions although most of them had less software development and testing experience indicates that abstract reasoning plays an important role in solving everyday problems. It is highly probable that theoretical computer science courses have strengthened their reasoning skills and helped them to acquire an analytical and critical point of view. Hence, we can conclude that confirmation bias is most probably affected by continuous usage of abstract reasoning and critical thinking.

Company size was not a differentiating factor in abstract reasoning, but differences in hypotheses testing behavior was observed between two groups of companies grouped according to their sizes. The large scale company performed better in the interactive test, but it has been shown that the group of students outperformed this group in terms of both tests. This has led us to perform additional analyses and reach the conclusion that hypotheses testing skills were better in the group of students. Thus, we conclude that there is a relationship between confirmation bias and continuous usage of and training in logical reasoning and critical thinking.

The analysis we made among developers and testers of the large scale telecommunications company, showed that there is a direct correlation between confirmation bias and defect proneness of the code. This is due to the fact that including unit testing in the development phase, all levels of software testing should aim to fail the code, which implies that both testers and developers should have low confirmation bias levels. However, in order to obtain Statistically significant results we need more data.

As future work, we intend to increase the size of our data regarding total number of defects introduced by each developer per lines of code changes made by that developer. Recently, we are collecting data from a company developing software for baking services and this data shall belong to 100 software developers and testers. All these data will help us to empirically analyze the effect of confirmation bias on software defect density to obtain statistically significant results.

7. Acknowledgements

This research is supported in part by Turkish Scientific Research Council, TUBITAK, under grant number EEEAG108E014.

8. References

Calikli, G., Bener, A., and Arslan, B. (2010) An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers, to appear in the Proceedings of 32nd International Conference on Software Engineering (ICSE 2010).

Evans, J. St. B. T. *(1972) Interpretation and* m*atching* b*ias in a* r*easoning* t*ask, Quarterly Journal of Experimental Psychology,* 24, 193-199

Evans, J. St. B. T., Newstead, S.E., Byrne, R. M. J. (1993) Human Reasoning, The Psychology of Deduction.

Hilgard, E. R. (1938) A summary of alternative procedures for the construction of Vincent curves, Psychology Bulletin, 35, 282-297.

Kahneman, D., Slovic, P., and Tversky, A. (1982) Judgment Under Uncertainty: Heuristics and Biases, New York: Cambridge University Press

Poletiek, F. (2001) Hypothesis Testing Behavior (Essays in Cognitive Psychology), Psychology Press Ltd.

Reich, S.S. and Ruth, P. (1982) Wason's selection task: verification, falsification and matching. British Journal of Psychology, 73:3, 395-404.

Stacy, W. and MacMillan, J. (1993) Cognitive bias in software engineering, Communication of the ACM, 38, 6 (June 1995), 57-63.

Teasley, B., Leventhal, L. M., and Rohlman, S. (1993) Positive test bias in software engineering professionals: What is right and what's wrong, In Empirical Studies of Programmers: Fifth Workshop

Wason, P. C. (1960) On the failure to eliminate hypotheses in a conceptual task, Quarterly Journal of Experimental Psychology (Psychology Press), 12. 129–140.

Wason, P. C. (1968) Reasoning about a rule, Quarterly Journal of Experimental Psychology (Psychology Press) 20: 273–28.

Enhancing user-centredness in agile teams: A study on programmer's values for a better understanding on how to position usability methods in XP

Michael Leitner[1]; Peter Wolkerstorfer[1], Arjan Geven[1], Manfred Tscheligi[1,2]

(1) CURE – Center for Usability Research & Engineering
Modecenterstraße 17 / Objekt 2
1110 Vienna, Austria
{leitner | wolkerstorfer | geven | tscheligi}@cure.at

(2) ICT&S Center – University of Salzburg
Sigmund-Haffner-Gasse 18
5020 Salzburg, Austria.
manfred.tscheligi@sbg.ac.at

Keywords: XP Programmers, programmer' high-level goals, user centred methods in agile teams

Abstract

We present a study on programmer's high-level goals in eXtreme Programming settings (XP). We talked to 10 programmers using so-called "laddering interviews". The result presented is a "Hierarchical Value Map (HVM)" indicating agile programmer's high-level goals. This study was done to better orchestrate usability methods and integrate them into agile development processes. The study's results were used to position and adapt usability methods in a way that they are better aligned to the programmer's goals and therefore are more likely to be accepted. We draw conclusions on the basis of the study's results and experiences using agile usability methods in practise.

1.Introduction

User-centeredness describes the fact that technology design and development are aligned and targeted at the end-user needs, requirements and limitations. To achieve this, different methods are applied to secure the usability and user experience of hard- and software. In classical development processes, these methods (e.g. user requirement analysis, usability reviews, etc.) are included in the development process iteratively and at certain stages of the process. There is plenty of know-how in reference to the use of usability methods in classical soft and hardware development processes. In contrast, for agile settings different authors have discussed the application of usability methods in agile settings, but still little explicit know-how is available of how to adapt and position these methods in agile development settings. With this work we define a further step towards the adaption and use of usability methods in agile teams by proposing essential enhancement of these methods.

In order to identify these necessary adaptations of usability methods and to define a strategy to better position these methods in the course of the agile work processes we need to consider that usability methods - in most of the cases - are not the prime focus of XP programmers, not under their constant attention nor necessarily fit for application in XP processes. Therefore, to achieve a higher user-centeredness and an enhanced usage and acceptance of usability methods in agile teams the following two pillars need to be fostered:

a) Adapt and position usability methods in a way that they fit the agile team structure and process without disturbing the primary task: software development (=adoption towards the organizational and process goals).
b) Align and position the usability methods towards the programmer's goals and values[1] in order to achieve acceptance and use of these methods beyond indoctrination (=adoption towards psychological and team member's goals).

[1] In this paper the term "(high-level) goals" equal the term "values". To define and describe "values/high-level goals" we refer to Rokeach's Value List, which classifies terminal and instrumental values. Terminal values describe desirable end-states of existence (= goals that a person would like to achieve during his or her lifetime). Instrumental values describe preferable modes of behaviour to achieve a terminal value (Rokeach, M.).

With the study at hand we especially want to aim at pillar b) (programmer's goals and values) trying to align and adapt usability methods on the basis of programmer's goals and valued circumstances of agile development. Hence, the objectives of this study are:

1. *Identifying programmer's high-level goals:* We conducted a study on programmer's goals (and values) in agile team settings to enhance the understanding of how to position and adapt usability methods respectively. For this study we chose a semi-structured interview technique called "laddering" that is used to identify people's high-level goals in reference to a given object (in our case: the agile team and development setting) and the interview data is presented using a so-called Hierarchical Value Map (HVM). We discuss these findings in reference to the current know-how in the area of "perception of agile teams" and highlight the relevant findings for our study.
2. *Adapt and position usability methods accordingly:* On that basis we discuss how usability methods need to be adapted and positioned in agile settings in order to be better accepted and aligned to the programmer's values. We are discussing the value study's results and use these as a basis for a deeper look into the need of methodological adaption. We propose a set of particular improvements of selected usability methods for their use in agile settings. Finishing up we discuss some practical experiences that underline our argumentation.

Overall our study extends the State of the Art in two areas. On the one hand we are able to show new findings and aspects in the perception of agile methods by agile team members (programmers) and on the other hand we extend the know-how in the area of usability engineering in and for agile teams.

In general this study was conducted in a project dedicated to orchestrating usability and agile methods. In this project an agile developer team is working on a certain piece of technology. The usability methods that are used in this project are "personas", "usability evaluation and ad-hoc usability (expert) evaluation/input" and to some extend as well "automated usability evaluation tools[2]". Hence, in this paper we focus on improving these specific methods.

Further, it is necessary to mention, that in the team setting of this project no usability experts are part of the team, in contrast all team members are dedicated to programming. Therefore all usability input is brought to the team from as external source (The external source in this case are the authors ["we"] of this paper, which are supporting the agile team with usability input and methodological know-how. Hence, all practical experiences reported in this paper describe the author's observations as "external members" of the agile team).

The exact setting of the project and methodological approaches are described in (Wolkerstorfer, 2008). We are currently using the above mentioned usability methods in our project together with the agile development team, so far mainly without the suggested improvements in this paper. Our aim is to adapt the methods accordingly (on the basis of the improvements in this paper) and evaluate these in the field and together with the agile team we are working with.

We start with a discussion on current state of the art in related fields, discuss the study setting on programmer's goals and finish up with a discussion on enhancement of usability methods for a fruitful integration in agile development settings.

2. State of the art

In this section we summarize related work that is relevant for the study. This is on one hand research in the area of "perception of agile development and agile team settings" and on the other hand research and studies towards the use of usability and user-centred methods in agile team settings.

[2] In this context an "automated usability evaluation tool" describes software with the ability to assess the usability of a graphical user interface (GUI) automatically - for instance through the means of image analysis. The tool's feedback might be an index describing the GUI's visual complexity from 1 (bad) to 100 (good).

Perception of agile development settings

Different authors have studied the perception of agile development methods in the last years. Several positive effects and perceptions were reported in reference to this organisational structure and work setting.

Dyba and Dingsøyr (2008) conducted a comprehensive state of the art analysis of research in the area of XP Programming. Citing Mannaro et al. they state that 90 % of XP Programmers would like to continue in the company they work, whereas only 40 % of Programmers in non-XP companies would like to do so. According to Mannaro's study XP Programmer's job satisfaction is higher (in comparison to non-XP Programmers). Further, XP programmers claim that their productivity is higher in XP settings. Citing Mann and Maurer Dyba and Dingsøyr state that XP Programmers think that the XP processes allows them to work more pinpointed towards the customer's goals.

Tessem and Maurer (2007) conducted semi-structured interviews and concluded that agile programmers feel "autonomy" (in daily work), "variety" (of type of work), "significance" (of each agile programmer in the team), "feedback" (related to own work done), "ability to complete whole tasks" (be responsible for the whole thing) and "motivation and job satisfaction". Likewise Law and Cheron reported the important factors "motivation" and "autonomy" in XP teams (2005). Whitworth (2007) conducted interviews with agile teams and reports similar findings (beside others). According to this work programmers reported that they "appreciate the knowledge share" and that the agile environment "amplifies the effectiveness of team meetings" and furthermore that these are a "forum for motivation". The agile setting would "enhance the team awareness", "foster the team-awareness", "the awareness of contribution of individuals". According to Withworth's study agile team setting supports team members taking a more active role and overall a positive self-image and self-esteem is emphasised/leveraged.

So far different studies and papers discussed programmer's perception on XP team settings, however, with our study we introduce a more "personal" view on these settings. Similar work was done by Withworth who introduced a more psychological view on XP team settings. With our study we enhance this and other studies by the following points:

a) Research the personal high-level goals (values) of programmers in XP work settings. So far no studies were explicitly dedicated to this factor in agile teams.

b) By the use of the means-end theory we are able to display the results gathered in a visual way (in a so-called "Hierarchical Value Map"). This visualization technique helps to understand the correlation between developer's high-level goals (values) and the attributes of the XP work setting.

Usability methods and user-centredness in agile teams

When Kent Beck introduced XP (extreme programming) in 1999 (Beck, 1999) UCD (user centred design processes) did not play a remarkable role in his considerations. These considerations followed later, as agile development was adopted by companies with greater frequency every year which lead to a special interest group on agile user experience (Miller & Sy, 2009). Some experts doubt that the XP process leads to true user-centred design (Hudson, 2005). The issues arising from this problem statement suggest that XP and UCD won't fit. But this perception is simplistic and misguided which has already been shown in practice where success is reported (McInerney & Maurer, 2005). We can see succeeding practitioners combining UCD (user centric development) and XP/agile methods by varying approaches (Constantine & Lockwood, 2003; Ferreira, Noble & Biddle, 2007; Göransson, Gulliksen & Boivie, 2003; Holzinger, 2005; Holzinger, Errath, Searle, Thurnher & Slany, 2005; Norman, 2006; Wolkerstorfer et al, 2008).

Various studies have been directed to usability in agile settings. However, so far no particular improvement and necessary adaptations of methods have been discussed. Holzinger (2005) claimed that it is very important to know which usability method to select. Ferreira et al (Ferreira et al, 2007) give insight into the changes of the XP process to better integrate UCD work. They argue that 1. iteration planning affects UI design, 2. the development iteration drive the UCD activities, 3. usability testing results in changes in development and 4. that agile development changes the relationship between software developers and HCI engineers. Finally Wolkerstorfer (2008) discussed the idea of adopted personas (adopted to better fit XP), expert evaluation (the adoption to XP was that the expert feedback given was presented in XP-story style cards), and automated usability evaluations included in the unit-test system as a method-mix to provide flexibility for this selection.

Overall, the current state of the art especially shows a lack of adequate user-centred methods for agile settings. Although work has been directed to usability methodology in agile settings so far no particular improvement and necessary adaptations of particular methods have been presented. With this work we extend the current state of the art by proposing particular improvements of usability and user-centred methods in order to be more usable in agile teams and further more accepted by programmers. We try to close this gap by postulating necessary adaptations to certain usability methods.

In the next section we present the study setup of the conducted study on programmers high-level goals (values) and discuss the results presented in as a Hierarchical Value Map (HVM). On the basis of these results we then discuss the necessary adaption towards goal-oriented usability methods for agile settings.

3. Study: Programmers high-level goals (values) in XP settings

This study was done to better understand programmers in agile teams in order to adapt and introduce usability methods. We start with the definition of the study's goals and proceed with a reflection on the method that we used ("laddering interviews") and its underlying theory: the means-end theory.

3.1 Study goals

This study was conducted to uncover the motivational aspects of agile settings. In particular the studies goals are:

- Identify programmer's high-level goals (values), corresponding attributes and consequences as perceived by agile developers.
- Display the results in a visual way in order to show correlations between attributes, consequences and high-level goals (values).
- Align our study's results with existing knowledge and postulate new findings in reference to agile team perception.

3.2. Hierarchical value maps and laddering interviews

Hierarchical Value Maps (HVM) are based on the means-end theory and represent users´ underlying cognitive structures when using a product. HVMs are built to identify important meanings that consumers associate with products (Reynolds, 2001). Overall, the means-end theory distinguishes three abstraction levels of meanings: *attributes*, *consequences* and *values*. First, attributes are on equal terms with characteristics of a product (e.g. a hard disk on a mobile multimedia product). Consequences are more abstract and describe possibilities offered by the product's characteristics; hence, the product is enabling the user to execute particular actions and behaviours (e.g. the hard disk of a mobile device may be used to store certain amounts of songs, which may then be shown to friends). Lastly, values represent abstract meanings, motivational constructs and beliefs that are directly tied to emotions (e.g. presenting a song-collection to a friend is *entertaining* and makes someone *happy*). As a result the means-end theory creates links between the different levels of abstraction and shows which attributes are important to users and to what consequences and values these attributes relate. The link between attributes, consequences and values is referred to as a Means-End Chain (MEC).

The methodology has its origin in marketing research and was used – as the description above highlights – to link basic product features to high-level goals (values). However, we have used the method for different purposes and concluded that under certain restrictions it is as well applicable for the purpose of this study, which is not dedicated to particular product but to an organisational setting.

Different problems have been reported in reference to laddering and means-end theory depending on the way HVMs are used[3] (Grunert and Grunert 1995; Gutman, 1997). These problems mainly occur when HVMs are used as a representation of cognitive structures that are used to explain or predict (for instance) buying behaviour. In our study – in contrast to this "cognitive view" - we use a HVM that represent a "motivational view" that gives a deeper insight to a specific area – in our case agile development – without claiming any power to predict or explain behaviour. HVMs that are used as a "motivational view" (which is how the authors interpret the results of this study) mainly avoid the reported problems (Grunert and Grunert 1995; Gutman, 1997).

3.3. Study setup and participants

We conducted semi-structured, qualitative, "laddering" interviews with 10 participants. We chose this method as it was designed to extract relevant means-end chains and values related to a product or service. We had already used the method in the realm of mobile multimedia and experienced it as a powerful tool for the extraction of particular behaviours and motives. Respondents were in the age of 25 to 40 (mean 33) and professional programmers. All participants were male and were at the time of the interview an active member of an XP developer team. The programmers interviewed were not part of the same agile development team; in contrast they were members of different agile teams (working even in different companies). As mentioned in the introduction of this paper, this study is part of a project in which the authors are collaborating with an agile developers team in order to orchestrate usability methods with agile structures. It is important to mention that for the interview series no programmers of this particular project were interviewed. Instead all participants were recruited from project external sources.

To create a better understanding of the methodology that were used, the next chapter gives an overview on the interview conduction as well as on the data analysis phase.

3.4. Interview conduction and data analysis

To indicate how Laddering interviews are conducted the following example taken from the study at hand is provided:

Interviewer: How do you personally perceive the XP team setting, how would you describe the XP setting?

Interviewee: I think it's a pinpointed work setting. [Attribute: "pinpointed"]

Interviewer: Why is this important to you?

Interviewee: I like this as this gives me the impression to work on something qualitative, something that is more likely to be used. [Consequence: "increase quality of the product"]

Interviewer: Why is this important to you?

Interviewee: It's somehow that I feel my work is honoured.

Interviewer: Why is this important to you?

Interviewee: It's a kind of award and appreciation to my work. [value: "appreciation / reward"]

The interviews talking to 10 participants provided us with 33 valid interview sequences - like the one described above - starting from attributes ("how do you personally perceive the XP team setting?") over consequences to values ("why is this important to you?"). Note that an interview sequence was considered as valid if it was started by an attribute, contains at minimum one consequence and

[3] The authors do not consider any further discussion on methodological issues in relation to means-end theory as relevant for the focus of this paper.

terminates in a high-level goal/value – the interview sequence was considered as invalid if it failed to terminate in a high-level goal/value due to different reasons, e.g. the respondent was not able to formulate a high-level goal. Overall, there were 4 invalid interview sequences that were not taken into account in data analysis. One interview sequence is also called a "Ladder", describing the subsequently raised level of abstraction in the interview sequence.

From answers to Content Codes: The next step in the analysis procedure was to merge the 33 valid interview sequences. In this step similar answers were grouped and summarized in certain categories, so-called "Content Codes". The challenge in this step was to identify answers with similar meanings among respondents and to find meaningful names for these categories ("Content Codes"). For instance, answers from respondents that refer semantically to the attribute "pinpointed" were classified under one Content Code, even if each of the respondents had named it by different wordings (e.g. "very accurate" and "exactly defined" would be classified under one category as they refer semantically to the same quality of the object in question). The result of this step was an overview of several groups of attributes, consequences and values that were mentioned by the participants and clustered by the authors. The groups are summarized in Table 1. Overall we identified 5 groups of attributes, 12 consequences and 4 groups of values. For each group we chose a meaningful name representing similar types of answers.

	Attributes	**Nr.**
1	challenging	6
2	flexible and adjustable	3
3	team-oriented and communicative	12
4	pinpointed and quality enhancing[4]	7
5	easy and relieving	5
	Consequences	
6	create something new	2
7	like diversified work	3
8	like challenging work	4
9	work on a democratic basis (communication and feedback)	8
10	define exact goals and pieces of work	2
11	work on something valuable/qualitative (enhancing quality)	7
12	work within a simple / changeable process / environment	6
13	work on interesting / personally preferred things	2
14	like felt progress of work (productivity)	8
15	avoid errors	2
16	learn new things/solutions at work	10
17	more motivated / engaged	6
	Values (taken from Rockeach's Value List)	
A	ambition / professional and personal advancement	12
B	appreciation / reward	3
C	satisfaction	5
D	happiness / relief / relaxation	13

Table 1: Content Codes are categories summarizing similar respondent's answers. The last column indicates the number of answers that were classified in this category.

[4] The expression "quality enhancing" summarizes participant's comments, which indicate that agile setups enhance the ability to develop a software product with a higher quality (in comparison to non-agile settings).

<u>*From Content Codes to Ladders:*</u> In the subsequent task this classification and grouping of answers on the basis of Content Codes allows the identification of the most relevant means-end chains. As discussed above, a means-end chain describes the correlation between attributes, consequences and high-level goals (values). In other words a means-end chain describes how a particular attribute, for instance an organisational structure that is perceived as pinpointed and quality enhancing, is supporting a particular consequence, for instance the ability to "increase the quality of a product" and finally is terminating in a certain high-level goal (value), for instance in the feeling of "happiness and relief". Note that a means-end chain is a result of several Ladders (interview sequences).

Once Content Codes were defined, each participant's interview sequences were translated into a sequence of Content Codes. Table 2 indicates a respondent's (respondent 2) interview sequence already translated to Content Codes. The proper translation of the Ladder displayed in bold characters indicates that respondent 2 stated that XP programming in his opinion is "team oriented and communicative" (Content Code 3). This is important to him as he "likes diversified work" (Content Code 7). Further, this is important to him as he "likes the felt progress and felt productivity" (Content Code 14), which is enhanced by the team-oriented and communicative setting of XP. Finally the felt progress is important to him as this increases his feeling of "satisfaction" (Content Code C). Note that Ladders are not bound to any particular length, however, each Ladder starts with an attribute, succeeds with one or more consequences and terminates with a high-level goal (value). In the given example the attribute ("team oriented and communicative"), as well the two consequences ("like diversified work" and "felt progress and productivity") support the value "satisfaction". This means that every item of a means-end chain is important to support a specific value. However, some of these items are directly "connected" to a value, some of them are indirectly related to a value (In the given examples "satisfaction" has a direct relation to "felt progress and productivity" and two indirect relations to "like diversified work" as well as to "team oriented and communicative").

Attribute	**Consequences**		**Value**
4	6	8	C
3	15	16	A
3	**7**	**14**	**C**

Table 2: Respondent's 2 interview sequence transformed to Ladders indicating the Content Codes in which the answers where classified.

<u>Implication Matrix:</u> Identifying means-end chains on the basis of several Ladders follows a pre-defined procedure (Reynolds, 2001). The categorized total amount of 33 Ladders was summarized by the so-called "Implication Matrix" as shown in Table 3. This matrix indicates the number of direct relations between the particular Content Codes in the total expenditure of Ladders. Once the Implication Matrix is constructed one is able to identify the most important nodes. In Table 3 the Implication Matrix indicates all relations higher than 2 (marked in grey). This is the cut-off level we chose for the given study. This means that relations are considered as relevant if at least two Ladders named by respondents show a direct relation. To analyze one starts with the first row in the application matrix observing nodes with a number of relations higher than the cut-off level. For instance, in Table 3 junction 2/14 shows a number of 2 direct relations. Further, one skips the row to Content Code 14. Again, node 14/A shows an amount of relations higher than the cut-off level. The constructed means-end chain is: 2, "flexible and adjustable" (attribute) – 14, "like felt progress of work and productivity" (consequence) – 4, "ambition / professional and personal advancement" (value).

	attributes					consequences												values			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**A**	**B**	**C**	**D**
1							1		2							2	1				
2														2		1					
3							2		4		1				2	3					
4						1			1	1	2	2									
5										1		3	1								
6								1										1			
7								1						1		1					
8						1												1		1	1
9								1			1			1		1	2				2
10														1			1				
11																1	1		1	1	3
12											2		1	1							2
13														1			1				
14											1							4	1	1	1
15																1				1	
16								1						1				4		1	3
17									1			1						2	1		1

Table 3: Implication Matrix: From Content Codes and Ladders to means-end chains and the HVM. Arrows indicate the construction of one "mean-end chain" on the basis of the implication matrix. Other means-end chains are constructed similarly and are displayed in the HVM. In this study we did consider a "cut-off" level of 2 meaning that attributes, consequences and high-level goals (values) are considered if they exceed two mentions (entries marked in grey).

In this manner the whole HVM (see Figure 1) was constructed. Doing all relevant nodes of the Implication Matrix where taken into account (= all nodes that show a number higher than the cut-off level, in Table 3 highlighted in grey). The HVM is a visual summary of all relevant relations and important Content Codes (Figure 1). Note that Content Codes with no relevant relations are not further considered (In our case Nodes below 2). Further details on the analysis process are presented in (Reynolds, 2001).

3.5. Study Results

On the basis of that analysis procedure the visual layout in Figure 1 was derived, representing the Hierarchical Value Map for XP Programmers. It describes the most dominant answers grouped and classified by the defined Content Codes. Note that overall the interview series with the 10 respondents produced 33 Ladders (One participant constructed more than one Ladder). Furthermore only the groups of answers (Content Codes) with a number of mentions higher than the cut-off level are displayed in the Hierarchical Value Map meaning that only the strongest correlations between Content Codes are considered in the construction of the HVM.

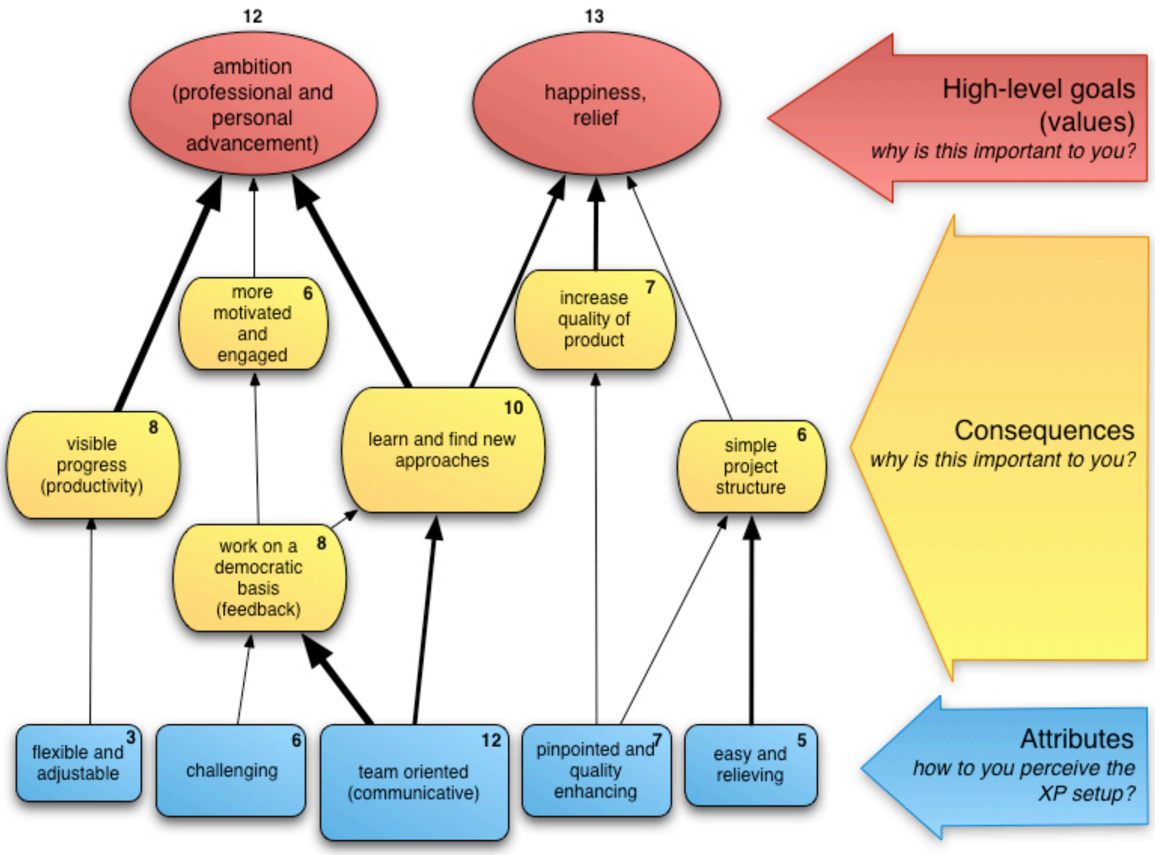

12
ambition (professional and personal advancement)
13
happiness, relief
High-level goals (values)
why is this important to you?
more motivated and engaged 6
increase quality of product 7
visible progress (productivity) 8
learn and find new approaches 10
simple project structure 6
Consequences
why is this important to you?
work on a democratic basis (feedback) 8
flexible and adjustable 3
challenging 6
team oriented (communicative) 12
pinpointed and quality enhancing 7
easy and relieving 5
Attributes
how to you perceive the XP setup?

Figure 1 – Hierarchical Value Map for XP Programmers. At the bottom the named attributes are displayed, in the middle the resulting consequences and at the top level the high-level goals (values). The digits in the body of the items indicate the number of answers that are categorized under this Content Code (Compare to Table 1). The thickness of arrows indicates the strongness of the correlation between the items (the thicker the stronger a correlation – Compare to Table 3). Note that for reasons of readability the described Content Codes in Table 1 are abbreviated.

Figure 1 gives an overview of the relevant items that were extracted of this interview series. Overall, the HVM gives a very personal view of programmer's perception of XP settings. As general conclusion we are able to state that XP is highly people centred and emotionally satisfying due to its team and organisational structure. The most important and interesting facts of this study on high-level goals and values are summarized in the following way:

<u>Team orientation and inter-team communication:</u> Figure 1 clearly indicates the most important fact influencing the perception of XP settings is the communicative and team-oriented work setting. This basic attribute supports a) the work on a democratic basis, which is heavily important to programmers as it provides several channels for feedback on their own work and b) due to the dynamic nature of the XP setting it gives programmers the impression to be able "to learn and find new approaches".

<u>Learn and find new approaches:</u> This consequence was identified as the most important consequence. In the interview series the study participants expressed that working on a "collective code" provides possibilities to access colleagues and co-workers solutions on particular problems. Further and on the basis of participant's comments and statements "pair programming" – a more interactive way to learn from co-workers - was perceived as a "forum" to learn and benefit from each other.

<u>Ambition (professional AND personal achievement):</u> As indicated in Figure 1 the study series revealed that XP settings are able to benefit programmers beyond the professional working borders. In the interviews participants expressed that by this flexible work setting they are able to align their work

to their personal interests. This lead to a perception of achieving something in a professional but as well in a personal way as the setting is able to support personal interests and ambitions.

The findings displayed in the Hierarchical Value Map confirm the results found in other studies. Especially findings of Whitworth (2007) who indicated that team-awareness, motivation and enhanced effectiveness and increased self-esteem are one of the most important points in agile settings. Other findings referring to satisfaction and better self-image are related to the findings of this study. However, by applying the means-end theory and using laddering interviews we tackled this area of research with a new and fruitful approach. Our study clearly provides new insights by structuring the findings in attributes, consequences and values. Secondly, an added value of the hierarchical visualization (=HVM) is identified, which shows linkages between attributes, consequences and high-level goals (values) in a structured way.

In the following step we use these findings on high-level goals (values) and motivational aspects of agile programming to discuss the use and necessary adaption of usability and user-centred methods in agile team settings.

4. Impact on Usability Methods in Agile Teams

4.1 Agile-oriented Usability Methods

In this section the study results are used to discuss different enhancements of usability methods for a better application in agile setups. The objectives are twofold:

a) We discuss the necessary adaptation and positioning of usability methods upon the results of this study on programmers values. In this chapter we define a strategy of how to integrate and adapt the usability methods in order to support the programmer's goals. In order to do so we elicited the most potential programmer values and motivational aspects from the foregoing study from the current point of view. For a deeper discussion about value impacts on the use of user-centred methods a broader range of values can be taken into account. For the current study we aim to consider the most predominant values for our purpose.

b) As mentioned in the introduction of this article the described study on programmer's values is part of a project dedicated to the orchestration and enhancement of usability methods in agile teams. In the following chapters only the usability and user-centred methods that are used in this project are discussed, which are the following (compare to [Wolkerstorfer, 2008]):

   - *Personas:* These are archetypical descriptions of real users, representing the target user group. Personas are often described in a narrative way and are designed to help software developers to get a better understanding of the real end-user they are developing for (Pruitt, J. and Adlin, 2005).
   - *Usability evaluation and ad-hoc usability (expert) evaluation/input:* These are ongoing usability reports (of the currently available solution) and as well ad-hoc inputs on current issues in development (given orally or face2face or via video or audio-conferencing).
   - *Automated usability evaluation (system evaluation) as extension to unit tests:* These are test cases that are integrated to the nightly builds. These tools should be able to indicate a certain level of usability of the current version. For this usability method we draw some principle implications but do not go in any detail due to the lack of practical experiences.

For our purposes these methods need to be adapted to the specific agile context in order to be better aligned to programmers' goals. The overall objective is to position these adapted methods in a way that leads to a higher acceptance level and an increased use (for teams that have no specific team member working on usability).

In order to enforce our methodological suggestions we discuss practical experiences that we gathered using "Personas" and "Usability evaluation and ad-hoc usability input" in practise. We are aware that this feedback is no valid proof of the methodological improvement suggested by this paper, however we believe that these short case studies provide a good insight and support our arguments. Note that for automated usability evaluation (AUE) no practical experiences are available at the moment. For this part we draw principle conclusions without any further discussion.

4.2. Implications and Conclusions for Usability Methods in Agile Teams

For each of the usability method defined above particular potential and necessary adaptations are required. For each of the methods that are discussed in the following paragraphs we elicited the high-level goals (values) that were found in this study and that have the most potential impact on the methods from the current point of view. On the basis of these selected results we conduct the following discussion. We are aware that for a broader discussion several values of the foregoing study can be used, however, for the purpose of this study we stick to the most dominant ones. We are further aware that these suggestions need to be evaluated in the field before we can draw any final conclusions. In general and on the basis of this paper a more detailed study evaluating the suggested improvements needs to be conducted.

Personas: In order to use personas in agile settings this methodology needs to be adjusted towards a more flexible use in the project setting. The traditional definition of personas foresees no changes to a defined personas description. In classical development process the personas are defined once (typically at the beginning of a project) and are not changed during the project. This method is too rigid for XP projects when it is used in its traditional form.

Personas can be used to focus on the programmer's consequences of "visible progress" and "increase quality of the product" in the following ways:

- Visible progress: It is very important for programmers to see the progress of their work. Static personas descriptions do not support this programmer goal. The implication is that personas need a list of "usability problems" (or something similar) that needs to be updated according to the work done by the developer team. This feedback serves as a "visible progress" as the list of usability problems for a specific personas decreases due to a checked-in solution. Likewise the usability problems of a persona could as well be added due to a current solution that was checked in. This would as well lead to a more visible progress in reference to usability problems.
- Increase quality of the product: Overall we argue that the use of personas in agile team settings supports programmer's aim to increase the quality of the product. Whereas in the interview study "quality of code" was mainly used as an indicator of quality, we believe that personas (or usability in general) can serve as alternative "metric" of quality of a software product in XP setups. In order to do so the persona marketing method must be enhanced (e.g. personas can send e-mail to programmers or personas can communicate over the programmer's mailing list discussing progress of the solution). Such approaches would introduce a new metric of the quality beyond "quality of code" (which is of course an important metric – however, new ways of quality assurance can be introduced by this way).

To strengthen these conclusions we report the following practical experiences using personas in agile settings[5]: Applying this method in our project on user centred-methods in agile setups we were using a quite traditional personas approach for the agile developers team. After a while we noticed that we used too little marketing and that we failed to adapt the personas description in a flexible way. The result was that the personas were not visible and therefore not used or considered as important in the project as we expected them to be. As extreme example of our case study the personas description posters were removed from the developer's room. We are aware that this is just one experience of a particular project and the reasons for the removal have not been discussed in detail with the agile programmers team, but we believe that this example is a quite striking describing the potential (mis-) use of personas in XP projects.

[5] Note: These observations were collected in a project that used agile development (as mentioned in the introduction of this paper). In this project the personas method was used without the suggested improvements claimed in this paper.

Usability evaluation and ad-hoc usability input: Traditionally, usability reviews done in structured ways result in a usability report that might be considered at various stages of a software project. Such reports may have numerous pages listing different problems and solutions. We believe that this form of input is not suitable for agile settings. Likewise the following programmer's values are not considered by this method:

- Simple project structure: Usability input has to consider the need and the expectation of a simple project structure. Highly elaborated usability reports are not supporting this goal. In contrast usability input needs to be selective in the form of little and well-structured pieces. We propose to feed the developer team with particular "usability user stories" and possibilities of ad-hoc usability input. This fits the programmer's expectations better than whole usability reports (with a lot of pages to consider and to disseminate, etc.)
- Work on a democratic basis: Referring to ad-hoc usability input the usability experts need to be accepted as "full project team member". We argue that by imposing usability reports on the team the developer's sense of a "democratic setting" is decreased. In contrast a bidirectional relationship has to be established.

To strengthen these conclusions we report the following practical experiences for the use of usability reports and ad-hoc expert usability input[6]: Working with the agile programmers we applied both, usability reports and ad-hoc input that was requested by programmers whenever they need the input. We noticed that the programmer teams did not value large usability reports in classical forms as much as expected (note: often a 40 pages report). Programmer's feedback was that they perceived this input as too exhaustive. In contrast, usability bugs reported like user stories had major impact (and were often considered). We got very positive feedback on this form of usability expert input e.g. via given via Skype or face-2-face. We got the impression that orally discussed problems (in contrast to written formal reports) satisfy the attributes of team-orientation and "simple project structure" in a much better way. An other form of creating usability awareness with major impact was observed during usability testing. As has been found elsewhere, programmers who were physically present during usability tests, watching and observing users dealing with their software, were observed to be quite concerned with usability.

Impact of automated usability evaluation (AUE): Automated usability evaluation can be a fruitful instrument to create a metric on the basis of ongoing code development. The idea of this setting is to check usability automatically as extension to unit tests. This could serve – similar to personas descriptions – as a qualitative metric of increased or decreased usability index. We argue that this would support the programmer's consequences "visible progress" and the wish to "increase the quality of the product". We conclude automated usability evaluation tools should be an add-on to the unit-tests with visible output in relation to usability metrics for each nightly build report (or similar). A set of metrics for each build (graphs or numbers) has to be introduced that communicates results to programmers in order to guarantee awareness. In this way AUE should be included in the workflow of the developers. As mentioned above, for these suggestions no practical experiences are reportable at the moment.

Summarizing we conclude that there is a need for a well-defined mix of usability methods (Without claiming that the methods that are discussed above constitute the whole range of methods that should be used in a setup – of course there should be more methods brought together). However, applying traditional usability methods in agile setups might lead to mis-use or non-use of these methods. In contrast, different adapted methods should be available that can be used flexibly by programmers. Overall the strategy is not to apply as many methods as possible within the agile setting but a good and well-defined range of methods that are adapted to the context of use accordingly.

[6] Note: These observations were collected in a project that used agile development. In this project usability evaluation was conducted without the suggested improvements claimed in this paper.

5. Conclusion and Future Work

We conducted a study on programmer's values in agile settings. The results of this study were used to discuss and adapt usability methods for the use in such agile settings.

With this study we are able to extend the current state of the art in two ways:

- *Identification of programmer's high-level goals (values):* We confirmed and extended current studies in the area of "perception of agile team settings" and introduce a new way of visualizing different facts about XP programmer values. We conclude that "team-orientation" is one of the most important facts about agile settings. Programmers like the fact that they are able to "learn and find new approaches" and that XP settings highly support their ambition (in a private AND professional sense). The presented HVM is able to show interrelations and correlations between different findings, which further enhances the understanding in this area of research.
- *Adapt usability and user-centred methods for agile settings*: We discussed the user centred methods "personas" and "usability reporting and ad-hoc usability input" and discussed some practical experiences in adapting them to an agile context. Our main conclusion is that these methods (except ad-hoc usability input) are too rigid in their current and classical way of use. Further, clear metrics need to be defined that communicate the progress in reference to usability. This has to apply to all of the discussed methods. As there is little know-how available so far on how usability methods need to be extended and adjusted for agile teams, with this work we lay the basis for further research in the area of usability in agile team settings by suggesting concrete and necessary enhancement of user centred methods and their positioning in agile teams.

We are aware that our current observations of the applicability and of the added value of our proposed adaptations of usability methods rely on qualitative observations and qualitative feedback of programmers. As future work we are aiming at measuring the impact of these adapted usability methods in a quantitative way.

5. References

Bjørnar Tessem, Frank Maurer, Job Satisfaction and Motivation in a Large Agile Team in Agile Processes in Software Engineering and Extreme Programming, (Page 54- 61), 2007; DOI - 10.1007/978-3-540-73101-6_8; http://www.springerlink.com/content/gh18347t4172k769

Constantine, L. L., & Lockwood, L. A. D. (2003). Usage-centered software engineering: an agile approach to integrating users, user interfaces, and usability into software engineering practice. In ICSE 2003: Proceedings of the 25th International Conference on Software Engineering (pp. 746-747). Los Alamitos: IEEE Computer Society.

Ferreira, J., Noble, J., & Biddle, R. (2007). Agile development iterations and UI design. In: Agile 2007 (pp. 50-58). Los Alamitos: IEEE Computer Society.

Göransson, B., Gulliksen, J., & Boivie, I. (2003). The usability design process – integrating user-centered systems design in the software development process. Software Process: Improvement and Practice, 8(2), 111–131.

Grunert G. Klaus; Grunert C. Suzanne; Measuring subjective meaning structures by the laddering method: Theretical considerations and methodological problems; International Journal of Research in Marketing 12, 209 – 225, 1995

Gutman, Jonathan; Means-End Chains as Goal Hierarchies; Psychology & Marketing, Vol. 14(6): 545-560, 1997

Holzinger, A. (2005). Usability Engineering for Software Developers. Communications of the ACM 48(1), 71-74.

Holzinger, A., Errath, M., Searle, G., Thurnher, B., & Slany, W. (2005). From extreme programming and usability engineering to extreme usability in software engineering education (XP+UE→XU). In COMPSAC 2005: Vol. 2. Proceedings of the 29th Annual International Computer Software and Applications Conference (pp. 169-172). Los Alamitos: IEEE Computer Society.

Hudson, W. (2005). A tale of two tutorials: a cognitive approach to interactive system design and interaction design meets agility. Interactions 12(1), 49-51.

Kent Beck, Extreme programming explained: embrace change, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999

Law, Amy; Charron, Raylene; Effects of Agile Practices on Social Factors; Human and Social Factors of Software Engineering (HSSE) May 16, 2005, St. Louis, Missouri, USA

McInerney, P., & Maurer, F. (2005). UCD in agile projects: dream team or odd couple? Interactions 12(6), 19–23.

Miller, Lynn and Sy, Desiree, Agile user experience SIG, in: CHI '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems, Boston, MA, USA, pages 2751--2754, ACM, 2009

Norman, D. A. (2006). Logic versus usage: the case for activity-centered design. Interactions 13(6), 45-ff.

Pruitt, J. and Adlin, T. 2005 The Persona Lifecycle: Keeping People in Mind Throughout Product Design (The Morgan Kaufmann Series in Interactive Technologies). Morgan Kaufmann Publishers Inc.

Reynolds, Thomas J. and Olson, Jerry C.; Lawrence, Understanding Consumer Decision Making, The means-end approach to marketing and advertising strategy; Erlbaum Associates, 2001

Rokeach, Milton, The nature of human values. New York, Free Press

T. Dyba˚ , T. Dingsøyr, Empirical studies of agile software development: A systematic review, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.01.006

Whitworth, E. and Biddle, R.; The Social Nature of Agile Teams; Agile, 2007, AGILE‘07, Conference, (26-36), 2007

Wolkerstorer, P.; Tscheligi, M.; Sefelin, R.; Milchrahm, H.; Hussain, Z.; Lechner, M.; Shahzad, S.; Probing an agile usability process; in: CHI '08: Proceedings of the 26th international conference extended abstracts on Human factors in computing systems, Florence, Italy, pages 2151--2158, ACM, 2008

A logical mind, not a programming mind: Psychology of a professional end-user

Alan F. Blackwell

Computer Laboratory
Cambridge University
Alan.Blackwell@cl.cam.ac.uk

Cecily Morrison

Engineering Design Centre
Cambridge University
cpm38@cam.ac.uk

Keywords: POP-I.A. social organisation and work, POP-I.C. health records POP-II.A. end-users

Abstract

This paper reports a case study of a specific end-user programming context, in which an electronic patient record system was being customised by a healthcare professional. Our research involved making an unusual intervention, employing a professional programmer as a quasi-experimental participant, in order to explore and contrast the different ways that the same situation was conceived by an end-user programmer and by a professional programmer. We found a range of pragmatic strategies that were employed by the end-user, causing her to resist some conventional views of how programs and source code should be interpreted. Rather than different 'cognitive styles', we believe these differing mental models can be accounted for by the context of the practical work the two need to achieve, and the organisational contexts within which they work. We make some observations and recommendations about the design of tools for end-user programmers, extrapolating from our in-depth observation of one particular product.

1. Introduction

This research investigates a context in which a complex software product is expected to be customisable by (some of) its users. The customisation facilities offered by the product include a range of scripting and programming facilities, some quite sophisticated. Although the product supplier intends that end-users should be able to use most of the customisation facilities, in practice some of these facilities would be more familiar to professional programmers. We present a case study in which we compare the experience of a non-programmer end-user who is responsible for local customisation of this product, with the experience of a professional programmer whom we employed temporarily in order to carry out our research. The interaction between the two can be contrasted with the types of situation studied by Segal (2005), in which scientific end-user programmers interact with professional software engineers. In Segal's research, the scientists being studied are confident programmers, but faced challenges when asked to integrate their work practices into more structured software engineering processes. In our research, it is the end-user who is most engaged with structures of professional practice (in this case, healthcare rather than engineering), while not being familiar with programming language concepts. In her own phrase, she has 'a logical mind, not a programming mind'.

From a psychology of programming perspective, it is well understood that end-user programmers and professionally-trained programmers are likely to have different mental models of programming tools. If that is understood, then one might expect that interaction between the two will reveal systematic challenges. In this paper we are interested both in the pragmatic consequences of those challenges – what can be done to assist such collaborations – and also in the opportunity to understand more about the mental models on both sides, by analysing the content of the conversation.

2. The Case-Study Context

We conducted this research in the Intensive Care Unit (ICU) of a specialist cardio-thoracic hospital, over a period of three years from the summer of 2006 until 2009. At the start of this period, the director of the ICU had agreed a contract to purchase and deploy an electronic patient record system (EPR). In a research-active hospital such as this one, new clinical initiatives are routinely accompanied by a research agenda. The ICU director therefore approached the University of Cambridge to identify researchers likely to have an interest in EPR deployment, and a research team was assembled via the Crucible network for research in interdisciplinary design. That team included researchers in Management Information Systems, Social Psychology, Anthropology and Computer Science. A number of studies have been conducted during the study period, but here we describe only one of these, conducted primarily by the two authors.

The EPR that had been selected after a competitive tender was MetaVision ICU, a product from Israeli company IMDsoft. Among the selection criteria applied by the ICU director, features supporting end-user customisation were a high priority. The shortlisted candidates had included another EPR product that had previously come to the attention of the End-User Software Engineering research community (Orrick 2006), meaning that end-user programming was identified from the outset as a research question of potential interest.

Over the following months, we observed the arrangements for deployment and commissioning of the product, the product training conducted by IMDsoft personnel, the customisation of the standard product to fit local practices, the operational training of the ICU nurses and clinicians, and the transition from paper records in the ICU to full reliance on the EPR. We also continued to observe over the next year as the system 'bedded-in', and became integrated into the ICU operation. In the third year, we observed ongoing maintenance and customisation work as an aspect of the routine operation of an ICU-based EPR. We also participated in convening a user group of other UK hospitals that were deploying MetaVision. The user group meets biannually in Cambridge, and provides an opportunity to compare the experiences of other hospitals to the one in which our case study has been located.

3. Programming Intervention

In the final six months of this study, we decided to experiment with an unconventional research technique, in order to gain a new perspective on end-user programming. By this stage, two members of the ICU staff had assumed primary responsibility for ongoing customisation and systems management of the EPR. The ICU director, a consultant anaesthetist, takes responsibility for medical customisation, while the EPR manager takes responsibility for customisations related to nursing, and for systems management. The EPR manager, C., had by this time become familiar with the limitations of the product, and with the constraints that it placed on the kinds of customisation she could achieve. In some cases, problems could be solved with assistance from IMDsoft technical support staff (although often remotely, from Israel). In other cases, it would have been possible to commission product customisation work from IMDsoft, but this was an expensive option, not anticipated in the ICU operating budget, and had only been done once since the EPR had been deployed.

We therefore proposed that we should employ a professional programmer for a short period of time, to provide an opportunity for a new research study. We advertised in a local forum for freelance programmers, and recruited J., a programmer with broad experience of Visual Basic-like scripting within a database environment, of the kind that the MetaVision system employed. We intentionally recruited a programmer without prior experience of MetaVision, of ICU practices, or of clinical computing more generally. The reason for this was that we wished to observe, record and analyse all conversations between J. and C., as a situation where each of them would need to make their understanding of the system very explicit to the other. We saw this as an opportunity to contrast the mental model of an experienced end-user programmer with that of an experienced professional programmer. We recorded all conversations between the two participants, and used those verbal protocols as our primary source material for analysis. The main purpose of our research intervention was thus to set up a situation in which we could elicit descriptions of mental models in a natural manner, rather than through think-aloud protocols. This research method is intended as a variant of 'constructive interaction' – a dialogue-based elicitation technique (Miyake 1986) that has previously been applied with some success in HCI (Kahler, Muller & Kensing 2000)

Our research intervention, in addition to eliciting dialogue for later analysis, also provided us with an opportunity to study a realistic professional situation. It is often the case that end-user programmers, when faced with a specific technical problem, or a design task that is outside the scope of their experience, will seek assistance from a more experienced or technically knowledgeable professional. We believed that it was important to gain more understanding of this kind of situation, and that observing a specific and well-defined task over a defined period of time would provide a reasonably well controlled opportunity for doing so.

The specific problem indentified by C. as a focus for this intervention was the creation of a more versatile reporting mechanism by which a paper report could be created with information about a specific patient. MetaVision has many standard reporting facilities, all of which are customisable, but it had proven impossible to create a particular report incorporating historical information on a single patient in a readable format. (In a previous publication (Morrison & Blackwell 2009), we have described the extent to which the ICU regularly customised its paper forms). The one piece of customisation work commissioned from IMDsoft in the past had, in fact, been the development of a particular customised report. However, the customisation of reports is a sufficiently common activity that it was not desirable to have this amount of expenditure every time a report was customised. C. had been disappointed, after that earlier experience, with the fact that the custom code created by IMDsoft was not delivered in a form that she was able to further modify it herself. The practical motivation for employing J. was therefore to observe the report customisation process, and if possible, learn to do it herself in future. This meant that there were two practical objectives – one short-term, in which J. might create a new customised report, and one long-term, in which C. might learn to create such reports herself in future. There is clearly a degree of conflict between these. On reviewing a draft of this paper, J. wished to emphasise that he could easily have achieved the first goal, and that it was the introduction of the second that gave rise to many of the results we report. This is precisely the dynamic that we consider to be interesting. As noted, C. had already commissioned a programmer to carry out system customisation, but now wanted to learn how to achieve the same results herself. We believe that this experimental situation, although to some extent manufactured through our intervention, is also typical of end-user programming experiences.

Our research budget provided resources to employ J. for a total of five days. Approximately 40% of this time was spent away from the ICU, carrying out background research into the MetaVision features and architecture, technical preparation and some exploration of potential approaches. The remainder of the time was spent in C's office at the ICU. The second author observed all of the periods in which C and J worked together, and recorded all conversations for later transcription. Both authors observed the final day of work, at which the results were reviewed and 'handed over'. At the close of the day, C and J were interviewed together, reflecting on their experience of the project. As before, these conversations were recorded and transcribed.

After transcription of all the above conversations, the two authors independently coded the transcripts, and then jointly reviewed those codes and their interpretations. An initial analysis of the project results has been prepared for a clinical audience (Morrison et.al., in press). The draft manuscript presenting results of that analysis was reviewed by C., and each author discussed with her the interpretations that we had drawn, modifying any misinterpretation or misunderstanding. The remainder of this paper presents the findings of our analysis as they relate to end-user programming more generally, rather than the specific clinical context. We also draw on observations made through the previous years of our case study, and on field recordings and notes by both of us, where necessary. As with our earlier report for a clinical audience, this paper has also been reviewed in manuscript by C. and by J. Both of them responded with detailed comments (C. returned a marked up manuscript), and we have taken those comments into account in our analysis. In the following text, we specifically indicate those situations in which we had initially drawn a particular interpretation, but one or both of them asked us to correct it.

4. Findings

4.1. Organisational context

As reported in more detail in (Morrison et.al., in press), this study revealed unanticipated aspects of the role played by end-user programmers within an organisation. Our observations reinforce the importance of the 'EUSES' view as advocated by Ko et al. (in press), which is that end-users must engage in all aspects of end-user software engineering, rather than simply end-user programming. However, the software engineering factors that arose in our research suggest some alternative emphases to those described in the survey by Ko et al.

The main theme of our findings in this area could be summarised as 'end-users have users too'. Despite not being professionally trained as a programmer, C. has assumed responsibility within the ICU for operation, maintenance and customisation of the EPR. Because she is the person with the most technical knowledge of the system, and because the system is known to be customisable (and to have been customised by her), she receives 'feature requests' from the other ICU staff, with proposals or recommendations for further customisation. In responding to those requests, she must not only consider programming work, but also business process analysis, user interface design, and many other aspects of software engineering that in a professional software engineering team are considered to require special training and experience for people taking particular roles within the team.

Once C. has identified necessary changes, either in response to requests, or as part of her own continuing refinement and enhancement of the system, she must plan the deployment of those changes. As with many end-user customisable systems, the customisation interface is a part of the live system. However, an ICU operates on a 24-hour, 365-day basis, as with many critical business systems that have dedicated software engineering teams. In such businesses, it is normal to maintain three parallel systems: one 'production' system that business users actually interact with, one 'test' system configured in the same way as the production system, and one 'development' system that the programmers interact with. Complex versioning and configuration control is maintained between these systems, with specialist teams dedicated to each of the different systems, and to the overall build and release process. In our case study, C. was responsible for all three kinds of work, but using one system for all of them, and with few tools for version management and configuration control. As noted in our earlier publication for the clinical audience, this situation is far from ideal, and ought to be addressed in future tools for end-user software engineers.

Furthermore, it should not be assumed that the same tools used for these purposes in professional software development organisations will also be appropriate in an end-user software engineering context such as this. C. herself, despite having users of her own, does not regard herself as a professional software engineer. When she engages with IMDsoft support staff, or in her collaboration with J., she approaches her work very much in the manner of an end-user, with user concerns her key focus. This places her constantly in a kind of border-land, where none of the people she interacts with are exactly peers. This is another respect in which she is unlike professional software developers, who usually work within teams of professional peers sharing their own assumptions and practices. At many points in our observation, it was clear to us that C. did not share the assumptions and practices advocated by J., and had no desire to do so. For a new software engineering tool to be of value to her, it should be possible for her to adapt that tool to her particular working practices, rather than requiring her to abandon them.

4.2. Mental models and motivation

In analysing the conversation between C. and J., we were particularly interested to identify differences in the way that they conceived of programming tools and the programming task. This aspect of our findings is the one that is most clearly related to established research concerns in psychology of programming.

We were impressed by how actively C. resisted the conventional programming conceptions of system behaviour. She described large parts of the system behaviour as involving technical facilities that were 'in the background'. Most of the operation of the underlying database management system, for example, fell into this category. She was aware of certain behaviours of this system, for example in carrying out data integrity checks, but did not want to be distracted by explanations or investigation of those behaviours. She clearly regarded these as important, but as lying within the domain of responsibility for IMDsoft. It would not be a good use of her own time to learn about aspects of the system that other people will be responsible for. We found that this highly pragmatic attitude of a busy and responsible person was in contrast to a typical technical attitude, occasionally expressed by J., which is that aspects of system behaviour are interesting in themselves, and that it might always be useful to know a little more about them for future use. C. regularly referred to these aspects as being 'esoteric' interests.

In part, this distinction is encouraged by IMDsoft, whose product documentation draws a clear distinction between the system model that must be understood in order to do customisation work, and the underlying technical behaviour of the system. As a company with long experience working with a particular community of end-user developers, their design approach seems to be an appropriate one. In the psychology of programming community, this has been described as 'the black box inside the glass box' (du Boulay, O'Shea & Monk 1981). However the work of du Boulay et al. was oriented toward the teaching of conventional programming languages. In this case, the end-users do not want or need to know about a conventional programming model, but a customisation model. For the purposes of tool design, the glass box must provide a virtual machine suitable for customisation work – but there are challenges in achieving this in the MetaVision product, as we discuss below.

C. is quite willing to spend time learning about aspects of the system that may not be directly relevant to her current task. In this respect, she is not completely captive to the 'paradox of the active user' (Carroll & Rosson 1987), and is able to make attention investment decisions with longer term payback (the attention investment model, previously presented at PPIG, describes the way in which some people choose whether to construct abstract/programming models of a task based on assessment of future payback – see Blackwell& Green 1999, Blackwell 2002). C described a conscious strategy of finding opportunities to make basic modifications in an area where she hadn't worked before, in order to open up possibilities for future enhancements, or understand the potential for new uses of the system capabilities within the ICU. However, these are very specific decisions, such that the various factors involved in the attention investment model become partitioned between areas of the system functionality. There are some aspects of the system where C. has done a significant amount of 'tinkering' (Beckwith et.al. 2006), and as a result has developed a high degree of self-efficacy with regard to that particular aspect. But at the same time, there are areas of system functionality that she avoids, and where she was not confident to make changes or 'tinker'. In terms of the early 'garden shed' analogy for tinkering behaviour (Blackwell 2006), C. is happy tinkering in certain parts of the shed, but there are others that she avoids.

Perhaps the most distinctive aspect we observed of the approach taken by C. to the end-user customisation tools was the way that she approached reading of the source code itself. A number of the MetaVision facilities (some report generation tools, a 'query wizard', event trigger rules and so on) use relatively conventional programming language syntax, generally quite closely related to Visual Basic. After several years of use, C. had become perfectly accustomed to reading, modifying and adapting chunks of this code, but the way in which she did so was quite different to J's reading of the same chunks. Whenever we observed the two of them reading and discussing an extract of source code, it seemed that any given word of text that was considered interesting by C. would not be considered interesting to J. and vice versa.

When reading source code, C. generally found the identifier names meaningful, and treated them as most likely to reveal the function of that code segment, and offer clues on how to modify or customise it. She did not generally comment on programming language keywords, control constructs, type declarations or syntax elements. Our impression was that these aspects of the code were relatively uninteresting, were not perceptually salient, and perhaps even annoying distractions. At one point where J. drew attention to them, she commented that these were typical of his 'esoteric' interests. J., on the other hand, like most professional programmers, was adept at reading past the (technically speaking, arbitrary) names chosen for identifiers, in order to read the underlying structure and behaviour of the code. In his reading, it was the keywords, control constructs, syntax and so on that were the important topics of conversation.

Of course we are not suggesting that either C. or J. are unaware of the importance of those aspects of the source code that the other found more salient. In particular, the way that we had set up our research intervention had left J. with a partial impression that we wished to study him ‘teaching’ C. more about programming (we address this further below). This may have caused him to emphasis formal features of the language in discussion precisely because he was concerned that she had been paying insufficient attention to them. Nevertheless, we believe that the design of programming tools for end-user programmers may have paid insufficient attention to their need to focus on their own domain as the primary feature of the program representation.

There is a further possibility that these different orientations toward the code arise from applying different personal styles within the professional context, of the kind reported by Wray (2007) and Hudson (2009). As might be expected from their respective professional backgrounds, we found that C. was more likely to describe the functions of the system in terms of people, and in terms of particular roles that people play within the operation of the ICU. In contrast, J. was more likely to describe the system functions from an abstract perspective, describing not only technical features, but even the illustrative examples he chose in terms of data ontologies and abstract functions. These alternate perspectives are completely typical of the end-user versus programmer orientation, but they also demonstrate a contrast between characteristically empathising and systemising styles, as observed by Wray and Hudson. Furthermore, they exhibit a contrast between human/social and computational thinking, as discussed by Blackwell, Church and Green (2008). As C. stated when reflecting on her own cognitive style, this experiment has led her to believe that she has ‘a logical mind, but not a programming mind’.

All of these factors lead C. to regard programming, and investing time in learning about programming, as an activity that although it has extrinsic benefits, does not have intrinsic ones (Nash, Church & Blackwell, submitted). She described herself as becoming ‘impatient’ in the course of the collaboration with J., because he persistently discussed topics that appeared to her not to be relevant to the work at hand. To some degree, this resulted from the central concern of the study, which was the extent to which C. might be able to learn how to do work of this kind in future. Although this had been an objective from the outset, time spent discussing ‘educational’ topics was also perceived by C. as time in which she was not making progress toward the immediate goal of creating the report. As a result, J’s priorities appeared to her to be motivated by an attitude to programming as having some intrinsic satisfaction, independent of the results achieved. This contrast between the priorities of technical and non-technical users has been a common theme in past work that has emphasised the importance of providing tools for end-users that offer visible progress toward achieving the user’s own goals, rather than teaching abstract principles with no clear relationship to user priorities.

4.3. The collaborative relationship

Although the collaboration that we set up between C. and J. was in many ways an artificial one (created explicitly as a component of a research project, with an explicit research agenda, paid for by research funding), we believe that the conversations were quite typical of settings in which short-term collaborations occur between end-user and professional programmers. Furthermore, in those settings, the quality of the collaborative relationship is of interest because it can have direct commercial value. We therefore analyse this relationship more closely.

As noted in our description of the way in which the study was set up, there was a fundamental tension between the immediate goal of creating a customised report, and the longer-term goal in which C. would learn how to create such customised reports herself. The first goal required J. to carry out conventional professional programming work, while the second required him to act in the capacity of a teacher. As he observed when reviewing a draft of this paper, he does not have teacher training or experience, so we could not have expected that he would find this aspect of the project easy. We agree that it may be unrealistic to expect professional programmers also to be professional teachers of programming. Nevertheless, this situation is one that arises relatively frequently in the experiences of end-user programmers. As end-users, they have often not received professional training in programming. Instead, they acquire knowledge from a variety of sources, including programmers such as J. who have relevant technical knowledge but no teaching experience.

Although almost certainly exacerbated by the artificial nature of the research project, we observed a significant degree of discomfort in the collaborative relationship between C. and J. Based on the discussion in the previous section, we believe that this arises to a large extent from the different psychological styles that are characteristic of 'programming' work and 'user' work. As described by Bucciarelli (1994), technical professionals work within an 'object world' rather than one of primarily human relations. However in professional software development organisations, there are a number of specific technical roles dedicated to bridging these two worlds, such as systems analysts, technical authors, helpdesk operators and so on. Each of these professions has an established culture, tools, methods and practices that allow them routinely to carry out this bridging work. In the case of an end-user developer such as C., her professional role does not include the cultural repertoire of tools and methods that support easy interaction with professional programmers such as J.

For his part, J. accepted his responsibility in part as an educational one, providing C. with valuable skills that would enable her to be more effective when carrying out future programming work. Given that the two had comparable professional seniority, this educational objective often amounted to a clash of alternative intellectual styles. In the educational approach taken by J., the future development needs described by C. could be achieved more effectively by adopting more abstract analysis and problem-solving strategies – strategies that are completely characteristic of programming work. However, as described in the Attention Investment model (Blackwell 2002), abstract strategies can be expensive ones, requiring greater initial investment of attention, with potentially low return, or even negative return. When pursuing the educational objectives in his collaborative relationship with C., J. was therefore acting as an advocate of a more abstract strategy, attempting to persuade her that she should adjust the attention investment decision factors that she already applied. In many of their conversations, C. resisted this advice, saying that she would prefer to make progress on the specific, concrete, problem that had motivated the project in the first place. For his part, J. tried to persuade her that it would be in her long-term interest to address not just this particular problem, but the general category of problems of this type, or indeed the general capabilities to be acquired by gaining experience of the most general capabilities of the programming language.

In this last respect, the conversation between the two became a tutorial discourse, in which J. spent time explaining the fundamental concepts of programming languages, such as classes and functions. In doing so, he used the same kinds of example that are routinely presented in programming language textbooks. To explain the notion of class type and aggregate relationships, he talked about the parts of a car, and to explain functions, he constructed an example based on financial calculations. Although these examples may have been effective tutorial illustrations of the necessary abstract principles, C. was not necessarily motivated to acquire those abstract principles in the first place. The fact that the tutorial examples were so far removed from her own problem domain exacerbated her impression that formal aspects of programming were esoteric and irrelevant, and fuelled her impatience with the collaboration as a whole.

After reviewing a draft of this paper, J. explained that he would certainly have preferred to use teaching examples from the ICU itself, but felt that this would be risky, given the very limited time that he had to become familiar with that domain. Although both collaborators felt frustrated by the constraints arising from the short-term nature of our study, it is also typical of end-user development dynamics, where it is always the case that the end-user is the person with the most complete understanding of their own domain, and professional programming assistance, when available, will come from people with less understanding of the end-user's own work. However, even where J. explored a particular approach in terms of C's own domain tasks by re-implementing an existing function (a common programming strategy), C. commented in her review of our draft paper that this had seemed especially unproductive, when the time could have been spent implementing new functionality.

Our study is open to critique, regarding the extent to which this collaborative relationship can be taken as generally representative of end-user programming experiences. Although C. has extensive responsibility for customisation, her experience specifically of programming is relatively limited. This experiment, casting her in the role of a potential end-user programmer, was therefore not completely typical of her work. J. also has extensive experience of working within client organisations, alongside end-users, but has not in the past been expected either to teach programming skills to those end-users, or to create productive collaborative relationships in such a short period of time. He commented that his typical contracts would include time to learn about the environment before technical work starts. Both entered into this experiment with an open attitude to collaboration, and it is likely that the challenges they encountered in working together arose in part from the experimental constraints that we had placed on them. Nevertheless, we believe that the experiment has been more illuminating of actual professional practice than most laboratory studies, and that the combination of skills and experience that we encountered within this specific professional situation is one that is often encountered in real end-user programming contexts.

5. Implications for design

In earlier sections of this paper, we have noted aspects of the MetaVision tool that could be improved to support C's working environment and work processes. In a number of cases, such as the need for change control and configuration management, these provide evidence for the importance of the end-user software engineering perspective. End-user programmers, if working in a professional environment, are very likely to need software engineering facilities like these. Failure to provide them could be taken as a lack of respect for the professionalism of MetaVision customers, because MetaVision's own software engineers would certainly not be prepared to work without such facilities. Many software companies are concerned to avoid this kind of attitude among their developers, which has led to the practice described as 'eating your own dog food', where developers are required to use their own products. In addition to the lack of change control features, we observed a number of annoying deficits in the MetaVision customisation tools that would probably not have been considered acceptable in professional development tools, such as screen space restrictions meaning that it was not possible to view a complete form layout at the same time as running the form customisation tool. We recommend that companies creating end-user programming tools should regularly use those tools, rigorously comparing the features and limitations to those of the tools used by the company itself.

We observed a number of issues related to the use of names in the MetaVision product. As discussed above, it is identifier names that C. finds particularly salient when viewing source code of customisation scripts. This suggests that the choice of identifier names is particularly important. Unfortunately, customisable products with database back ends seem often to impose restrictions on the choice of identifier names, and make it difficult to change these names, where they correspond to aspects of the underlying database schema. We have previously noted the problems this caused during the deployment of a student record system in Cambridge (Blackwell, Church Green 2008). In the case of MetaVision, some identifier names can be altered when the system is first installed, but become embedded through scripts over time, such that they are very difficult to change or modify. It might be sensible to avoid this kind of premature commitment, by separating identifier names from the behavioural relations they describe, so that they can be modified if necessary without breaking the system.

We also observed some problems with the names of built-in functions, although these may have arisen in part from the fact that the product was developed by engineers who were not native English speakers. One example is that in MetaVision, the table of site-specific data values, which can be extended and used in scripts and forms by the end-user, are described as 'parameters'. In terms of the design model of MetaVision's own engineers, this no doubt refers to the aspects of the system behaviour that can be 'parameterised' – customised at each customer site. However, the term 'parameter' is confusing to MetaVision users. To users without previous programming experience, the word 'parameter' is relatively meaningless, and hence unhelpful. To end-users who do have previous programming experience, the word is confusing, because in their view of the system, these values do not act as parameters. In the collaboration between C. and J., the word ‘parameter’ was a source of confusion, because when J. was assuming his educational stance, he tried to describe the conventional definition of a parameter in the context of functions and classes, but these explanations did not correspond to C’s experience of the product. In order to avoid problems of this kind, we therefore recommend that end-user development products should not use existing programming language terms unless they are implementing a feature that has precisely the expected technical meaning.

As might be expected in a project of this kind, we noted a number of relatively routine usability problems with various aspects of the MetaVision system. Some might be addressed by using techniques borrowed from other end-user programming research – for example, the scripting language used for event processing looked as though it would have been easier to use if it had presented a publish/subscribe event model such as that used in the Scratch language, and the report scripts could have benefited from a better way of defining layout constraints subject to varying amounts of data on the page. Others appeared to have more fundamental problems, as in the case of the Query Wizard, which all MetaVision users appear to find problematic. In that case, it was interesting for us to observe the workarounds that C. had adopted to gain some value from the tool, using the wizard to generate samples of script code that she then cut and pasted into other parts of the system, making minor adaptations if necessary. This generate-and-paste approach, although somewhat clumsy by comparison to the intended operation of the Wizard, was in practice both pragmatic and empowering.

In the course of observing the collaboration between C. and J., we also saw some unexpected and positive uses of tools. When explaining the operation of Visual Basic within a Word macro, J. demonstrated the code execution by following it in the VB debugger. C. had not seen this view of code before, and found it very compelling as a visualisation of the system behaviour. We suspect that a simple script debugger could do much to assist more sophisticated end-user developers. In combination with the Wizard strategy, one might imagine an approach to scripting in which candidate scripts are generated, and their operation is animated, as an alternative to programming approaches in which code is generated from scratch, relying on an execution model that the programmer is supposed to have learned and internalised before starting work.

6. Conclusions

We have presented findings from a field study of a real customisation project that involved collaboration between a professional programmer and an end-user programmer. The context in which they were working drew attention to the ways in which end-user programmers working within an organisational context can have many of the same responsibilities as professional programmers (for example, they have users too), yet without access to the same tools that professional programmers take for granted. We observed a number of differences between the approaches taken by the two participants in our study, providing rich illustrations that tend to confirm previous research observations regarding the challenges of end-user programming relative to professional programming. However, we also observed a number of ways in which the collaboration itself is problematic, as a result of significant differences in mental models and in strategic approaches to technical problems. We believe that collaborations of this kind are relatively frequent, though little studied, and that further research is justified. Finally, we have developed a research method that is relatively novel in psychology of programming research, involving a funded intervention in an organisational context, with ethnographic observation and analysis of the resulting collaboration.

7. Acknowledgements

This research was funded by Boeing Corporation. We are grateful to our collaborators at the ICU described in this study, including ICU director Alain Vuylsteke who initially invited us to study the deployment of the MetaVision system, and especially C. and J. who have been both patient and tolerant to a generous degree.

8. References

Beckwith, L., Kissinger, C., Burnett, B., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. In *Proceedings of CHI 2006,* pp. 231-240.

Blackwell, A.F. & Green, T.R.G. (1999). Investment of Attention as an Analytic Approach to Cognitive Dimensions. In T. Green, R. Abdullah & P. Brna (Eds.) *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11),* pp. 24-35.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.

Blackwell, A.F. (2006). Gender in domestic programming: From bricolage to séances d'essayage. Presentation at *CHI Workshop on End User Software Engineering*.

Blackwell, A.F., Church, L. and Green, T.R.G. (2008). The abstract is 'an enemy': Alternative perspectives to computational thinking. In *Proceedings PPIG'08, 20th annual workshop of the Psychology of Programming Interest Group*, pp. 34-43.

du Boulay, J.B.H., O'Shea, T. and Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man Machine Studies*, 14:237-249.

Bucciarelli, L.L., (1994). *Designing Engineers*. MIT Press.

Carroll, J.M. and Rosson, M.B. (1987). Paradox of the active user. In: J.M. Carroll, Editor, Interfacing Thought: Cognitive Aspects of Human–Computer Interaction, Bradford Books/MIT Press, pp. 80–111.

Hudson, W. (2009). Reduced empathizing skills increase challenges for user-centered design. In *Proceedings of the 27th international Conference on Human Factors in Computing Systems (CHI'09),* pp. 1327-1330.

Kahler, H., Muller, M., Kensing, F. (2000). Methods & tools: constructive interaction and collaborative work: introducing a method for testing collaborative systems. *Interactions* **7**, 27-34.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (in press). The State of the Art in End-User Software Engineering. Accepted for publication in *ACM Computing Surveys*.

Miyake,N. (1986). Constructive interaction and the iterative process of understanding. *Cognitive Science* **10**, 151-177.

Morrison, C & Blackwell, A.F. (2009) Observing end-user customisation of electronic patient records. In V. Pipek, M.-B. Rosson, B. de Ruyter and V. Wulf (Eds). *Proc. 2nd International Symposium on End-User Development, IS-EUD'09*. Springer Verlag (Lecture Notes in Computer Science - LNCS 5435), pp. 275-284.

Morrison, C., Blackwell, A. and Vuylsteke, A. (in press) Practitioner-Customizable Clinical Information Systems: a case-study to ground further research and development opportunities. Accepted for publication in *Journal of Clinical Engineering*.

Nash, C., Church, L. And Blackwell, A.F. (submitted). Designing Computational Tools for Creativity, Flow and Mastery. Paper submitted to *VL/HCC 2010*.

Orrick, E. (2006). Position Paper for the CHI 2006 Workshop on End-User Software Engineering. Available online from http://eusesconsortium.org/weuseii/docs/ErikaOrrick.pdf

Segal J. (2005). When software engineers met research scientists: a case study. *Empirical Software Engineering* **10**(4), 517-536.

Wray, S. (2007). SQ Minus EQ can Predict Programming Aptitude. In Proceedings PPIG'07, *10th annual workshop of the Psychology of Programming Interest Gro*up, pp. 243-254.

Empirically-Observed End-User Programming Behaviors in Yahoo! Pipes

Matthew D. Dinmore

Applied Information Sciences Department
Johns Hopkins University
Applied Physics Laboratory
Matthew.Dinmore@jhuapl.edu

C. Curtis Boylls

Applied Information Sciences Department
Johns Hopkins University
Applied Physics Laboratory
Curt.Boylls@jhuapl.edu

Keywords: end-users, visual languages, mental models, phenomenology

Abstract

Yahoo! Pipes is a well-known, widely used visual programming environment for creating data mashups by aggregating, manipulating, and publishing web feeds. It provides a natural laboratory for observing a range of end-user programming (EUP) behaviors on a large scale. We have examined more than 30,000 Pipes compositions in a search for regularities that might inform the design of EUP systems and their services. Although Pipes primitives span a broad range of functionality and can be richly parameterized and composed, we find a number of patterns that govern the structure and parameterization of Pipes in the wild. Most users sample only a tiny fraction of the available design space, and simple models describe their composition behaviors. Our findings are consistent with the idea that users attempt to minimize the degrees of freedom associated with a composition as it is built and used.

1. Introduction

Research focused on user behavior in end-user programming (EUP) has typically been conducted through observation of the activities of a few users in a relatively small environment, for example a workgroup within an organization or laboratory experiment (Myers, et. al., 2006). The growth of the Internet and EUP tools for it now offers a large-scale, open-access, live environment in which it is possible to observe many end users at work, albeit primarily through the artifacts they create rather than observation of their direct interactions with the EUP tool.

Here we report preliminary results from an empirical examination of artifacts created by users of the Yahoo! Pipes web-based visual programming environment. Our research questions are motivated by our interest in understanding users' compositional behaviors, and in particular, how they make use of the collection of primitives provided by the environment, whether this collection adequately covers the range of needs they attempt to satisfy with their compositions, and how they manage complexity. This paper proceeds as follows: after a brief review of the literature, we describe the Pipes environment, the measurements that can be derived from it, and our rationale for selecting it for this study. We then detail our data collection and preparation methods. Using the resulting data corpus, we next examine the question of how users who create pipes (end-user programmers) compose their solutions given the primitives provided by the environment, and we identify and discuss several models that fit our observations.

1.1 Background

Our primary motivation for this work is a desire to understand how end users compose solutions to problems in their domain; our focus is on the collection of primitives offered to the user and their behaviors in using them. From an engineering point of view, how can we model these behaviors for the purposes of either designing a set of primitives, or given alternative collections of primitives, choosing among them for a particular task? Theoretical models such as Blackwell's (2002) Attention Investment model consider this question in the context of the "cost" for a user to compose a solution in a particular environment. Can empirical observations relate measurable costs to typical behaviors?

Likewise, what are the reuse behaviors of users? It has been observed that end users tend to learn by building on the work of others (Gantt & Nardi, 1992); can an environment facilitate this? To what degree to users make use of the ability to copy and modify? If offered the opportunity, do users create hierarchical abstractions – reusable modules – for themselves and others? And, once a user has solved a problem, to what extent are parameters exposed to other users for run-time customization? This study begins to address these questions, with the focus primarily fixed on the design-time choices users make in composing their solutions.

1.2 Other Large-Scale Studies

Few large-scale empirical studies of end-user programming have been previously reported. Several studies have made uses of the EUSES spreadsheet corpus (Fisher & Rothermel, 2005) to examine the behaviors of end-user spreadsheet developers; this corpus contains 4,498 artifacts. Bogart, et. al. (2008) studied the CoScripter web scripting environment, and asked questions similar to our own about user behavior. While both CoScripter and Pipes automate web processes, CoScripter is different in that it could be described as text-based rather than visually-based, and execution of scripts occurs at the browser (client), rather than at the web server. The authors examined a corpus of 1445 unique scripts, in which they characterized user behaviors and scripting processes, to include reuse.

While Yahoo! Pipes is often cited as an example in work on web mashups, there have been few specific studies of it or its artifacts. The largest we are aware of involved an examination of the social network around Pipes by Jones and Churchill (2009), who looked at communications among users on web groups associated with Pipes. They studied the posts of over 2,000 users, categorizing the user network and the kinds of exchanges hosted on it.

2. The Yahoo! Pipes Environment

Yahoo!Pipes is a web-based, visual programming environment introduced by Yahoo! in 2007 with the intent of enabling users to "rewire the web."[1] Pipes is a dataflow system in which the data is sourced from the web (RSS feeds, web pages, raw data) and flows through an interconnected set of modules that act upon it, ultimately producing some result; the Pipes name is inspired by the concept of pipes in Unix operating systems that enabled the composition of command line sequences of Unix tools through which data “flow” for processing.

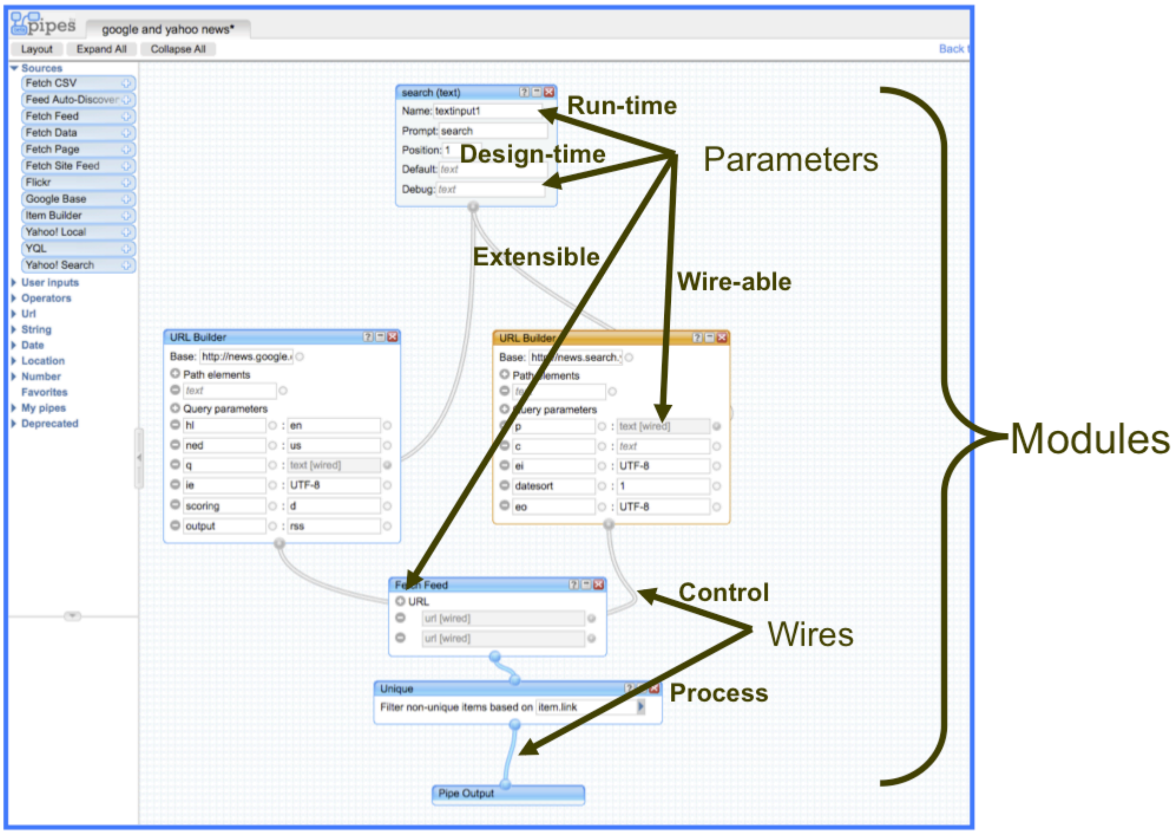
As a visual programming environment, Pipes is well suited to representing solutions to dataflow-based processing problems (Whitley, 2006), and is also quite accessible to end-user programmers. Here, we use the goals-based definition of end-user programmer—one who is creating software to solve a problem in his/her domain of expertise, as contrasted with a professional programmer who creates software for users in other domains (Ko, et. al., 2008); we suspect our data contain artifacts created by both trained and untrained programmers.

The visual programming paradigm is one of the more common end-user programming modalities (Burnett, 1999) and has been the subject of many end-user programming studies and tool development projects (Kelleher & Pausch, 2005). Pipes offers the opportunity to extend this research to incorporate the behaviors of thousands of users motivated to mash up content and services available on the web for their own purposes. The compositions that they create in this process can be examined from a number of perspectives, and doing so *en masse* offers an aggregate reflection of their approach to problem solving through programming.

Pipes consists of a publicly accessible website that allows users to find existing pipes (by browsing or searching by keyword), and to execute those pipes. Execution often involves entering runtime parameters and results in the production of a web page. Users can also edit existing pipes and create entirely new compositions. Our focus is on the structure of composed pipes, and in particular, three elements of Pipes composition over which the users have direct control: the selection of modules, the wiring of those modules, and the settings of parameters within the modules. Related to the last of

[1] http://pipes.yahoo.com/

these are also the parameters that the end-user developer exposes to users of the pipe in the form of runtime settings. Figure 1 depicts these elements as they occur in a typical pipe.



search (text)
Name: textinput1
Prompt: search
Position: 1
Default: text
Debug: text

URL Builder
Base: http://news.google.
Path elements
text
Query parameters
hl : en
ned : us
q : text [wired]
ie : UTF-8
scoring : d
output : rss

URL Builder
Base: http://news.search.
Path elements
Query parameters
p : text [wired]
c : text
ei : UTF-8
datesort : 1
eo : UTF-8

Fetch Feed
URL
url [wired]
url [wired]

Unique
Filter non-unique items based on item.link

Pipe Output

Figure 1. Annotated view of a typical pipe's elements.

A pipe is assembled from a selection of modules. Presented as visual tokens along the left side of the Pipes editor, these modules can be dragged onto the composition canvas, where their design-time parameters are presented for editing, and they can be connected to other modules to form the pipe. There are 51 primitive module types arranged in eight categories, including sources, user inputs (that present a run-time parameter to a user), and classes for manipulating strings, URLs and other types of data.

Within the pipe, the primary or "process" dataflow, which must originate in one or more source modules, is denoted with blue "wires." Pipes also contain "control" wires, colored gray, that enable parameters captured by one module to be wired to the outputs of other modules, allowing for those parameters to be programmatically set.

The parameters for each module are presented as elements in a form within the body of the module. Some offer a selection of settings (including binary on/off, radio buttons or a pick-list), while others are freeform. As noted previously, certain parameters can be wired at design-time to the outputs of other modules.

Pipes created by a user are saved in the user's account; they can also be published, which makes them visible in searches to other users (however, it is not necessary for a pipe to be published to be publicly accessible, provided its URL is known). A user can also clone another user's pipe and employ it as the starting point for developing their own; this facilitates a common "scaffolding" behavior in end-user programming (e.g. Repenning & Ioannidou, 2006; Nardi & Miller, 1990) and represents a kind of reuse.

Users also have the ability to augment the suite of primitive Pipes modules by designating an existing pipe as a "subpipe," a property that we will exploit in the analysis below. In this case, the (sub)pipe appears in the user's workspace as another module, and its runtime parameters are presented as the module's design-time parameters. Subpipes can be embedded within other pipes to any number of levels. Subpipes do have one significant limitation: they cannot accept a blue-wire dataflow as an input. It thus is not possible to create a pass-through "processing" subpipe. As a result, subpipes are typically used as specialized data sources. Because users design and employ subpipes to "extend" the repertoire of Pipes modules, we might expect to see different usage behaviors between pipes and subpipes. We thus have separately analyzed the data from each, looking for such differences and their implications.

3. The Pipes Data Corpus

3.1 Data collection

We collected the data set for this study over the 52 days between 31 December 2008 and 21 February 2009. Yahoo! Pipes does not provide direct access to the underlying "source code," nor does it offer the ability to simply list all available pipes. Instead, it is possible to browse a list of pipes presented by the interface, and one can also search by keyword. However, access to search results is limited to 1010 items. These restrictions and others necessitated a snowball-styled sampling approach that proceeded as follows: those pipes offered through the browsing mechanism were collected and their most frequent keywords extracted. These keywords were then used to initiate searches, and those pipes were collected, as well. Part of the collection process involved searching for a pipe by searching on its specific title; in addition to returning the pipe of interest, this also often resulted in other hits, which were similarly collected. Finally, some pipes included subpipes, which were subsequently collected in their own right, as well as pipes related to them. The entire process resulted in a collection of 70169 pipes.

We obtained two kinds of data about each pipe instance. First, we gathered metadata about the pipe: when it was created, the author, the title and descriptive text provided by the author, whether it had been cloned or identified as a favorite, and its run count (the number of invocations) on the date collected. All of these fields were databased for analysis by collection date; pipes were revisited to attempt to capture changes made over time, resulting in some cases in multiple records for each pipe.

We also collected the structure of the pipe by downloading its JavaScript Object Notation (JSON) file and associating it with the metadata. This structure includes all of the details necessary to reconstruct the pipe, including the modules and their placement, the wiring, and the parameter settings.

3.2 Data refinement

In our early examination of the data, we removed duplicates as well as those that had damaged JSON files. It became clear there were large numbers of replicated pipes that were structurally the same, but differed in the feeds they were creating. These proved to be targeting web pornography. As these instances biased our data with their large numbers and did not satisfy our interest in problem solving by end-user developers, we chose to exclude them from our analysis. We found that this could be done by purging pipes in Pipes accounts containing 115 or more pipes; 41 accounts representing 27115 pipes, or 661 pipes per account appeared to be in this business (again, without exception). This reduced our collection to 43054 pipe instances. For an additional 1599 instances, we found that we were unable to collect some portion of the data (typically the structural file), and these were also discarded.

We note here that we have made a decision about a class of functionality that is not of interest to us (namely mass-produced pornography-related pipes) based on the observation that the class consists largely of replicated pipes having different parameterizations. We have also not otherwise attempted to understand the semantics of the pipes that we are studying (see Discussion), so our data may include other instances of pipe replication with little novel end-user design, as well as instances in which we have retained pipes with no meaningful utility; we believe the numbers of these would be small, relative to the class of pipes we eliminated, but this nonetheless represents a limitation in our understanding of the corpus.

For the remaining 41455 instances, we databased the following data pertinent to our study:

- The unique pipe identifier (UID) assigned by Yahoo! when the pipe was created
- A self-assigned account identifier for the owner of the pipe (nominally the "author")
- The pipe configuration: modules with their identifiers, inter-module wiring, parameters
- The total number of invocations ("runs") of the pipe at the time it was collected
- Whether or not the pipe was used as a subpipe, and which other pipes "called" it
- Whether or not the pipe was published (made visible and searchable by visitors to the site)
- Whether or not the pipe was marked as having been "cloned"

We also obtained other information (e.g., the order in which users selected modules for the pipe, the location of module representations in the visual workspace, etc.) that will be useful in further investigations (see Discussion).

We quickly discovered that "raw" Pipes data include phenomena that could obscure our view into the final programming choices made by end-user authors. Most of these nuisance variables arise because Pipes is a live construction zone: authors are free to leave and return from work in progress arbitrarily, and no Pipes metadata element, including publication status, reliably indicates that a pipe is "complete." However, we can easily spot and eliminate metadata from two types of non-functional pipes that probably represent intermediate stages of construction:

a. All pipes are required to have an "output" module, so any pipe having only one module cannot be doing anything interesting. We found 86 instances in our data.

b. All pipes having N modules, but fewer than N-1 wires, cannot be connected as a single pipe. At least one module must be an "orphan." Our data contained 2656 examples.

Eliminating these two categories, our corpus shrank to 38713 instances. We also considered culling the data further by ruling out unpublished pipes (2384 instances) or pipes with few invocations, but we decided against that because the affected population is small and the possible relationship with end-user programming behavior is ambiguous.

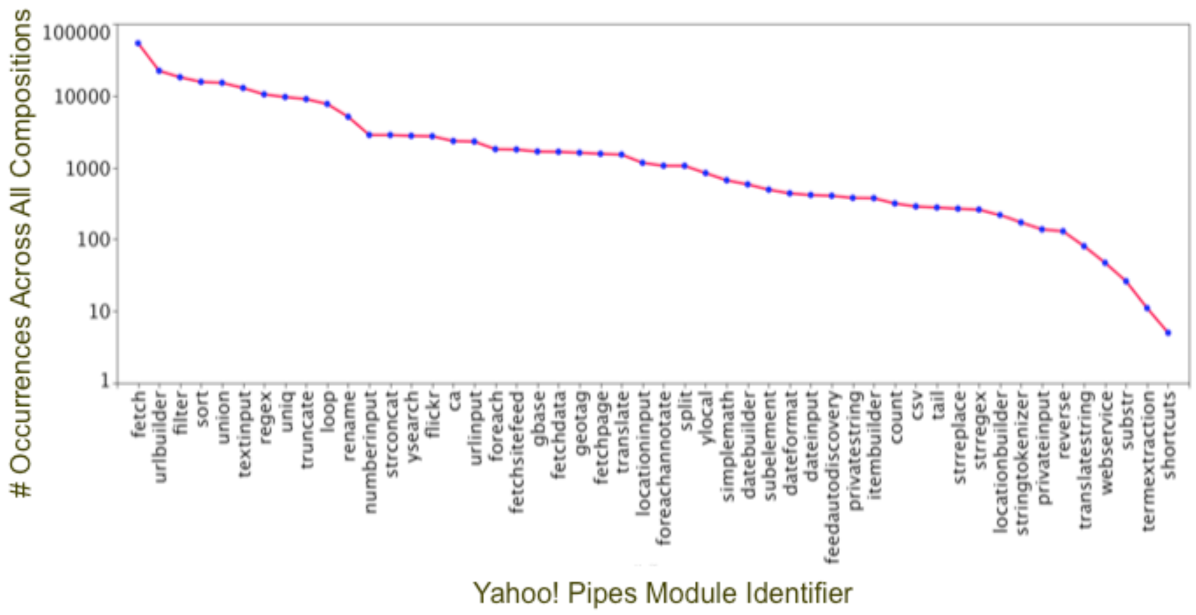


Figure 2. Primitive module usage.

We next attempted to infer something about the tasks that end-user programmers are accomplishing with Pipes. We counted the number of times that each module type was found in a Pipes composition and rank-ordered the modules accordingly. Fig. 2 plots the results of this module "popularity contest" (see section 4.3 for another use of this information). Note that the most frequently used modules (by far) are those associated with fetching and parameterizing data (*fetch, URI builder*), followed by modules that filter, transform, and merge data (*filter, sort, union*, etc.). Pipes is advertised as a vehicle for creating data mashups, and these observations are consistent with that use. In section 4.3, we will use the same "popularity" information to assess other influences on users' choices of modules for compositions.

Pipes metadata does not directly tell us anything about the actual end-users whose work is represented in the corpus. Users must create Pipes accounts to author compositions; but since multiple accounts are allowed, one user, in principle, could have created all of the data. We did, however, observe 22285[2] unique account identifiers across all instances in the corpus, an average of 1.74 instances per account. The maximum number of instances owned by one account was 110 (recall that we eliminated pipes by authors with 115 or more instances, as these were all pornographic and probably mass produced). We cannot draw strong conclusions about the size of the user population from these figures alone. But it does seem reasonable to assume that a goodly number of "real humans" account for the data, perhaps a population larger than has heretofore been the case in studies of end-user programming in the wild. Further, we made no attempt to characterize the degree of programming experience each user had, accepting that those with professional training may approach problem solving differently than those without. We do believe that that future efforts with this data, particularly with amplifying information such as that presented by Jones and Churchill (2009), could potentially categorize users based on the artifacts we can observe.

Lastly, we partitioned the corpus into metadata from pipes that were never used as subpipes (36676 instances) from those that were (2037). Recall that, once designed and published, subpipes take on the persona of a Pipes data-source module, including an analogous visual representation in the workspace. This property will allow us to examine how end-users extend the Pipes environment as meta-designers for other end-users.

4. End-User Programming Behaviors in Pipes

4.1 Choosing modules

To create a new composition in Yahoo! Pipes, an end-user must first drag modules from a menu into a workspace before wiring them together and setting module parameters. Other than requiring a single "output" module (that Pipes supplies automatically), Pipes places no limits on how many modules a user employs, and the visual workspace expands to accommodate additions. Pipes also supplies "layout" functions to regularize complex constructions.

Despite the support provided for creating large compositions, the pipes and subpipes in our corpus proved to be quite parsimonious, as shown by the descriptive statistics in Table 1. We see that, while outliers exist, the median number of modules is only 4 for pipes and 6 for subpipes, and compositions of roughly 40 or fewer modules account for more than 99% of the data.

	Pipes	Subpipes
Instances	36676	2037
Minimum modules/composition	2	2
Maximum modules/composition	177	87
Mode	3	4
Mean	6.2	7.4
Median	4	6
99.5% Quantile	33	43

Table 1. Modules in compositions.

[2] We note that Jones and Churchill (2009) report "over 90,000 developers" at the time of their study in December, 2008, suggesting that our contemporaneous sample of pipes represents roughly 1/3rd of the author population.

If we look at the empirical probability of finding compositions of length N in our collection (Fig. 3), we see that the distributions for pipes and subpipes are similar, but that subpipes are biased toward including more modules (see also Table 1).

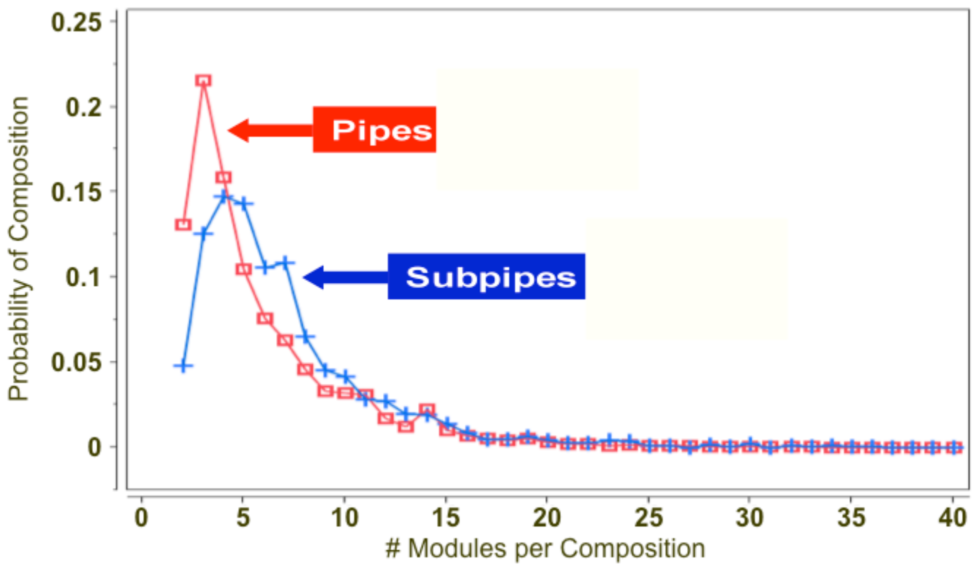


Figure 3. Distribution of pipe and subpipe size (number of modules).

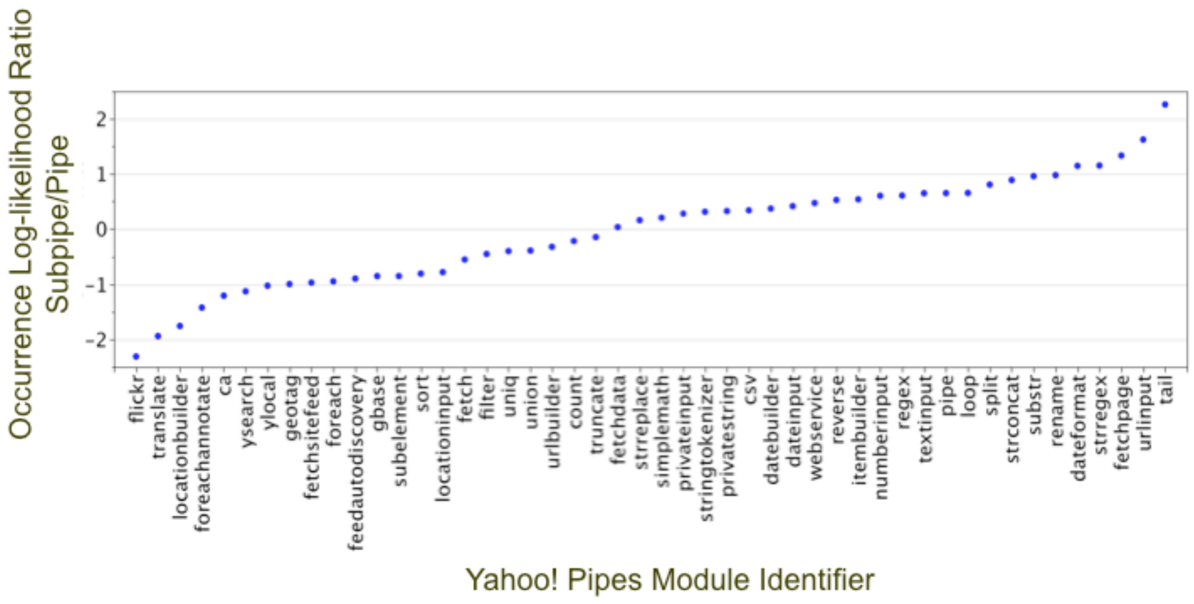


Figure 4. Log-likelihood ratio of use in a pipe vs. subpipe for each module type.

If, through log-likelihoods, we compare how often specific modules are used in pipes versus subpipes (Fig. 4), we see that the utilization of roughly ¼ of Pipes modules differs by several orders of magnitude between pipes and subpipes. Subpipes are not just a random subset of pipes that users have decided to encapsulate as extensions of the Pipes module set.

Given their tendency to include more, and different, modules in subpipes relative to pipes, it might be that users are attempting to use subpipes to hide complexity behind simple interfaces. In Fig. 5, we explore this hypothesis by comparing, for both pipes and subpipes, the average number of modules encapsulated in a composition—a coarse-grained indicator of functional complexity—with the number of interface parameters[3] that the author provides to users of the composition. We see that as the count grows, the number of parameters in both pipes and subpipes initially increases, but then levels off, even as "complexity" increases (see upper and middle panels of Fig. 5). This favors the "complexity-hiding" argument. However, except for very small compositions, the subpipes:pipes ratio of these counts hovers around 2.5:1, indicating that complexity-hiding is found in both types of compositions. Note: the erratic subpipe behavior at higher module counts is a sparse-data artifact.

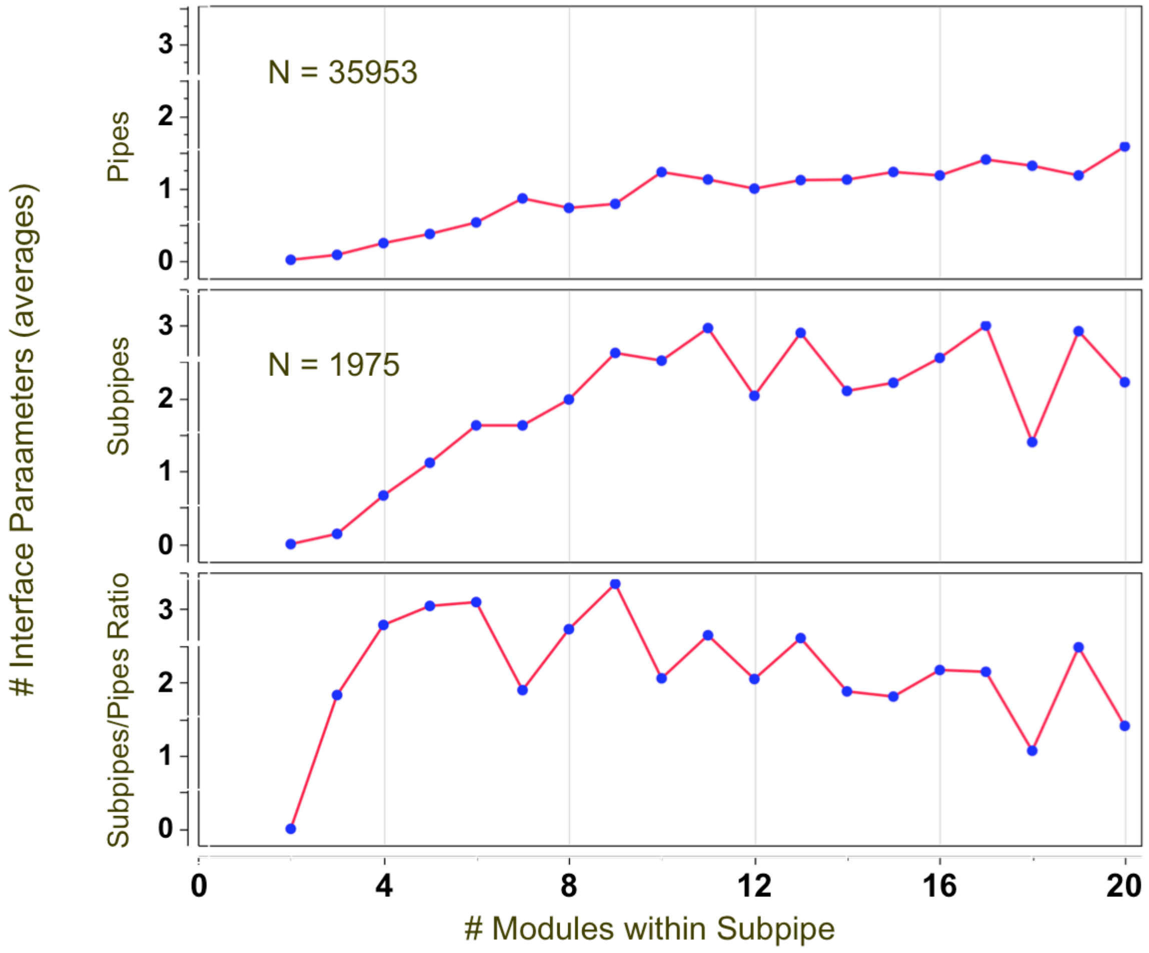


Figure 5. Comparative use of exposed interface parameters in pipes vs. subpipes

The "exponential" shape of the right-hand tail of each distribution in Fig. 2 led us to examine the log probabilities in those tails (Fig. 6). Not only is the exponential fit reasonably good, but the slopes of the fitted regression lines are both approximately 0.18. This suggests that, to a first approximation, an end-user adds modules to a composition based upon the outcome of tossing a figurative coin with p = exp(-0.18) = 0.84 in favor of the addition. Furthermore, this (binomial) process qualitatively describes module composition for both pipes and subpipes.

[3] When a pipe is converted into a subpipe, the pipe's run-time parameters become the subpipe's design-time parameters (section 2). We refer to both collectively as "interface parameters."

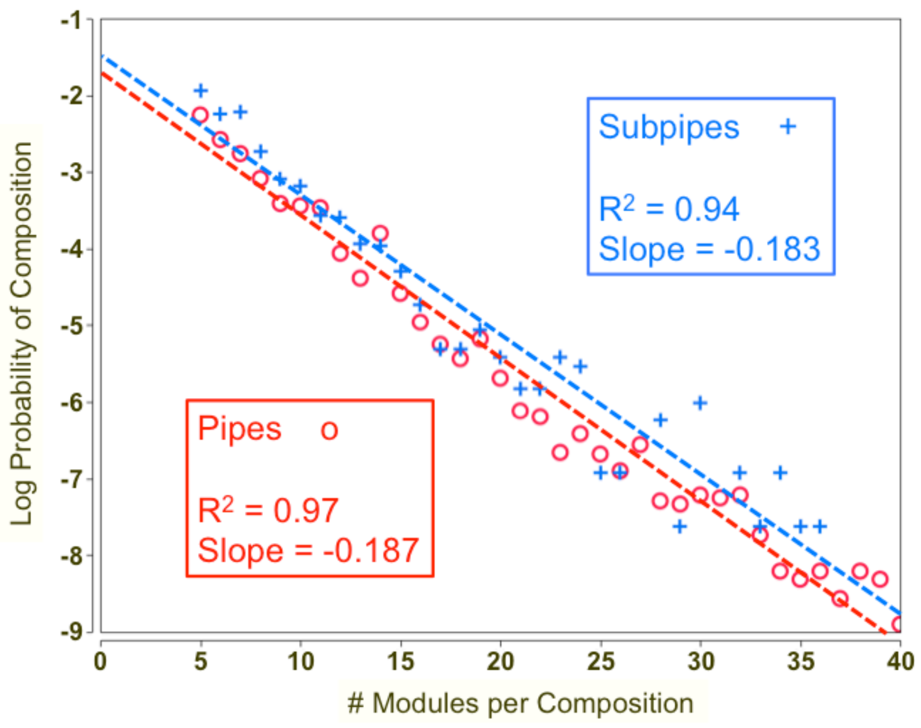


Figure 6. Log fit of probability of observing a pipe or subpipe containing N modules.

4.2 Choosing wires and configurations

As an environment originally intended for mashing-up RSS feeds, the Pipes environment includes features that encourage convergent "stream-processing." Not only is a composition allowed only one output, but wires are segregated into those that carry data and those that carry parameters; dataflow in both is one-way, and most modules have only one output (as does a pipe composition) and default to a handful of inputs. However, Pipes does include explicit "split" and "union" modules that allow streams to be split, merged, and processed in parallel, and a "loop" construct allows iteration. Many modules also allow expansion of parameters and their associated wires. The question is, do end-user programmers exploit this richness?

The answer is, "not very often." Fig. 7 plots the average and the range of the number of wires in a composition as a function of the number of modules in that composition. We do this for both pipes and subpipes, and we make no distinction between data- and parameter-carrying wires. For both pipes and subpipes, the average number of wires is almost perfectly predicted by the relation: # wires = # modules – 1. In other words, users tend to employ only the minimum number of wires needed to connect modules. Moreover, as evident in the range bars in Fig. 7, this propensity is even stronger in subpipes than it is in pipes. Given this finding, it seems reasonable to propose that end-users configure most Pipes as directed acyclic graphs, often a linear chain or cascade of modules.

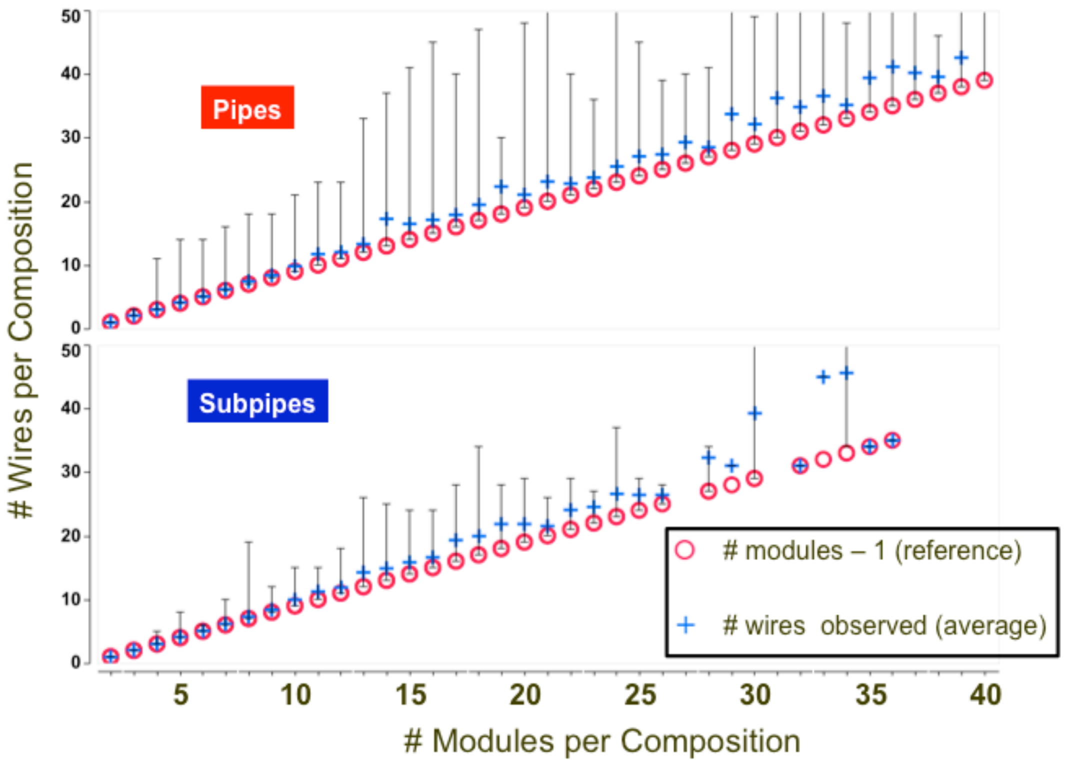


Figure 7. Relationship of number of modules and connecting wires for pipes and subpipes.

4.3 Choosing parameters

Most of the 51 primitive Pipes modules have parameters that end-user programmers must either default or set, some when the pipe or subpipe is configured ("design-time"), and some when it is executed ("run-time"). Fig. 8 shows how the design-time parameters distribute over the module collection as initially presented to the user as a "module menu."

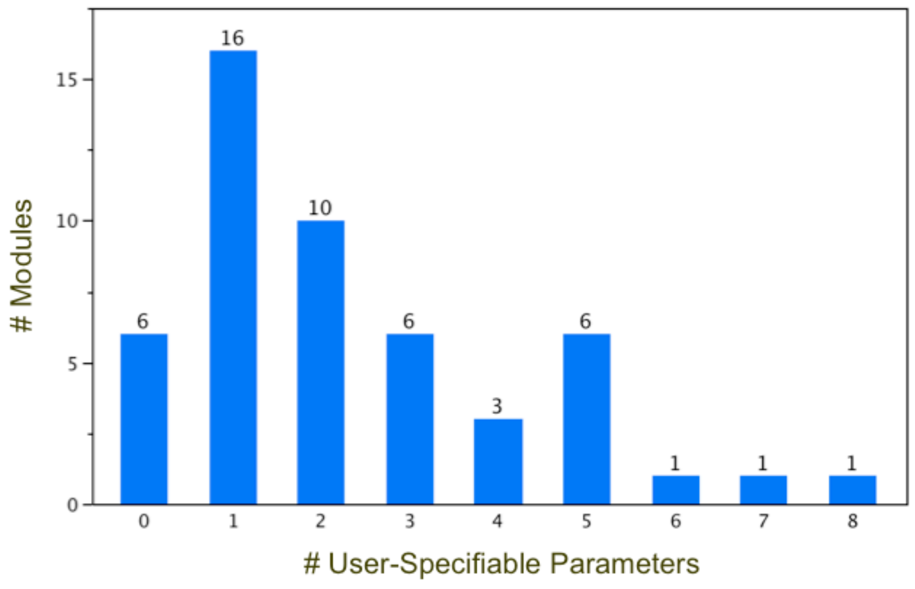


Figure 8. Distribution of available design-time parameters in the Pipes primitives.

We know from the module "popularity contest" in Fig. 2 (section 3.2) that users have strong preferences for certain modules in compositions, and that these preferences correlate with Pipe's use as a data mash-up environment. However, we also wondered if there might be a relationship with the number of parameters that a user must contend with in using a module: too few parameters might seem to limit flexibility; too many increases complexity (or "cognitive load") and work-factor. Fig. 9 plots the number of parameters in Pipes modules as a function of module usage or "popularity." As in Fig. 2, the most frequently used modules appear on the left, and the least-used on the right. We see no obvious pattern, perhaps because none exists, or perhaps because of two confounding factors: first, Pipes modules are functionally independent. One cannot readily derive the function of a given module from some allowable combination of the others. Pipes users thus have no choice about using certain modules for certain functions; unless users forego those functions altogether, their "preferences" make no difference in module choice. Second, during composition, users are able to add an arbitrary number of parameters to many modules (example: additional regular-expression patterns to the *regex* module). A "simple" module can thus become very complex at the user's discretion. Fig. 9 shows only the default number of parameters, not the results of such extensions.

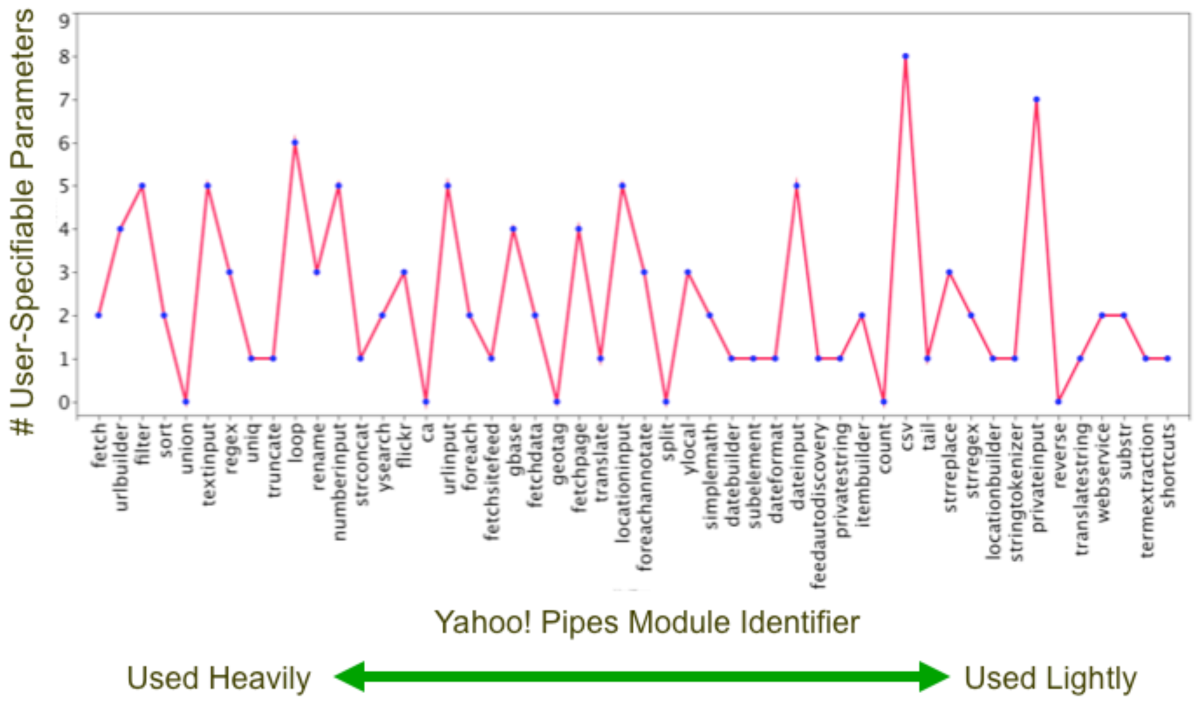


Figure 9. Relationship between primitive selection and design-time parameter count.

We can, however, gain insight into the influence of module parameters on end-user compositional behavior by looking at the fabrication and utilization of modules that users themselves create: the subpipes. Recall that when an end-user author publishes a subpipe, any interface parameters for the subpipe appear in the subpipe's representation as an addition to the "module menu." Authors are free to specify as many parameters as they wish. But if we look at how interface parameters actually distribute across subpipes, we see a familiar pattern (Fig. 10): subpipes with no parameters are the most common, and the frequency falls steeply as the number of parameters increases. The falloff is exponential and fits a binomial, coin-tossing model with p approximately 0.4. Thus, the process of adding interface parameters when a subpipe is originally composed is empirically akin to the process of adding modules to compositions in general (section 4.1).

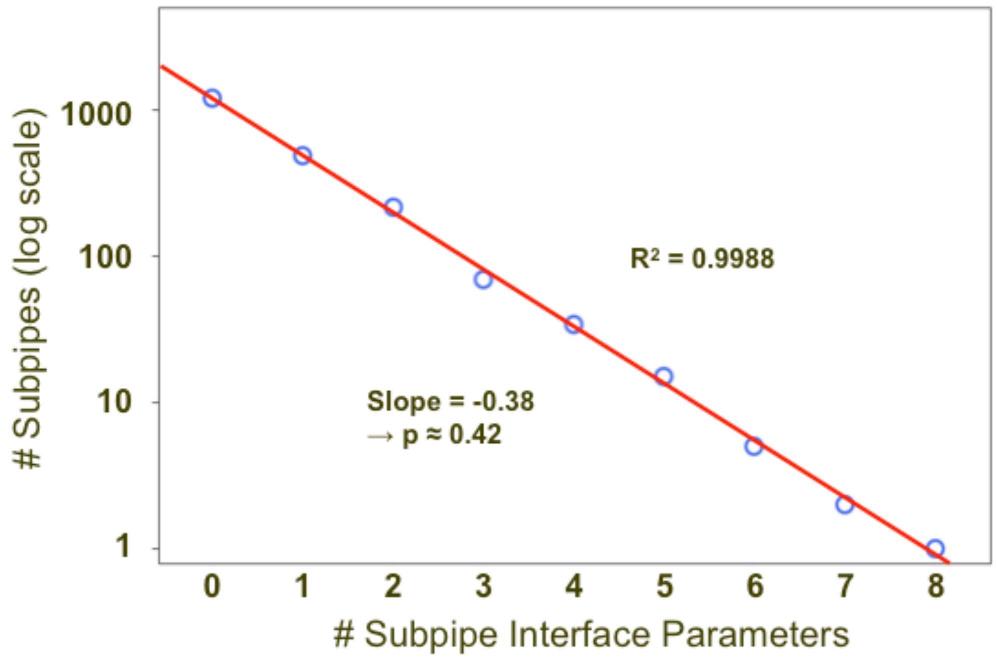


Figure 10: Log fit of relationship between observed subpipes and number of interface parameters.

How, then, might the number of interface parameters affect the likelihood that an end-user will choose to employ a published subpipe in another composition? Since subpipes are presumably created to save users the effort of fashioning specialized functions, we might expect to see the same "random" relationship between parameters and the likelihood of subpipe utilization as we do with other Pipes modules (Fig. 9). To examine this, we first need to find those subpipes that: (a) have actually been made available to other end-users by their authors; and (b) have actually seen service. Of the 2037 subpipes in the corpus, only 199 meet these criteria. Fig. 11 illustrates how often these subpipes were used in other compositions (average and range) as a function of the number of interface parameters that they present. It appears to us that, except when a subpipe has no parameters at all, the number of parameters has little influence on whether a subpipe is selected for use.

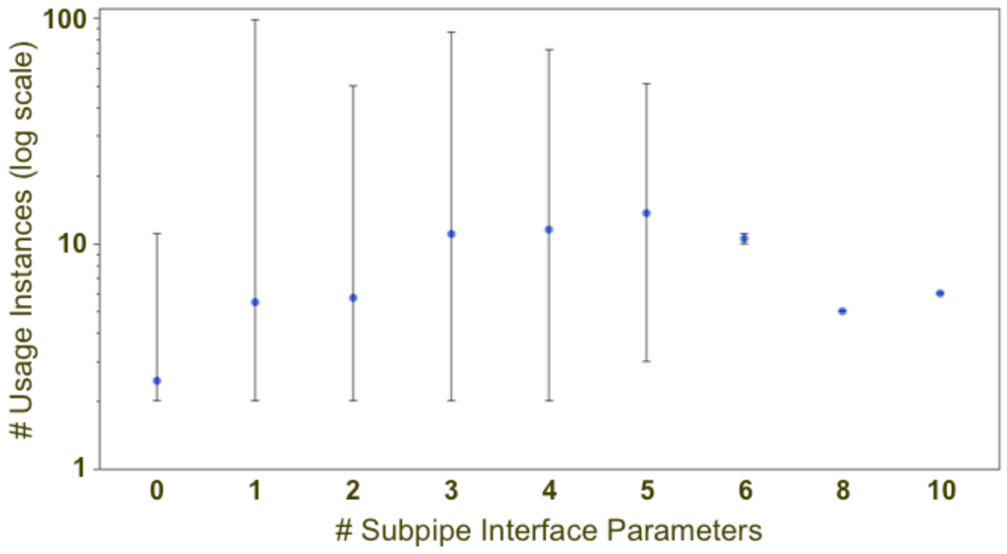


Figure 11. Relationship of exposed subpipe interface parameters and observed reuse.

A final question that we asked involved what we refer to as the reduction in degrees of freedom from the composition space to the use space. When a user composes a pipe, he or she has to make choices about which parameters to set within the composition at design time – thereby fixing those values – and which to expose to other users as run-time parameters. The ratio for a given pipe of parameters offered to users at run-time to those set and fixed at design-time is the reduction in degrees of freedom. We would expect that developers of pipes seek to make a task easier, and therefore attempt to reduce the number of parameters that need to be set by a user. However, providing too few run-time parameters may make the pipe less generalizable to other purposes, reducing its utility to users other than the developer. Figure 12 shows the distributions of design-time parameters actually set by the developer (this *includes* extended parameters, but *excludes* defaulted parameters), as well as the distribution of run-time parameters.

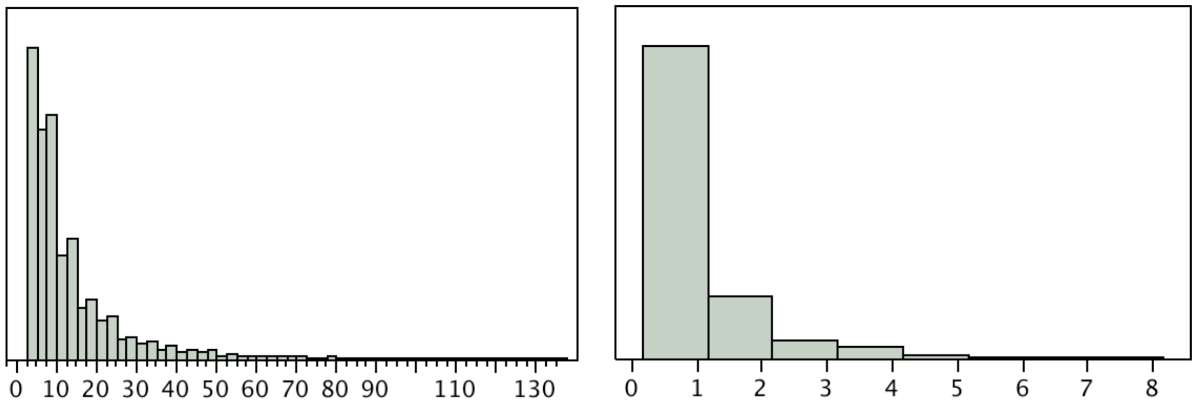


Figure 12. Design-time (left) and run-time (right) parameter frequencies.

The first thing that we note is that the majority (76%) of pipes offer users no run-time parameters at all. In the cases of both design-time and run-time parameters, we again see the binomial coin-tossing model (see fig. 10). However, when compared to each other (fig. 12), there does not appear to be an obvious relationship that would enable us to predict the number of run-time parameters given the number of design-time parameters. To better visualize this, we grouped ranges of design-time parameters (15 per bin), and then plotted the number of pipes corresponding to each binned design-time parameter count and run-time parameter counts as bubbles; it appears that these selection processes are independent (fig. 13).

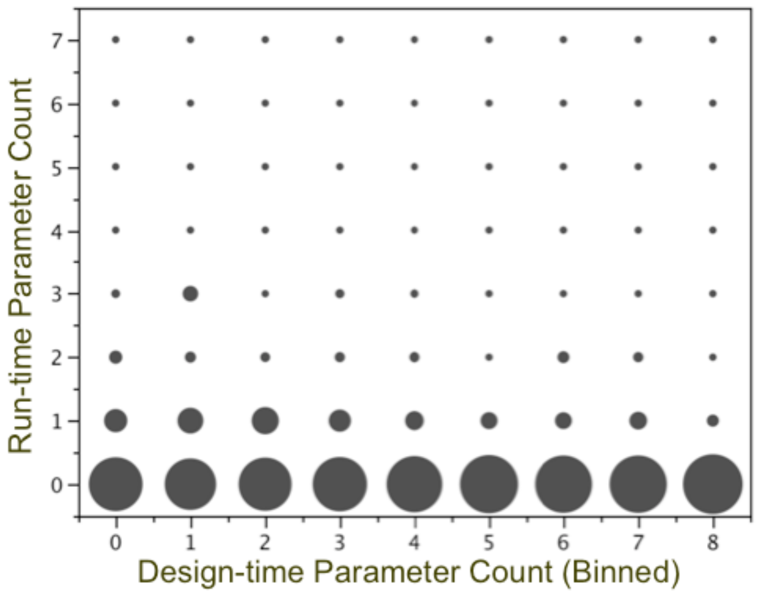


Figure 13. Run-time parameters as predicted by design-time parameters.

5. Discussion

In our view, it would be premature to aim for deep conclusions about end-user programming behavior from the empirical results offered here. Pipes is a specialized EUP environment, and we clearly need to compare our findings with those from other domains operating at similar scale. Nevertheless, certain principles do seem to govern how end-users approach Pipes:

First, and most obviously, users employ only a small number of the programming options that Pipes offers to them,. They regularly make use of perhaps one-quarter of the available primitives and typically compose them in pipes consisting of only three or four modules. Further, while they are able to create more complex, multi-branch pipelines, they strongly favor simple, straight-through pipes.

Second, we repeatedly see behaviors that fit simple models quite well. In particular, the recurring "coin tossing" model suggests that the decision process for adding a module or interface parameter to a pipe is invariant and independent. This design behavior seems to reflect a process in which, in order to achieve a particular objective, users choose modules from those available, at each step determining if they have or have not achieved the objective (the metaphorical coin toss). The result is what would appear to be the simplest working solution to the problem, evidenced by the strongly linear structure of pipes, in effect, leaving no real choice for wiring. Similarly, users set design-time parameters – as few as possible – to achieve an objective, again asking the question, “am I done?” Decoupled from this is the choice of parameters to expose at run-time. Usually, users simply hardwire the pipe and offer no run-time parameters, though when they do, we again see the coin-toss behavior.

5.1 Future Work

Our study of the Pipes data has only looked at the surface aspects of compositional behavior. There are several other research questions that we propose could be answered by this data. First is a deeper study of the nature of subpipes, clones and other reuse behaviors. Initial research we have done shows that it is possible to identify the lineage of individual pipes to form "families" of related pipes; we have found that the cloning metadata associated with a pipe is an unreliable indicator of actual cloning behavior.

Additional design-time information available to us will allow us to examine questions such as the order in which a user chose modules to place in the workspace, and the spatial organization of pipes. The first may offer insights into intent and planning, while the second may show the evolution of the design and whether the available screen space is a constraint, despite the availability of an essentially unlimited canvas.

We also collected other use-time information, such as the run count for each pipe; this was collected longitudinally, so we can examine the relationship between pipe design (as studied here) and "popularity" at an instant and over time.

Of course, we have completely ignored the semantics of the pipes involved; what was the user trying to accomplish? Information available in tags associated with the pipes, text content, and perhaps even the use of certain modules, could all be used to categorize pipes. An interesting question is whether any particular category exhibits significant differences from the aggregate behaviors we have described here.

6. Acknowledgments

The authors would like to thank our colleagues Rose Daley and Peter Li for their contributions to our study of the Pipes data.

7. References

Blackwell, A. (2002). *First Steps in Programming: A Rationale for Attention Investment Models*. Paper presented at the IEEE Symposia on Human-Centric Computing Languages and Environments.

Bogart, C., Burnett, M., Cypher, A., & Scaffidi, C. (2008). *End-User Programming in the Wild: A Field Study of CoScripter Scripts*. Paper presented at the IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC '08), Herrsching am Ammersee, Germany.

Burnett, M. (1999). Visual Programming. In J. G. Webster (Ed.), *Encyclopedia of Electrical and Electronics Engineering.* New York: John Wiley & Sons.

Fisher, M., & Rothermel, G. (2005). *The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms*. Paper presented at the Proceedings of the first workshop on End-user software engineering, St. Louis, Missouri.

Gantt, M., & Nardi, B. A. (1992). *Gardeners and gurus: patterns of cooperation among CAD users*. Paper presented at the SIGCHI conference on Human factors in computing systems, Monterey, California, United States.

Jones, M. C., & Churchill, E. F. (2009). *Conversations in developer communities: a preliminary analysis of the yahoo! pipes community*. Paper presented at the The fourth international conference on Communities and technologies, University Park, PA, USA.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv*, *37*(2), 83-137.

Ko, A., Abraham, R., Beckwith, L., Blackwell, A. F., Burnett, M., Erwig, M. et al. (2008). The State of the Art in End-User Software Engineering. *ACM Computing Surveys*.

Myers, B. A., Ko, A. J., & Burnett, M. M. (2006). *Invited research overview: end-user programming*. Paper presented at the CHI '06 extended abstracts on Human factors in computing systems, Montréal, Québec, Canada.

Nardi, B., A., & Miller, J., R. (1990). *An ethnographic study of distributed problem solving in spreadsheet development*. Paper presented at the Proceedings of the 1990 ACM conference on Computer-supported cooperative work.

Repenning, A., & Ioannidou, A. (2006). What Makes End-User Development Tick? 13 Design Guidelines. *End User Development,* 51-85.

Whitley, K., N., Novick, L., R., & Fisher, D. (2006). Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility. *Int. J. Hum.-Comput. Stud*, *64*(4), 281-303.

Bricolage Programming in the Creative Arts

Alex McLean and Geraint Wiggins

Centre for Cognition, Computation and Culture
Department of Computing
Goldsmiths, University of London

Abstract. In this paper we consider artists who create their work by writing algorithms, which when interpreted by a computer generates their plotted drawings, synthesised music, animated digital video, or whatever target medium they have chosen. We examine the demands that such artists place upon their environments, the relationships between concepts and algorithms, and of cognition and computation. We begin by considering an artist's creative process, and situating it within the bricolage style of programming. An embodied view of bricolage programming is related, underpinned by theories of cognitive metaphor and computational creativity, and finally with consideration of the bricolage programmer's relation to time.

1 Introduction

Over the last decade, computer programming has enjoyed a major resurgence as a medium for the arts. A wealth of new programming environments for the arts, such as Processing, SuperCollider, ChucK, VVVV and OpenFrameworks have joined more established environments such as PureData and Max which have themselves gained enthusiastic adoption outside their traditional academic base. These environments offer varied approaches to supporting artistic use, including alternative programming languages, interfaces and workflow.

The purpose of the present discussion is to examine psychological issues which the resurgence of artistic programming has brought to the fore. What is the relationship between an artist, their creative process, their program, and their artistic works? We will look for answers from perspectives of psychology, cognitive linguistics, computer science and computational creativity, but first from the perspective of the artist.

2 Creative Processes

The painter Paul Klee [1953, p. 33] describes a creative process as a feedback loop: "Already at the very beginning of the productive act, shortly after the initial motion to create, occurs the first counter motion, the initial movement of receptivity. This means: the creator controls whether what he has produced so far is good. The work as human action (genesis) is productive as well as receptive. It is **continuity**." This is creativity without planning, a feedback loop of making a mark on canvas, perceiving the effect, and reacting with a further mark. Being engaged in a tight creative feedback loop places the artist close to their work, guiding an idea to unforeseeable conclusion through a flow of creative perception and action. Klee writes as a painter, working directly with his medium. Programmer-artists instead work using computer language as a textual representation of their medium, and it might seem that this extra level of abstraction could hinder creative feedback. We will see however that this is not necessarily the case, beginning with the account of Turkle and Papert [1992], describing a *bricolage* approach to programming by analogy with painting:

> The bricoleur resembles the painter who stands back between brushstrokes, looks at the canvas, and only after this contemplation, decides what to do next. Bricoleurs use a mastery of associations and interactions. For planners, mistakes are missteps; bricoleurs use a navigation of midcourse corrections. For planners, a program is an instrument for premeditated control; bricoleurs have goals but set out to realize them in the spirit of

a collaborative venture with the machine. For planners, getting a program to work is like "saying one's piece"; for bricoleurs, it is more like a conversation than a monologue. [Turkle and Papert, 1990, p. 136]

Although Turkle and Papert address gender issues in education, this quote should not be misread as dividing all programmers into two types; while associating bricolage with feminine and planning with male traits, they are careful to note that these are extremes of a behavioural continuum. Indeed, programming style is clearly task specific: for example a project requiring a large team needs more planning than a short script written by the end user.

Bricolage programming seems particularly applicable to artistic tasks, such as writing software to generate music, video animation or still images. Imagine a visual artist, programming their work using Processing. They may begin with an urge to draw superimposed curved lines, become interested in a tree-like structure they perceive in the output of their first implementation, and change their program to explore this new theme further. The addition of the algorithmic step would appear to affect the creative process as a whole, and we seek to understand how in the following.

2.1 Creative Process of Bricolage

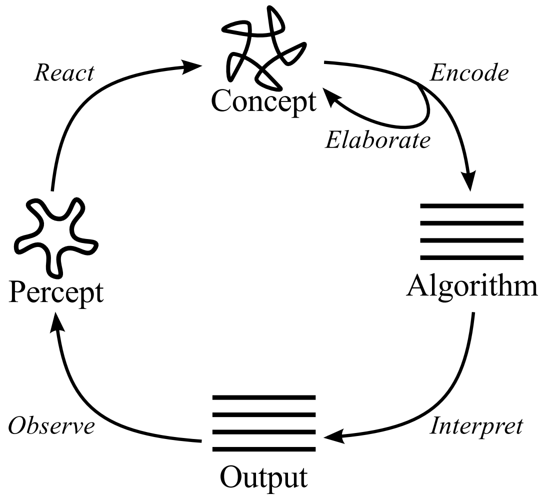


Fig. 1. The process of action and reaction in bricolage programming

Figure 1 shows bricolage programming as a creative feedback loop encompassing the written algorithm, its interpretation, and the programmer's perception and reaction to its output or behaviour. The addition of the algorithmic component in the creative feedback loop makes an additional inner loop explicit between the programmer and their text. At the beginning, the programmer may have a half-formed concept, which only reaches internal consistency through the process of being expressed as an algorithm. The inner loop is where the programmer elaborates upon their imagination of what might be, and the outer where this trajectory is grounded in the pragmatics of what they have actually made. Through this process both algorithm and concept are developed, until the programmer feels they accord with one another or otherwise judges the creative process to be finished.

Representations in the computer and the brain are evidently distinct from one another. Computer output evokes perception, but that percept will both exclude features that are explicit in the output and include features that are not, due to a host of effects including attention, knowledge and illusion. Equally, a human concept is distinct from a computer algorithm. Perhaps a program written in a declarative rather than imperative style is somewhat closer to a concept, being not an algorithm for how to carry out a task, but rather a description of what is to be done. But still, there is a clear line to be drawn between a string of discrete symbols in code and the morass of symbolic, spatial and relational representations we assume underlies cognition.

There is however something curious about how the programmer's creative process spawns a second, computational one. The computational process is lacking in the cognitive abilities of its author, but is nonetheless both faster and more accurate at certain tasks by several orders of magnitude. It would seem that the programmer uses the programming language and its interpreter as a cognitive resource, augmenting their own abilities in line with the extended mind hypothesis [Clark, 2008]. We will revisit this issue within a formal framework in §5, after first looking more broadly at how we relate programming to human experience, and related issues of representation.

3 Anthropomorphism and Metaphor in Programming

Metaphor permeates our understanding of programming. Perhaps this is due to the abstract nature of computer programs, requiring metaphorical constructs to allow us to ground programming language in everyday reasoning. Petre and Blackwell [1999] gave subjects programming tasks, and asked them to introspect upon their imagination while they worked. These self reports are rich and varied, including exploration of a landscape of solutions, dealing with interacting creatures, transforming a dance of symbols, hearing missing code as auditory buzzing, combinatorial graph operations, munching machines, dynamic mapping and conversation. While we cannot rely on these introspective reports as authoritative on the inner workings of the mind, the diversity of response hints at highly personalised creative processes, related to physical operations in visual or sonic environments. It would seem that a programmer uses metaphorical constructs defined largely by themselves and not by the computer languages they use. However mechanisms for sharing metaphor within a culture do exist. Blackwell [2006a] used corpus linguistic techniques on programming language documentation in order to investigate the conceptual systems of programmers, identifying a number of conceptual metaphors listed in Figure 2. Rather than finding metaphors supporting a mechanical, mathematical or logical approach as you might expect, components were instead described as actors with beliefs and intentions, being social entities acting as proxies for their developers.

The above research suggests that programmers understand the operation of their programs by metaphorical relation to their experience as a human. Indeed the feedback loop described in §2 is by nature anthropomorphic; by embedding the development of an algorithm in a human creative process, the algorithm itself becomes a human expression. Dijkstra [1988] strongly opposed such approaches: "I have now encountered programs wanting things, knowing things, expecting things, believing things, etc., and each time that gave rise to avoidable confusions. The analogy that underlies this personification is so shallow that it is not only misleading but also paralyzing." Dijkstra's claim is that by focusing on the operation of algorithms, the programmer submits to a combinatorial explosion of possibilities for how a program might run; not every case can be covered, and so bugs result. Dijkstra argues for a strict, declarative approach to computer science and programming in general, which he views as so radical that we should not associate it with our daily existence, or else limit its development and produce bad software.

The alternative view presented here is that metaphors necessarily structure our understanding of computation. This view is sympathetic to a common assumption in the field of cognitive linguistics, that our concepts are organised in relation to each other and to our bodies, through conceptual systems of metaphor. Software now permeates Western society, and is required to function reliably according to human perception of time and environment. Metaphors of software as human activity are therefore becoming ever more relevant.

Components are agents of action in a causal universe.
Programs operate in historical time.
Program state can be measured in quantitative terms.
Components are members of a society.
Components own and trade data.
Components are subject to legal constraints.
Method calls are speech acts.
Components have communicative intent.
A component has beliefs and intentions.
Components observe and seek information in the execution environment.
Components are subject to moral and aesthetic judgement.
Programs operate in a spatial world with containment and extent.
Execution is a journey in some landscape.
Program logic is a physical structure, with material properties and subject to decay.
Data is a substance that flows and is stored.
Technical relationships are violent encounters.
Programs can author texts.
Programs can construct displays.
Data is a genetic, metabolizing lifeform with body parts.
Software tasks and behaviour are delegated by automaticity.
Software exists in a cultural/historical context.
Software components are social proxies for their authors.

Fig. 2. Conceptual metaphors derived from analysis of Java library documentation by Blackwell [2006a]. Program components are described metaphorically as actors with beliefs and intentions, rather than mechanical imperative or mathematical declarative models.

4 Symbols and Space

We now turn our attention to how the components of the bricolage programming process shown in Figure 1 are represented, in order to ground understanding of how they may interrelate. Building upon the anthropomorphic view taken above, we propose that in bricolage programming, the human cognitive representation of programs centres around perception. Perception is a low dimensional representation of sensory input, giving us a somewhat coherent, spatial view of our environment. By spatial we do not just mean in terms of physical objects, but also in terms of features in the spaces of all possible tastes, sounds, tactile textures and so on. This scene is built through a process of dimensional reduction from tens of thousands of chemo-, photo-, mechano- and thermoreceptor signals. Algorithms on the other hand are represented in discrete symbolic sequences, as is their output, which must go through some form of digital-to-analogue conversion before being presented to our sensory apparatus, for example as light from a monitor screen or sound pressure waves from speakers, a process we call observation. Recall the programmer from §2, who saw something not represented in the algorithm or even in its output, but only in their own perception of the output; observation may itself be a creative act.

The component from Figure 1 not yet mentioned in this section is that of programmers' concepts. A concept is 'a mental representation of a class of things' [Murphy, 2002, p.5]. Figure 1 shows concepts mediating between spatial perception and symbolic algorithms, leading us to ask; are concepts represented more like spatial geometry, like percepts, or symbolic language, like algorithms? Our focus on metaphor leads us to take the former view, that conceptual representation is grounded in perception and the body. This view is taken from Conceptual Metaphor Theory (CMT) introduced by Lakoff and Johnson [1980], which proposes that concepts are primarily structured by metaphorical relations, the majority of which are orientational, understood relative to the human body in space or time. In other words, the conceptual system is grounded in the perceptual system. Gärdenfors [2000] builds upon this by further proposing that the semantic meanings of concepts and the metaphorical relationships between them are geometrical properties and relationships. Concepts themselves are represented by geometric regions of low dimensional spaces defined by quality dimensions, either mapped directly from, or structured by metaphorical relation to perceptual qualities. For example "red" and "blue" are regions in perceptual colour space, and the metaphoric semantics of concepts within the spaces of mood, temperature and importance may be defined relative to geometric relationships of such colours.

Gärdenforsian conceptual spaces are compelling when applied to concepts related to bodily perception, emotion and movement, and Forth et al. [2008] report early success in computational representations of conceptual spaces of musical rhythm and timbre, through reference to research in music perception. However, it is difficult to imagine taking a similar approach to computer programs. What would the quality dimensions of a geometrical space containing all computer programs be? There is no place to begin to answer this question; computer programs are symbolic in nature, and cannot be coherently mapped to a geometrical space grounded in perception.

For clarity we turn once again to Gärdenfors [2000], who points out that spatial representation is not in opposition to symbolic representation; they are distinct but support one another. This is clear in computing, hardware exists in our world of continuous space, but thanks to reliable electronics, conjures up a symbolic world of discrete computation. Our minds are able to do the same, for example by computing calculations in our head, or encoding concepts into phonetic movements of the vocal tract or alphabetic symbols on the page. We can think of ourselves as spatial beings able to simulate a symbolic environment to conduct abstract thought and open channels of communication. On the other hand, a piece of computer software is a symbolic being able to simulate spatial environments, perhaps to create a game world or guide robotic movements, both of which may include some kind of model of human perception.

Computer language operates in the domain of abstraction and communication but in general does not at base include spatial semantics. In some cases computer languages are described as 'visual' even when spatial arrangement is purely secondary notation, ignored by the interpreter, such as in Patcher languages [Puckette, 1988]. In fact spatial layout is a feature of secondary notation in mainstream 'textual' languages too, through use of whitespace with no syntactical meaning. That programmers need to use spatial layout as a crutch while composing symbolic sequences is telling; to the interpreter, a block may be a subsequence between braces, but to an experienced programmer it is a perceptual gestalt grouped by indentation. From this we can understand computation as separate from spatial reasoning, but supported by it, with secondary notation helping bridge the divide.

An important aspect of CMT is that a conceptual system of semantic meaning exists within an individual, not in the world. Through language, metaphors become established in a culture and shared by its participants, but this is an effect of individual conceptual systems interacting, and not individuals inferring and adopting external truths of the world (or of possible worlds). This would account for the varied range of metaphor in programming discussed in §3, as well as the general failure of attempts at designing metaphor into computer interfaces [Blackwell, 2006b]. Each programmer has a different set of worldly interests and experiences, and so establishes different metaphorical systems to support their programming activities.

5 Components of creativity

We now have sufficient grounds to fully characterise how the creative process operates in our case study of bricolage programming. For this we employ the Creative Systems Framework (CSF), a high-level formalisation of creativity introduced by Wiggins [2006a,b] and based upon the work of Boden [2003]. Creativity is characterised as a search in a space of concepts. Three sets of rules are employed in this search; $\mathcal{R}$ defining the *search space* itself, $\mathcal{T}$ defining *traversal* of the space and $\mathcal{E}$ defining *evaluation* of concepts found in the space. However, the CSF describes much more than a reactive process of traversal and evaluation. Creativity also requires introspection, self-modification and for boundaries to be broken. In other words, the rulesets $\mathcal{R}$, $\mathcal{T}$ and $\mathcal{E}$ are examined and challenged by the creative agent following them.

Using the terms of Gärdenfors [2000], $\mathcal{R}$ is a concept defining a space of concept instances.[1] For example in a creative search for music within a genre, the genre would be the concept and a piece of music conforming to a genre would be a instance of that concept. Crucially, $\mathcal{R}$ is not a closed space, but rather defined as a subspace of the universe of all possible concepts. This means that a creative agent may creatively push beyond the boundaries of the search as we will see.

We are now in a position to clarify the bricolage programming process introduced in §2.1 within the CSF. As shown in Figure 3, the ruleset $\mathcal{R}$ defines the programmer's concept, being their current artistic focus structured by learnt techniques and conventions, the traversal strategy $\mathcal{T}$ is the process of encoding and interpreting the algorithm, and the evaluation function $\mathcal{E}$ is the perceptual process of observation and reaction.

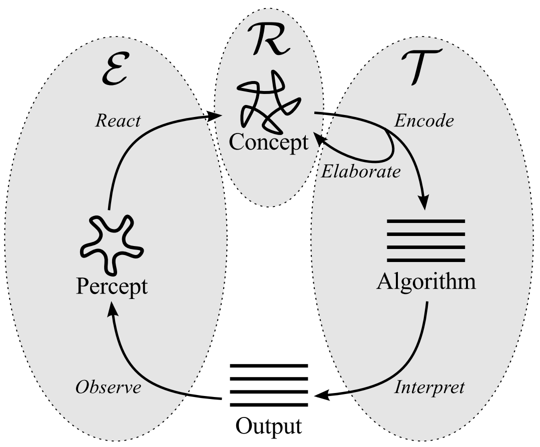


Fig. 3. The process of action and reaction in bricolage programming from Figure 1, annotated with the $\mathcal{R}$ conceptual space, $\mathcal{T}$ traversal strategy and $\mathcal{E}$ evaluation components of the Creative Systems Framework.

In §2, we alluded to the extended mind hypothesis [Clark, 2008], claiming that bricolage programming takes part of the human creative process outside of the mind and into the computer. The above makes clear what we claim is being externalised: part of the traversal strategy $\mathcal{T}$. The programmer's concept $\mathcal{R}$ motivates a development of the strategy $\mathcal{T}$ to be encoded in a program, but the programmer does not necessarily have the cognitive ability to fully evaluate the program. That task is taken on by the interpreter running on a computer system, meaning that $\mathcal{T}$ encompasses both encoding by the human and interpretation by the computer.

The traversal strategy $\mathcal{T}$ is structured by the techniques and conventions employed to convert concepts into operational algorithms. These may include *design patterns*, a standardised set of *ways of building* that have become established around imperative programming languages. Each design pattern identifies a *kind* of problem, and describes a *kind* of structure as a *kind* of solution.[2]

The creative process is constrained by $\mathcal{R}$, being the programmer's idea of what is a valid end result. This is shaped by the programmer's current artistic focus, being the perceptual qualities they are currently interested in, perhaps congruent with a cultural theme such as a musical genre or artistic movement. Transformational creativity can be triggered in the CSF when application of $\mathcal{T}$ results in a concept instance that exists outside the constraining bounds of $\mathcal{R}$, shown in Figure 4. If the instance is valued according to $\mathcal{E}$, then $\mathcal{R}$ is changed to include

[1] The terms used by Gärdenfors [2000] diverge from those used by Wiggins [2006a,b]. Wiggins uses the term *conceptual space* in the place of Gärdenfors' *concept*, and *concept* in the place of *concept instance*. The meaning is however the same, particularly when the recursive hierarchy of Wiggins' theory is taken into account.

[2] Interestingly, this structural heuristic approach to problem solving originated in the field of urban design [Alexander et al., 1977].

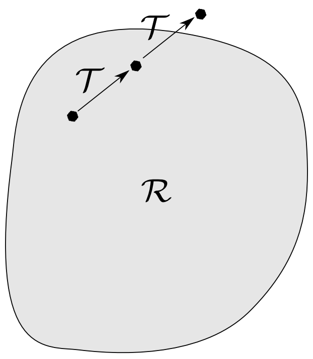


Fig. 4. Application of a traversal strategy $\mathcal{T}$ leading outside the concept $\mathcal{R}$, triggering transformational creativity.

it. If the instance is not valued, then $\mathcal{T}$ is changed to avoid that instance in the future. As a result of including external interpretation in $\mathcal{T}$, the programmer is likely to be less successful in writing software that meets their preconceptions, but as a result more successful in being surprised by the results. In other words, the artist's act of externalising part of $\mathcal{T}$ as a computer program makes the results less predictable, and transformational creativity more likely.

In artistic bricolage programming, then, we conclude that creativity is a process of imagining a concept $\mathcal{R}$, encoding an operational algorithm as part of $\mathcal{T}$ to explore within and beyond $\mathcal{R}$, and a perceptual process $\mathcal{E}$ to evaluate the output. Through this process both $\mathcal{R}$ and $\mathcal{T}$ are continually transformed in respect of one another, in creative feedback.

According to our embodied view, not only is perception crucial in evaluating output within bricolage programming, but also in structuring the space in which programs are conceptualised. Indeed if the embodied view of CMT holds in general, the same would apply to all creative endeavour. From this we find a message for the field of computational creativity: a prerequisite for an artificial creative agent is in acquiring computational models of perception sufficient to both evaluate its own works and structure its conceptual system. Only then will the agent have a basis for guiding changes to its own conceptual system and generative traversal strategy, able to modify itself to find artifacts that it was not programmed to find, and place value judgements on them. Such an agent would need to adapt to human culture in order to interact with shifting cultural norms, keeping its conceptual system and resultant creative process as coherent within that culture. For now however this is all wishful thinking, and we must be content with generative computer programs which extend human creativity, but are not creative agents in their own right.

6 Programming in Time

"She is not manipulating the machine by turning knobs or pressing buttons. She is writing messages to it by spelling out instructions letter by letter. Her painfully slow typing seems laborious to adults, but she carries on with an absorption that makes it clear that time has lost its meaning for her." Sherry Turkle [2005, p. 92], on Robin, aged 4, programming a computer.

Having investigated the representation and operation of bricolage programming we now examine how the creative process operates in time. Considering computer programs as operating in time at all, rather than as logic abstract from the world, is itself a form of the anthropomorphism examined in §3. However from the above quotation it seems that Robin stepped out of any notion of physical time, and into the algorithm she was composing, entering a timeless state. Speaking anecdotally, programmers report losing hours as they get 'in the flow' when writing software. Perhaps a programmer is thinking in algorithmic time, attending to control flow as it replays over and over in their imagination, and not to the world around them. Or perhaps they are not attending to the passage of time at all, thinking entirely of declarative abstract logic, in a timeless state of building. In either case, it would seem that the human is entering time relationships of their software, rather than the opposite, anthropomorphic direction of software entering human time. However there are ways in which human and computational time may be united, which we will come to shortly.

Temporal relationships are generally not represented in source code. When a programmer needs to do so, for example as an experimental psychologist requiring accurate time measurements, or a musician needing accurate synchronisation between processes, they run into problems. With the wide proliferation of interacting embedded systems, this is becoming a broad concern [Lee, 2009]. In commodity systems time has been decentralised, abstracted away through layers of caching, where exact temporal dependencies and intervals between events are not deemed worthy of general interest. Programmers talk of 'processing cycles' as a valuable resource which their processes should conserve, but they generally no longer have programmatic access to the high frequency oscillations of the central processing units (now, frequently plural) in their computer. The allocation of time to processes is organised top-down by an overseeing scheduler, and programmers must work to achieve what timing guarantees are available. All is not lost however, realtime kernels are now available for commodity systems, allowing psychologists [Finney, 2001] and musicians (e.g. via `http://jackaudio.org/`) to get closer to physical time. Further, the representation of time semantics in programming is undergoing active research in a subfield of computer science known as *reactive programming* [Elliott, 2009], with applications emerging in music [McLean and Wiggins, 2010].

6.1 Interactive programming

Programmers who 'think' in algorithmic time, like Robin earlier, are well served by dynamically interpreted languages. These allow a programmer to examine an algorithm while it is interpreted, taking on live changes without restarts. This is known as *interactive programming*, and unites the time flow of a program with that of its development. Interactive programming makes a dynamic creative process of test-while-implement possible, rather than the conventional implement-compile-test cycle, so that arrows shown in Figures 1 and 3 show concurrent influences between components rather than time-ordered steps.

Interactive programming not only provides a more efficient creative feedback loop, but also allows a programmer to connect software development with time based art. Since 2003 an active group of practitioners and researchers have been developing new approaches to making computer music and video animation, collectively known as *Live coding* [Blackwell and Collins, 2005, Ward et al., 2004, Collins et al., 2003, Rohrhuber et al., 2005]. The archetypal live coding performance involves programmers writing code on stage, with their screens projected for an audience, while the code is dynamically interpreted to generate music or video. Here the process of development is the performance, with the work generated not by a finished program, but its journey of development from an empty text editor to complex algorithm, generating continuously changing musical or visual form along the way. This is bricolage programming perhaps taken to a logical and artistic conclusion.

7 Conclusion

What we have seen provides strong motivation for programming which address the concerns of artists. These include concerns of workflow, where any time elapsed between source code edit and interpreted output is slows the creative process. Concerns of interfaces are also important, where in certain situations greater emphasis is placed on presentation of short scripts in their entirety as per bricolage programming, rather than hierarchical views of larger codebases. Perhaps most importantly, we have seen motivated the development of programming languages to greater support artistic expression.

From the embodied view we have taken, it would seem useful to integrate time and space further into programming languages. In practice integrating time can mean on one hand including temporal representations in core language semantics, and on the other uniting development time with execution time, as we have seen with interactive programming. Temporal semantics and interactive programming both already feature strongly in some programming languages for the arts, as we saw in §6, but how about analogous developments in integrating space into the semantics and activity of programming? This is a less well understood area requiring further research, but it would seem that novel approaches to the integration of computational geometry and perceptual models such as computer vision into programming language could serve artists well. By harnessing and extending research into visual programming languages, this could extend to notation, taking for example the ReacTable as inspiration [Jordà et al., 2007].

We began with Paul Klee, a painter whose production was limited by his two hands. The artist-programmer is not so limited, but shares what Klee called his limitation of reception, by the "limitations of the perceiving eye". This is perhaps a limitation to be expanded but not overcome, rather celebrated and fully explored using all we have, including our new computer languages. We have characterised a bricolage approach to artistic programming as an embodied, creative feedback loop. This places the programmer close to their work, grounding symbolic computation in orientational and temporal metaphors of their human experience. However the computer interpreter extends the programmer's abilities beyond their own imagination, making unexpected results likely, leading the programmer to new creative possibilities.

Bibliography

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, first edition, August 1977. ISBN 0195019199.

Alan Blackwell and Nick Collins. The programming language as a musical instrument. In *Proceedings of PPIG05.* University of Sussex, 2005.

Alan F. Blackwell. Metaphors we program by: Space, action and society in java. In *Proceedings of the Psychology of Programming Interest Group 2006*, 2006a.

Alan F. Blackwell. The reification of metaphor as a design tool. *ACM Trans. Comput.-Hum. Interact.*, 13(4):490–530, December 2006b. ISSN 1073-0516. doi: 10.1145/1188816.1188820.

Margaret A. Boden. *The Creative Mind: Myths and Mechanisms.* Routledge, 2 edition, November 2003. ISBN 0415314534.

Andy Clark. *Supersizing the Mind: Embodiment, Action, and Cognitive Extension (Philosophy of Mind Series).* OUP USA, November 2008. ISBN 0195333217.

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003. doi: 10.1017/S135577180300030X.

Edsger W. Dijkstra. On the cruelty of really teaching computing science. 1988.

Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

Steven A. Finney. Real-time data collection in linux: A case study. *Behavior Research Methods, Instruments, & Computers*, 33(2):167–173, May 2001.

Jamie Forth, Alex McLean, and Geraint Wiggins. Musical creativity on the conceptual level. In *IJWCC 2008*, 2008.

Peter Gärdenfors. *Conceptual Spaces: The Geometry of Thought.* The MIT Press, March 2000. ISBN 0262071991.

S. Jordà, G. Geiger, M. Alonso, and M. Kaltenbrunner. The reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proc. Intl. Conf. Tangible and Embedded Interaction (TEI07)*, 2007.

Paul Klee. *Pedagogical sketchbook.* Faber and Faber, 1953.

George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, first edition edition, April 1980. ISBN 0226468011.

Edward A. Lee. Computing needs time. *Commun. ACM*, 52(5):70–79, 2009. ISSN 0001-0782. doi: 10.1145/1506409.1506426.

Alex McLean and Geraint Wiggins. Petrol: Reactive pattern language for improvised music. In *Proceedings of the International Computer Music Conference*, June 2010.

Gregory L. Murphy. *The Big Book of Concepts (Bradford Books).* The MIT Press, August 2002. ISBN 0262632993.

Marian Petre and Alan F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51:7–30, 1999.

M. Puckette. The patcher. In *Proceedings of International Computer Music Conference*, 1988.

Julian Rohrhuber, Alberto de Campo, and Renate Wieser. Algorithms today: Notes on language design for just in time programming. In *Proceedings of the 2005 International Computer Music Conference*, 2005.

Sherry Turkle. *The Second Self: Computers and the Human Spirit, Twentieth Anniversary Edition.* The MIT Press, 20 anv edition, July 2005. ISBN 0262701111.

Sherry Turkle and Seymour Papert. Epistemological pluralism: Styles and voices within the computer culture. *Signs*, 16(1):128–157, 1990. ISSN 00979740. doi: 10.2307/3174610.

Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33, March 1992.

Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. Live algorithm programming and a temporary organisation for its promotion. In Olga Goriunova and Alexei Shulgin, editors, *read_me — Software Art and Cultures*, 2004.

G. A. Wiggins. A preliminary framework for description, analysis and comparison of creative systems. *Journal of Knowledge Based Systems*, 2006a.

G. A. Wiggins. Searching for computational creativity. *New Generation Computing*, 24(3): 209–222, 2006b.

PPIG2010

http://www.ppig2010.org