Usability of Programming Languages

MPhil in Advanced Computer Science
University of Cambridge Computer Laboratory

Lecture Notes – Lent Term 2016

Alan Blackwell

Introduction

This is a practical course, in which most of the learning will be achieved by the design and execution of original research experiments. The purpose of these lecture notes is to give a broad introduction to research in the field, both classic research (as collected in the Hoc, Green Samurçay and Gilmore book Psychology of Programming – henceforth called 'PoP'), and contemporary research currently being carried out in the major world centres.

Lecture 1 describes the theoretical principles that might be applied in your experiments, including the classic approaches covered in the PoP book, and also current trends in leading research. It provides an overview of the candidate research methods for experimental work, including their relative advantages and disadvantages, with references to those relevant chapters of the PoP book and of the Cairns and Cox book on Research Methods for HCI that provide more detailed introductions of specific methods.

Lecture 2 discusses the specific classes of user for which there are challenging issues in programming language usability. The so-called 'general purpose programming language' as a focus of computer science research has become relatively stabilised, but also serves a relatively small segment of the population. This lecture considers the larger populations that can benefit from more usable programming languages. The final part of this lecture will be directed by the specific research interests of the class. I include an outline of the topics that will be addressed.

Reading List

- ⇒ Online proceedings of the Psychology of Programming Interest Group (http://www.ppig.org)
- ⇒ Cambridge guidance for human participants in technology research (http://bit.ly/hptps-guide)
- ⇒ Cairns, P. and Cox, A.L. (2008) Research Methods for Human-Computer Interaction. Cambridge University Press.
- ⇒ Hoc, J.M, Green , T.R.G, Samurcay, R and Gilmore, D.J (Eds.) (1990) Psychology of Programming. Academic Press.
- ⇒ Carroll, J.M. (Ed) (2003). HCI Models, Theories and Frameworks: Toward a multidisciplinary science. Morgan Kaufmann.

Part 1: Principles of human factors in programming

Cognitive models of programming

When we consider the programming language as an 'interface' between the programmer and the machine, a convenient engineering approximation could describe the two sides of that interface in equivalent terms. In this view, the programmer has 'I/O subsystems' via which the interface is connected (eyes and hands) controlled by a 'central processor' (the brain) that includes both persistent and working storage (long term and short term memory). This convenient analogy between human and computer can be a useful tool for straightforward engineering purposes. In basic human-computer interaction, we can often measures of typical performance, for example speed and accuracy of hand movement, or spatial resolution and scanning time of the eye. We can also test the capacity and persistence characteristics of different kinds of memory – for example 7 +/- 2 verbal 'chunks' like names and digits, or a single detailed visual scene. This kind of data is sometimes described as 'human factors' or 'ergonomics' considerations. When we invent new kinds of interaction devices or user interfaces, it is possible to make some performance predictions on the basis of these mechanical analogies, and they make valuable contributions to usability analysis of conventional user interfaces.

In the case of programming, the relatively simple action sequences that have been the target of conventional HCI research are only one part of the problem. For most everyday user interfaces, it is relatively clear what the 'correct' sequence of actions should be, if you know what the user is trying to achieve, so a mechanical analogy to how machines might complete that sequence can be useful. However in the case of programming, much of the challenge comes from thinking about what you want to do. Improving the usability of systems for thinking is clearly more challenging than systems that only involve seeing, choosing and pointing. The necessary theoretical approach has been described as 'cognitive ergonomics'. This refers to the field of cognitive science (artificial intelligence is regarded as a subfield of cognitive science), which greatly extends the 'computational theory of mind' analogy that is implicit in simpler I/O oriented HCI.

The classic cognitive models of programming are recognizably derived from classic artificial intelligence / cognitive science research. In this view, programming can be described in terms of 'problem solving', 'planning' and 'knowledge representation', all of which correspond to well-established AI strategies and architectures. However, when these internal process descriptions become more complex it becomes harder to draw conclusions about them, given that (unlike AI systems) we can only directly observe external behaviour of human programmers. As a result, it is easy to fall back on the assumption that the 'internal' representation of the problem in the programmer's head corresponds quite closely to the 'external' representation of the program that he or she eventually creates.

With these critical cautions in mind, chapters 1.4 of the PoP book "Human Cognition and Programming" gives an overview of programming as a problem solving and planning activity, and chapter 3.1 "Expert Programming Knowledge: A Schema-based Approach" provide thorough descriptions of how cognitive/AI theories can be used to characterize the human reasoning processes that are involved in programming tasks. A more specialized view of cognitive problem-solving, derived more closely from observations of human reasoning than from computational simulations, is applied in chapter 2.3 “Language Semantics, Mental Models and Analogy”. The core insight in this tradition is that humans often solve problems by analogy to others that they have seen before. This applies to individual programming tasks (if you are asked to write a program where you can use a similar structure to one that you have written before, your previous solution would be a good starting point). But more dramatically, it applies to the understanding of programming languages themselves – the argument here is that even if you have no idea what programming is, you will understand it by analogy to things you’ve done before – perhaps natural language, perhaps mathematics, or perhaps Lego bricks. This latter kind of analogy to physical situations with similar structure is related to the principle of user interface “metaphor”, where system function is presented by analogy to folders and filing cabinets, or other physical apparatus for information processing.

Programming within the software development process

The cognitive science research tradition concentrates on individual humans who are solving problems in controlled contexts – often experimental psychology laboratories. This corresponds very well to the customary constraints of research programmes in AI (because robots are seldom competent to act in complex social situations such as city streets or dance halls). In the laboratory, the structure of the task, and the nature of the inputs and outputs, can be closely constrained to suit the capabilities of a robot, or of a cognitive theory. For the same reasons, most of the experimental investigations into cognitive models of programming (whether based on schemas, plans, or analogy) tend to have been focused on individual programmers, under observation in laboratories, addressing carefully constructed experimental tasks that probe hypotheses related to the specific model being investigated.

In contrast to these experimental situations, many of the situations where software development is of commercial relevance are more complex. They often involve as much understanding of the “problem domain” as they do of the programming language itself. There may be opportunities simply to avoid hard programming problems, by negotiating a change to the specifications. At an early stage of the project, there may be many different ways of formulating which problem is to be solved. These various tasks broaden the characterization of programming from a “problem-solving” activity to a “design” activity. Design theorist Horst Rittel described a class of “wicked problems”, in which conventional models of AI planning cannot be applied, because the goals and criteria for success are under-specified, the constraints conflict with each other, the resources are unknown or negotiable and so on.

A broader view of the tasks of programming is presented in Chapter 1.3 of the PoP book “The Tasks of Programming”. A great deal of progress in cognitive accounts of design has been made by PoP researcher Willemien Visser, who also contributed Chapter 3.3 of the PoP book “Expert Software Design Strategies”. The broader organizational context of software development requires whole theories of management science. This is outside the scope of this course – there is a whole academic field of “Information Systems” that deals with it. However a useful introduction, from a relatively familiar engineering perspective, can be found in Chapter 4.1 of the PoP book, "The Psychology of Programming in the Large: Team and Organizational Behaviour". There has been rapid change in software development methods since the PoP book was published, however. There are many recent studies published at the PPIG conferences that have explored new practices such as the pair programming that is popular in some agile software development methods. Pair programming is sufficiently well constrained that it can be studied in a controlled manner, unlike large software development teams in a complex organizational context. Studies of larger teams are more likely to be found in information systems or management research literature.

Individual variation

Cognitive theories of human behaviour are intended to be general theories. The experimental methods of experimental psychology (and of traditional HCI) are rather reliant on finding aspects of human performance that are consistent, so that an experimental sample will be representative of the wider population, and so that statistical arguments can be applied within a hypothesis testing context. However, even casual observation of professional programming contexts makes it clear that some programmers are far more productive than others. Furthermore, all programmers are more productive in a language they know than one they don't know. This makes it difficult to test modifications to existing languages and tools, because it is necessary to control for the previous experience of the individual programmers. Further complication comes from the fact that programming performance appears to be correlated with other psychometric variables, such as general intelligence, self-efficacy (personal confidence in one's own ability) and even some diagnostic tests for autism.

The most consistent interest in the PPIG research community, as in much traditional HCI, has been in the contrast between "expert" and "novice" users. These should be treated as technical terms, with care to avoid the potentially derogatory implications of calling somebody a "novice". The technical reference is to the psychology literature in problem-solving, which often tries to characterize the knowledge that is necessary to solve a problem by comparing experimental subjects who do know how to solve the problem (experts) to those who do not know how to solve it in advance (novices). In the PPIG context, the same technique is often used to study programming knowledge, via experimental comparisons of those who do have it to those who don't. In a controlled experiment, the "experts" might be people who have completed a training course in a programming language (say second year undergraduates) while the "novices" are people who have not (first year undergraduates). In the past, there was often great interest in studying people who had never seen any kind of programming language before, who had no expectations, or knowledge that might have 'crippled' or 'mutilated' their understanding of programming (an accusation made by Dijkstra against the languages BASIC and COBOL). These desirably virginal novices were often described as "naïve" users – another term that should be used carefully, because it would be derogatory outside a technical context. These methodological issues are discussed in chapter 1.5 of the PoP book.

However, there is also interest in studying people who are real experts, either to understand the nature of their expertise better, help other people to become expert, or provide tools that better support the needs of the expert practitioner. This kind of research into expertise is reviewed in

Chapter 2.1 of the PoP book: "Expert Programmers and Programming Languages". The author of the chapter Marian Petre, has also worked extensively in the study of expert designers in other technical fields, work that is published in the Design Research literature. (Your lecturer has collaborated with Petre in this area, and also publishes broader studies in design research).

Major research centres and programmes

Psychology of Programming research continues to be actively pursued and presented at the Psychology of Programming Interest Group (PPIG), which holds an annual international conference, and also an annual "Work in Progress" meeting (PPIG-WIP) for younger researchers and practitioners who with to present experience reports rather than full academic studies. The proceedings of the main PPIG conference are available online. The PPIG-WIP proceedings are not published. Research in the field was previously carried out under the auspices of the European Association for Cognitive Ergonomics (EACE), and the Empirical Studies of Programmers foundation (ESP). Representatives from all of these groups contributed to the PoP book. Since then, the remaining activities of ESP have effectively been merged with the IEEE conference on Visual Languages and Human Centric Computing (your lecturer convened an ESP symposium under that banner in Auckland 2003). The conference/workshop series on Program Comprehension (ICPC, formerly IWPC) is a parallel body that has been running nearly as long as PPIG. A more recent workshop series is the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). Psychology of programming research has always been a topic of peripheral interest at major HCI conferences such as the ACM CHI series, and some leading figures in PPIG/ESP are also leaders in HCI/CHI. The conference series on Evaluation and Assessment in Software Engineering (EASE) tends to focus on larger-scale issues, but does include a number of publications describing studies of programming languages and features. The same applies to meetings of the International Software Engineering Research Network (ISERN). The International Symposium on End-User Development is a regular series that focuses on the needs of non-professional programmers (see later in these notes). Smaller meetings are regularly convened in association with programming technology conferences – for example Cooperative and Human Aspects of Software Engineering (CHASE) has been organized in association with the International Conference on Software Engineering since 2011, or the Onward! Symposium on New Ideas in Programming and Reflections on Software, held in association with the SPLASH (previously OOPSLA) conferences.

In these notes, the research field as a whole is described for convenience as PPIG. However, readers should be aware that PPIG itself is simply the longest-established venue in the field (and one that has conveniently published a textbook, and made its research archives freely available online). Many of the individual researchers described below would choose other venues as representing their primary community.

⇒ PPIG organization and conferences - http://www.ppig.org

⇒ PLATEAU workshop series - https://sites.google.com/site/workshopplateau/

⇒ CHASE workshop series - http://www.chaseresearch.org/home

⇒ VL/HCC - http://conferences.computer.org/VLHCC/

⇒ ICPC - http://www.program-comprehension.org/

⇒ IS-EUD - http://dei.inf.uc3m.es/iseud2015/

⇒ History of ESP - http://www.ppig.org/news/2006-06-01/whatever-happened-empirical-studies-programmers

⇒ EACE - http://www.eace.net/

⇒ EASE - http://www.scm.keele.ac.uk/ease/

⇒ ISERN - http://isern.iese.de/Portal/

⇒ CHI - http://www.sigchi.org/

The largest programme of funded research in recent years has been the EUSES consortium (End-Users Shaping Effective Software), funded by the American National Science Foundation. This was managed by Margaret Burnett at Oregon State University, with collaborators at Penn State (led by Mary Beth Rosson), Carnegie Mellon (led by Brad Myers), Drexel (led by Susan Wiedenbeck), Nebraska (led by Gregg Rothermel) and Cambridge (your lecturer).

⇒ EUSES consortium http://eusesconsortium.org/

⇒ Margaret Burnett http://web.engr.oregonstate.edu/~burnett/

⇒ Mary Beth Rosson http://mrosson.ist.psu.edu/

⇒ Brad Myers http://www.cs.cmu.edu/~bam/

⇒ Susan Wiedenbeck http://www.cis.drexel.edu/faculty/wiedenbeck/index.html

⇒ Gregg Rothermel http://cse.unl.edu/~grother/

In the UK, senior members of the PPIG research community are currently active at Salford (Maria Kutar – PPIG chair), Sheffield Hallam (Chris Roast), the Open University (Marian Petre and Judith Segal), Sussex (Judith Good and Ben du Boulay) and Cambridge. There are strong groups in Finland (Jorma Sajaniemi, Markku Tukiainen and Roman Bednarik at Joensuu), Ireland (Jim Buckley at Limerick), Paris (Francoise Detienne and Willemien Visser at INRIA), as well as many smaller groups that conduct occasional research projects in the field. There was a European 'Network of Excellence' on the theme of End-User Development, that had close links to the EUSES consortium. Active members included Volker Wulf (Fraunhofer Institute/University of Siegen), Yvonne Dittrich (IT University of Copenhagen) and Fabio Paterno (ISTI Pisa, the network convenor).

⇒ Maria Kutar - http://www.business.salford.ac.uk/staff/mariakutar

⇒ Chris Roast - http://extra.shu.ac.uk/crr/

⇒ Marian Petre - http://mcs.open.ac.uk/mp8/

⇒ Judith Segal - http://mcs.open.ac.uk/jas583/

⇒ Judith Good - http://www.informatics.sussex.ac.uk/users/judithg/About_Me.html

⇒ Jim Buckley - http://www.csis.ul.ie/staff/JimBuckley/

⇒ INRIA Eiffel group - http://www.inria.fr/en/teams/eiffel

⇒ Jorma Sajaniemi - http://cs.joensuu.fi/~saja/

⇒ Volker Wulf's group http://www.wiwi.uni-siegen.de/wirtschaftsinformatik/mitarbeiter/wulf/index.html.en

⇒ Yvonne Dittrich - http://www.itu.dk/people/ydi/

Other active centres in the USA include the University of Colorado at Boulder (Clayton Lewis, Gerhard Fischer and Alexander Repenning), MIT Media Lab (Henry Lieberman), IBM Research (Rachel Bellamy at TJ Watson, Allen Cypher at Almaden), and the Human Interactions in Programming (HIP group) in Microsoft Research at Redmond (Rob DeLine, Gina Venolia and Andrew Begel). Younger researchers who did their PhDs in these and related groups, are still active in the field, but are now based at other locations include Rob Miller, Chris Hundhausen, Andrew Ko, Laura Beckwith and many others.

⇒ Gerhard Fischer - http://l3d.cs.colorado.edu/~gerhard/

⇒ Alexander Repenning - http://www.cs.colorado.edu/~ralex/

⇒ Henry Lieberman - http://web.media.mit.edu/~lieber/

⇒ Rachel Bellamy - https://researcher.ibm.com/researcher/view.php?person=us-rachel

⇒ Allen Cypher - https://researcher.ibm.com/researcher/view.php?person=us-acypher

⇒ Microsoft HIP group - http://research.microsoft.com/en-us/groups/hip/

⇒ Andy Ko - http://faculty.washington.edu/ajko/

Current areas of theoretical attention

The relationship between theory and research in HCI is complex, as discussed in Chapter 9 of the Research Methods for HCI book. A great deal of HCI research does not in fact involve any explicit theory at all – or if it does, those theories are borrowed from other disciplines (psychology or sociology). The same is true of much PPIG research. In the past (as can be seen in the PoP book), cognitive science theories were considered central, and the theory was expressed in the form of cognitive models, of the kind described in chapter 4 of the Research Methods in HCI book, and the chapter on GOMS in the Carroll book. The Carroll book contains extended descriptions of most of the substantial theoretical perspectives in HCI. In principle, any of these could be applied to programming languages and environments. However there is only one that emerged explicitly from this field.

Cognitive Dimensions: The most influential framework in the field is the Cognitive Dimensions of Notations framework, originally proposed by Thomas Green at the Cambridge Applied Psychology Unit, and developed with substantial input originally from Marian Petre, and then by your lecturer with various collaborators. The most widely cited publications are a tutorial developed by Green and Blackwell, a short-form evaluation questionnaire by Blackwell and Green, and an analysis of visual language usability by Green and Petre in the Journal of Visual Languages and Computing. A special issue of JVLC, published ten years after that paper, reviewed subsequent developments, including extensions of the framework outside the programming context to tangible user interfaces and collaboration tools. All researchers in the PPIG field must be aware of the CDs framework, but it might be an exaggeration to describe it as a 'theory'. This was certainly not the intention of Green (he says), since he presented the framework as an informal source of advice for designers rather than a new piece of cognitive science. Nevertheless, the framework does have a clear theoretical motivation, based on Green's conception of programming as interaction with an information structure (Chapter 2.2 of the PoP book – although this does not specifically mention CDs, which was only nascent at the time).

The basic principle is that any visible 'notation' (in fact, any 'information artefact', whether a programming language, a design notation, a recipe, novel, or a sales chart) encodes an information structure. The information structure is considered to have different parts (which may be components, modules, elements, entries and so on). These parts have a variety of relationships to each other (membership, dependency, reference, equivalence, subsidiarity …). Visible notations can also be analysed in terms of their graphical elements, and the graphical relationships between those elements (see the entry in the online Interaction Design encyclopedia on visual representation). Notations are used and interpreted in terms of the correspondence between the visual structure and the information structure that it represents.

⇒ http://www.interaction-design.org/encyclopedia/visual_representation.html

The individual dimensions such as *viscosity* (quick definition: a viscous system is one that is difficult to change) are defined in terms of the relationship between what the user needs to do to the information structure, and the tools that are provided to make the corresponding change to the notation. The full set of dimensions is described elsewhere – in these notes I elaborate on a couple of more subtle theoretical issues.

Notational Layers: It is often the case that one information structure is derived from another, such that the parts and relationships in one of them can be viewed as arising from the parts and relationships in the original. For example, the structure of an e-commerce web page is related to the structure of the database application that generated that web page, which is related to the structure of a design model for that application, which is related to the structure of the ideas in the designer's head, which is related to the structure of the company that commissioned the design. It would be possible to express each of these information structures in a different visual notation (a screen shot, a PHP program, a UML diagram, a whiteboard sketch, a business plan respectively). These can be described as 'layers' of notation that combine to make up the whole problem.

Notational Activities: The critical question for usability is what the user needs to do with this information structure. Simply finding pieces of information within a familiar structure is quite easy. Trying to understand the relationships in a structure you haven't seen before is more complex. Adding new pieces of information, if their relationship to the existing structure is similar to other parts, may be easy. Changing the structure is likely to be more difficult. Creating a new structure may be relatively easy if it is being derived from another layer, so that the new parts and relationships are defined in terms that can be anticipated on the basis of the existing structure. Creating a new structure when you don't know beforehand what the appropriate parts and relationships are is most difficult of all. These different activities are described in the CDs framework as 'search', 'exploratory understanding', 'incrementation', 'modification', 'transcription' and 'exploratory design' respectively. There are many aspects of programming work that can be related to these different activities. A recent paper by Blackwell & Fincher considers CDs in terms of the user experiences that may be associated with notational activities, by analogy to the 'design patterns' introduced in Architecture by Christopher Alexander, and popularised in computer science by various authors.

The developing theoretical integration between the idea of the information structure and the way this idea can be applied as a design tool is further explored in the chapter by Blackwell & Green in the Carroll textbook, and Chapter 8 of the Research Methods for HCI book by Blandford & Green.

⇒ A variety of material related to CDs can be found from the following resource site: http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/index.html

Attention Investment: A programme initiated by Blackwell to create a more rigorous theoretical characterization of interaction with information structures led to the development of the "Attention Investment Model of Abstraction Use", which has been extensively applied in Cambridge, at Microsoft, and at Oregon State University. The main focus of Attention Investment is to compare the amount of mental effort (for example, focused concentration) that is required to carry out a programming task, to the amount of effort that would be saved (in terms of automation) once the program has been created. This can be described as a cost-benefit equation. However, there is a degree of risk associated with both the costs and the benefits. It can be hard to anticipate the actual effort that will be involved in getting a program working. The benefits are also uncertain, for example if the program has a bug – in fact a severe bug might result in the program causing even more harm than it does good, so the net return on the investment is negative!

The attention investment model predicts why people might be reluctant to engage in programming, either because they over-estimate the costs involved, or over-estimate the risks of a negative return. Some professional programmers, in contrast, under-estimate the costs, and over-estimate the benefits. These kinds of bias can be understood in terms of the heuristics by which humans make decisions on the basis of their previous experiences (as with problem-solving by analogy, decision by heuristic biases is a better model of human reasoning than AI systems that tend to have relatively little prior knowledge of the world, so reason from first principles).

Well-designed programming environments should help reduce the two kinds of error that can result from heuristic biases. One of the design objectives motivated by Attention Investment has been described as a 'gentle slope' for programming tools, making simple things simple to do, with gradually increasing difficulty for more complex tasks. Many systems fail to achieve this, so that users face a 'cliff' of complexity when they need to do something slightly more complex (for example, the transition from cell formulae to macro programming in Excel). Another application developed by Margaret Burnett and colleagues at OSU is the 'surprise, explain, reward' strategy to encouraging testing and debugging.

⇒ 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. http://www.cl.cam.ac.uk/~afb21/publications/HCC02a.pdf

Gender HCI: Collaboration with Laura Beckwith at OSU led to attention investment being integrated with self-efficacy theory to explain some gender differences in programming. That programme of research continues under the name of "Gender HCI", although its primarily cognitive orientation sits uncomfortably alongside more recent research by Bardzell and others that applies gender studies in an HCI context. Some work to reconcile the two can be found in the work of Jennifer Rode, who initially worked on this in Cambridge, and is now based at Drexel with Susan Wiedenbeck.

⇒ Gender HCI - http://eusesconsortium.org/gender/gender.php

Programming by Example: There has been a long tradition of applying inference or machine learning techniques to develop systems that can infer programs from examples of the required output. This research is collected in two books, the first edited by Allen Cypher and the second by Henry Lieberman. Ongoing research in Cambridge is exploring the relationship between this work and Attention Investment.

⇒ Watch What I Do (Ed. Cypher) http://acypher.com/wwid/

⇒ Your Wish is My Command (Ed. Lieberman) http://web.media.mit.edu/~lieber/PBE/Your-Wish/

Natural Programming: Myers' group at Carnegie Mellon have carried out a programme of study to describe "natural programming", by which they mean ways of describing algorithms and data structures that are meaningful in natural language and everyday usage. The intention is that the results should allow programming languages to be designed based on naïve or novice understanding, rather than requiring more expert training. Some prototype systems based on this idea have been developed by John Pane, Rob Miller, Andrew Ko and Christopher Scaffidi.

⇒ Natural Programming Project - http://www.cs.cmu.edu/~NatProg/index.html

Variable Roles: There have been a number of productive studies in which aspects of programmer behaviour are inferred from analysis of source code corpuses, in addition to watching programmers at work. Research in Finland developed the conception of 'variable roles', as a way of characterizing local programming strategies of using variables. These have become useful as an inspiration for new design and visualization tools, and also as a guide for educators. Many of the original publications can be found in proceedings of the PPIG workshops.

⇒ Roles of Variables home page - http://www.cs.joensuu.fi/~saja/var_roles/

Collaboration and agile programming has been a significant focus of research at the PPIG workshops in recent years. Experimental observation and analysis of the interaction between people doing pair programming has been a particularly productive area of attention. At present, this work has not been so directly related to the design of programming languages themselves, and the theoretical orientation tends more toward sociology rather than psychology. Those topics are out of the scope of the current course, but those who are interested can learn more by exploring the field of Computer-Supported Collaborative Work (CSCW) rather than HCI. Yvonne Dittrich (Copenhagen) and Helen Sharp (Open University) are among the leaders in this field.

Aptitude: The challenge of how to identify good programmers is of perennial concern at PPIG and elsewhere. This is relevant to commercial contexts, of course, where good programmers are commercially valuable, but not always easy to identify. It is also of interest to the academics who write papers for PPIG, because they want to identify which students will show most talent, or are most likely to need additional help. Measures of programmer aptitude are sometimes presented without any serious theoretical explanation, but they can also build on a range of psychometric characterisations of individual differences, such as cognitive style, personality measures, or even diagnostic tools for autism spectrum conditions.

Development in organizational contexts: The Microsoft HIP group have carried out a substantial number of long-term studies of professional programmers working in realistic team contexts, and maintaining code bases on an industrial scale. This kind of research is generally beyond the resources of academic research budgets, and relies on access to commercially sensitive information. Despite this necessarily specialist community, the group engage actively with academic researchers, and share their results widely.

Syntax and tools

As can be seen from the above examples, many of those who study the usability of programming languages also develop new languages. In the past, programming was seen as an interesting object of study in its own right, for example in the work of Green at the Applied Psychology Unit. However, now that Cognitive Neuroscience has replaced most experimental psychology research, it is difficult to conduct pure research into higher cognitive functions such as programming. As a result, empirical studies of programmers are now carried out in computer science departments, where there is also more desire and capacity to develop new experimental tools (even if many of the senior researchers may have qualifications in psychology or cognitive science, so that they are relatively unlikely to develop tools themselves). This means that most active research groups have specialized interests not only in particular theories of human cognition and behaviour, but also in particular kinds of language syntax, or particular kinds of software development tool. Some of the most popular are:

Integrated development environments: Professional programming relies on availability of a larger software environment that manages project modules, integrates editors with compilers, provides debugging tools and so on. Some research is conducted by creating custom plug-ins for IDEs such as Eclipse, but it seems that the novel interaction styles of most interest to programming usability researchers are hard to achieve within the Eclipse architecture. Popular educational tools such as BlueJ or Scratch have simplified IDEs at a level appropriate for their intended audiences, but the effort of maintaining these for a large user base means that there is little remaining resource to carry out experimental modifications. One exception is the CMU Alice project, which Ko used as the experimental target for his 'WhyLine' debugging aid, and Kelleher extended to explore the use of teaching strategies that incorporate storytelling. Burnett's Forms/3 has been developed over many years, with a number of experimental extensions, but has not been deployed outside the research context.

Visual languages: The concept of a visual language is an old one, dating from ideas to make executable flow charts, to Sutherland's object-oriented graphical constraint system Sketchpad in the early 60s, and David Smith's Pygmalion in the 70s. Although much of this research was motivated by the goal of improving usability, the idea of measuring or assessing the improvement did not become well established until 1996, when keynote speakers at the annual VL conference were HCI authority Ben Shneiderman, and Thomas Green, reporting both his recent Cognitive Dimensions work and his studies of flow charts dating back to the 1970s. Your lecturer also presented a critique of the (sometimes mistaken) implicit psychological assumptions that had driven the field until then.

As noted above, there are now many visual languages, in education contexts and elsewhere. Pioneering commercial products were the National Instruments LabVIEW system, and Prograph, a commercial spinout from VL research by Philip Cox and Trevor Smedley at Dalhousie. New visual languages are now being announced at a rapid rate, although often described in ways that suggests the marketing people have never heard of the idea before. Interesting recent examples include Yahoo Pipes, Microsoft Kodu, and Google Blockly. All of these are relatively straightforward adaptations of previous academic systems, although few have benefited either from sophisticated evaluation of the underlying theoretical assumptions, or application of the Cognitive Dimensions framework.

Spreadsheets: The most widely used programming technology at present is the spreadsheet. There have been many empirical studies of spreadsheet users, both using conventional spreadsheets (Excel) and Margaret Burnett's Forms/3, which allows cells to be arranged in a free format. These studies have led to a wide range of usability improvements to spreadsheets, including testing and debugging facilities, and type systems in the work of Erwig. Spreadsheets were used as the experimental target for a number of experiments in Gender HCI, which led to the characterization of 'tinkering' as a kind of programming behaviour that can be beneficial for those with low self-efficacy, although problematic for users (often male) who are over-confident. Burnett and Blackwell, with Haskell architect Simon Peyton-Jones, designed a functional programming extension to Excel that allowed patches of spreadsheet to act as first class functions. Felienne Hermans at the Technical University of Delft has an impressive blog and video series presenting recent research on these topics.

⇒ http://web.engr.oregonstate.edu/~erwig/units/

⇒ http://research.microsoft.com/en-us/um/people/simonpj/Papers/excel/excel.pdf

⇒ http://www.felienne.com/

Scripting languages: Many software users benefit from the capability to customize their tools, and many advanced tools include facilities to let them do so, with scripting or macro languages. Familiar examples include the use of Visual Basic in Microsoft Word, but more specialist professional examples include LISP variants – in AutoCAD and the programmer's editor EMACS. Scripting languages often turn up in unexpected places, such as the programming tools that can be used to create new behaviours in Second Life. Some sophisticated scripting capabilities come and go, such as Apple Hypercard and Automator. Computer Lab researcher Luke Church is the principle architect of a powerful new experimental scripting environment from AutoDesk called DesignScript.

Research by Tessa Lau and Allen Cypher at IBM resulted in CoScripter, an intriguing system for scripting repeated sequences of web navigation and form actions, extending programming by example techniques. A future growth area may be the development of scripting languages for home automation. Many of these systems are in principle intended to be accessible by users who do not have professional programming training. This introduces both educational and usability challenges – this topic will be discussed later in the course.

⇒ http://designscript.org/

⇒ http://acypher.com/coscripter/

Part 2: Research methods in the study of programming.

Ethical issues in research

All academic research involving human participants must consider any possible ethical concerns. Detailed guidance has been compiled for research carried out in the Cambridge School of Technology. This guidance is constantly extended and refined – please contribute any useful observations that you might have to the site maintainers (coordinated by your lecturer). The experiments that you will be carrying out for the practical element of this course must be reviewed by the Computer Lab ethics committee.

⇒ Cambridge Technology Ethics guide - http://bit.ly/hptps-guide

⇒ Computer Lab Ethics committee http://www.cl.cam.ac.uk/local/committees/it-strategy/ethics.html

Controlled experimental methods

The material in the remainder of this section has been rendered largely redundant by the publication of an excellent paper with detailed advice on controlled experiments of software engineering tools with human participants. A copy of the following paper can also be found on the course materials page:

⇒ Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empirical Software Engineering. 20, 1 (February 2015), 110-141. http://dx.doi.org/10.1007/s10664-013-9279-3

The rest of this section can be consulted for interest or comparison, but I recommend the Ko et al paper as a more comprehensive treatment of the topic.

The classical cognitive approach to study of programming language usability uses controlled experimental methods, in which a sample of 'participants' (or 'subjects' in old terminology) completes an experimental 'task' while their 'performance' is measured – typically in terms of speed and accuracy. (Comparison may be easier if only correct results are considered). Participants may complete a number of trials, each involving a different task. Different experimental 'conditions' involve manipulating the task in different ways – typically by modifying the programming language, using different languages, or different features of the programming environment. The 'effect' of those modifications can be assessed by comparing performance. This must be done statistically, preferably 'within subjects' (each participant completes tasks using all versions of the programming language), but if necessary 'between subjects' (some participants use one version, and some use another).

This course assumes that you have had previous experience in the design and analysis of simple hypothesis-testing controlled experiments with human participants. The basic approach used in HCI experiments (comparing speed and accuracy in two different conditions with alternative technical designs) is directly applicable to simple experiments in the usability of programming languages. However, this approach requires that a relatively straightforward experimental task can be identified.

⇒ A more detailed review of basic principles in experimental design is provided in the PoP book, Chapter 1.5: Methodological Issues in the Study of Programming.

Typical experimental tasks

Classic approaches to the study of natural language consider both production tasks (speaking or writing) and comprehension tasks (understanding, interpretation or recall). Experimental studies in psycholinguistics often measure only one of these at a time – the same is true in many studies of computer programming languages. A combination of speed and accuracy seems to be directly relevant to production (write a program that is correct, and write it quickly), while accuracy and completeness, rather than speed, are more relevant to comprehension.

However, application to real world situations must recognize that competent language use involves both production and comprehension. In the case of spoken natural language, this might involve exchange with a conversational partner, but programming more often involves reading back and modifying code that you have written yourself. This introduces the need for search tasks – finding the place in the code that is responsible for a particular piece of functionality, or that must be modified to correct a bug or add new behaviour. As I write this, it occurs that the experimental tasks in PoP might reasonably be classified in terms of the six types of notational activity defined in the CDs framework. As far as I am aware, this has not yet been attempted.

In conventional experimental psychology, standardized tasks are used as much as possible, in order that the results of one experiment can be compared to another. A classic problem solving task is the 'Wason selection task', and a classic planning task is the 'Towers of Hanoi' task. Each of these is useful, from a cognitive science perspective, because they are well-formed ('toy') problems where the correct solution is easily expressed as a computer algorithm.

⇒ http://en.wikipedia.org/wiki/Wason_selection_task

⇒ http://en.wikipedia.org/wiki/Towers_of_Hanoi

In the early days of ESP/PPIG research, specific kinds of experimental tasks were used in multiple studies. Your lecturer compiled a list of the tasks that might be considered. These are a useful reference source to see what kinds of task granularity might be used in a successful experiment. Of course most of these tasks share the disadvantages of 'toy problems' in broader cognitive science – that they do not often resemble real programming problems that a professional programmer might encounter (although some of them do resemble student exercises). This presents a problem of 'external validity', if you want to make claims that your results are relevant to real programming. However, poor external validity is often associated with good internal validity, as a general characteristic of experimental research, so this is a trade-off that you may have to make.

⇒ Section 2 of Chapter 1.5: Methodological Issues in the Study of Programming provides further discussion of experimental tasks.

⇒ Blackwell's list of PoP tasks - http://www.cl.cam.ac.uk/~afb21/poptasks.html

Experimental manipulations of programming tools

If you wish to study the effect of a particular feature in a programming language or environment, the most straightforward controlled comparison would be to compare a version with that feature to another version without it. For some cases, it may be relatively straightforward to create two versions of a new prototype, one that is complete, and one that has a crucial aspect disabled. However, this strategy introduces a number of practical problems. Is it possible to make a version that works without the new feature? Will the experimental task be meaningful if the feature is disabled? In a within-subjects comparison, the experiment may seem illogical to participants unless the 'improved' version with the new feature is presented in the second trial, which means that the utility of the feature is conflated with an order effect (a problem of internal validity). Finally, if your experimental system has been created specifically in order to support this feature, then it may be comparatively poorly designed in other respects. As a result, the comparison to other existing systems may not be fair, because performance with your experimental system will not be representative of typical systems of the kind (a problem of external validity).

Despite these problems, direct comparative studies of specific features can be valuable research contributions, especially to estimate the productivity gain (experimental effect size) that could result from a new invention. A more challenging ambition is to manipulate programming tools in order to investigate some research question related to more fundamental debates among advocates of different approaches to programming. Classical debates of past years have included the debate between advocates of imperative and declarative programming paradigms, or between textual and visual syntax. The problem here is that it is very difficult to create two languages that are properly representative of the two alternatives, yet are also equivalent in other respects. Furthermore, even if this has been achieved, it is hard to design experimental tasks that are equally suited to different paradigms. As a result, attempts to settle this kind of debate via controlled experiments with good internal validity have pretty much been abandoned. Fortunately, other study techniques, many with better external validity, are still available.

The sheer complexity of programming tools provides a further obstacle to experimental manipulation. There is the straightforward problem that a conceptually simple user interface improvement may be computationally infeasible, or require years of development effort. A slightly more subtle problem is that an existing system may provide so many essential features that it is not feasible to duplicate them all to a sufficient level of functionality to support a realistic experimental task. In conventional HCI research, it is normal to 'cheat' by evaluating paper prototypes or screen mockups that simply simulate the appearance of the working system. A Wizard of Oz (originally a 'man behind the curtain' as in the movie) can manually simulate the system behaviour that would result in response to user actions. However, this manual simulation is seldom feasible in programming research. One alternative proposed by Blackwell et al. is that a mockup of the new feature can be overlaid on an existing product in a way that simulates a proposed modification (that paper describes simulating the appearance of a rather fundamental change to Excel, by pasting small gif images into the cells of a spreadsheet).

⇒ Blackwell, A.F., Burnett, M.M. and Peyton Jones, S. (2004). Champagne Prototyping: A research technique for early evaluation of complex end-user programming systems. In Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC04), pp. 47-54.

Measurement, observation and protocol analysis techniques

Classical controlled experiments in psychology and HCI measure **speed** (reaction time) and **accuracy** (number of (non)errors). In the case of a production task, we might record how long participants take to produce or modify a program after being given a specification, and whether the resulting program or modification is correct. In simple reaction-time experiments, there is often a trade-off between speed and accuracy, and the purpose of measuring both is to compensate for that trade-off. In programming usability research, it is more often the case that poor performance is characterized by both low accuracy and long task completion times. This generally makes analysis easier, because two measures are available as estimators of task complexity. Nevertheless, correlation between speed and accuracy should usually be tested, to confirm whether this is the case.

Task completion: For practical reasons, it is sometimes necessary to stop an experimental task after a fixed amount of time, rather than waiting for participants to complete it. If this is the case, then an alternative speed measure can be derived from the proportion of the overall task that was completed, for example, number of bugs fixed, or number of function points implemented. If a wide range of performance is expected, this can be useful for management of experiments – it is not desirable for experimental sessions to last longer than 1-2 hours. However, repeating large numbers of relatively trivial tasks may be uninteresting for experimental participants, and also provide relatively poor external validity.

Subjective **self-report** of ease of use can be a useful research tool. This is justified on both applied and theoretical grounds. The applied justification is that new technology products must ultimately appeal to market consumers (whether commercial customers, or a research audience). If a new language seems easy to use at first sight, it is more likely to be successful in the market, so assessing this at an early stage in research can provide valuable information. The theoretical justification is that reported ease of use may provide an estimate of factors such as 'cognitive load' that are detrimental to task performance.

However, there is significant danger of bias in self-report measures. Subjective impressions of task performance do not always correspond accurately to real task performance. Furthermore, they are highly subject to 'experimental demand' – people who have volunteered to participate in experiments want to be helpful to the experimenter, and this can include overly generous feedback on the quality of the experimenter's work. Because they want to be nice, they will often respond to questions about desirability and ease of use in a way that is biased in favour of the experimental system. In order to avoid this, it can be useful to disguise which manipulation is the novel one, or for the experimenter to distance him or herself from the system (e.g. "We are trying to evaluate this new system that someone has proposed, but we don't know whether it is any good or not").

Despite the well-established techniques for controlled comparison of human performance, there are many important aspects of programming that are hard to measure quantitatively. Speed and accuracy are certainly of interest to programming language users, but they are even more likely to be concerned with whether a new language helps programmers to understand their problem better, allows them to be more creative, makes people more enthusiastic to learn programming and so on. All of these questions can be formulated in self-report questionnaires ('on a scale of 1 to 5, how creative would you say you are feeling now'), but this approach is highly subject to bias, and poor reliability. An alternative is to collect 'richer' data by observing people as they use programming systems, and asking them to talk about their experiences. Observations might consist of screen-capture from programming sessions, perhaps supplemented with video recording (or entries in an observer's notebook) to capture use of reference materials other than screen content.

A common strategy is to ask participants to **think-aloud**, describing everything that passes through their mind as they are carrying out the task (sometimes called 'concurrent verbalization'). However, this has two problems. One is that, if the task is hard, it becomes even harder if you have to do it while speaking aloud. The other is that during the most intense periods of problem solving or insight, participants simply stop talking, so you fail to capture information about the periods that are of most interest. One way around this problem is to make a screen recording or video recording of the experimental sessions, then play this back to the participant who makes a retrospective verbal report of what they were doing.

It is also possible to use **eye-tracking** to gather more information about which part of the screen the programmer is looking at as they complete the task. This can either be processed automatically, to infer information about the task strategy, or the eye-tracking trace can be used for retrospective verbalization. Bednarik in Finland has reported on a series of eye-tracking studies at PPIG workshops. Similar techniques can be used without an eye-tracker, by blurring parts of the screen and requiring the user to mouse over them. This 'restricted focus view' (RFV) approach was developed by Blackwell & Jansen, and has been used in studies of programming reported at PPIG and elsewhere by Romero and Cox.

⇒ Chapter 4 of the research methods in HCI book: 'Eyetracking in HCI' – gives a great deal more information about the practicalities of using eyetracking methods.

⇒ Bednarik's comparison of RFV to eyetracking: http://www.ppig.org/papers/16th-bednarik.pdf

If rich, verbal, non-numerical or 'qualitative' data has been collected, then the data analysis process becomes a far more central element of the research. The usual starting point is to transcribe the recorded data, so that you have a record of every word the participant said – probably correlated with the aspects of the task that they were working on at the time. This is described as a 'verbal protocol'. (It is also possible to make detailed analyses of video recordings, which are then called 'video protocols'). There are two broad strategies for analysis. The first is to treat the protocol as representing those aspects of human behaviour about which you had a prior hypothesis. A '**coding frame'** is created, defining the different categories of behaviour that you are concerned with. The protocol is segmented into episodes, utterances or phrases, and each of these is classified into a relevant category. The behaviour of different groups of participants, or in different experimental conditions can then be compared statistically. It is also reasonably common to use this technique to study the time course of how people approach tasks, comparing the frequency or order in which different episodes occur over time (as in work by Pennington). Interpretation of episode categories can often be ambiguous, and is subject to experimenter bias, so it is important to recruit multiple independent 'raters', and carry out 'inter-rater reliability analysis'.

An alternative strategy for the analysis of qualitative data is **grounded theory**, in which the researcher starts by carrying out 'open coding', observing patterns in the verbal data as he or she goes along. This is even more interpretive, and potentially open to bias, so a carefully controlled analytic process is recommended. Open coding is followed by a sequence of thematic grouping and generalization strategies, undertaken together with 'constant comparison' to ensure that the interpretations being developed are still compatible with the actual words used by participants.

⇒ Chapter 7 of the research methods in HCI book: 'A qualitative approach to HCI research' – gives a good introduction and overview of the ground theory approach.

Experiment design

The 'experiment design' in a controlled experiment refers to the combination of participants (perhaps in groups), tasks (perhaps in blocks of trials), conditions and measures, and the hypothesized effects of the manipulation conditions on your chosen measures.

Statistical **significance testing** requires you to demonstrate that the difference in means that you observe between the two groups or manipulation conditions is unlikely to occur by chance. In order for this to be shown, the size of the difference between the means for each condition (the effect size) should be relatively small when compared to the variance within each group. This is the primary reason why within-subjects experiment designs are preferred in psychology of programming tasks, because there is so much variation between people in programming performance. In a between-subjects design, that individual variation will almost certainly be large relative to your effect size, so a statistically significant result becomes unlikely.

However, there is a major challenge in the use of within-subjects designs for psychology of programming. 'Order effects' mean that whichever condition the participant carries out second will benefit from the fact that they have learned how to use the system, so will appear to be faster and more accurate. A further order effect results from task familiarity – you cannot ask participants to carry out the same programming task twice, because they will already know how to do it. You can use a different task in each condition, but it is very hard to calibrate tasks so that they are precisely equivalent, without actually being the same problem. It is therefore necessary to 'balance' the different experimental conditions with order, and with task, so that each version of the programming language is used with each task, and each combination is presented in both the first and second position in the experiment. For two conditions, two tasks, and two orders, a 'latin square' balanced design requires multiples of four participants.

⇒ Chapter 1 of the research methods in HCI book describes experiment design and latin squares.

Experiments should always be designed with an understanding of how the data is going to be analysed. If at all possible, you want your quantitative data to be normally distributed, so that you can make statistical tests using a t-test, ANOVA, or Pearson correlation. If it seems in advance that this might not happen, it might be wise to consider a different design. Distributions of task completion times are often skewed, with a 'long tail' of a small number of individuals who complete the task quite slowly. In traditional psychological experiments, those individuals are sometimes excluded form analysis as 'outliers' who are not of interest because they are atypical. However, in programming, we often observe that some individuals have a lot of difficulty with programming tasks – we would like to create systems that benefit them, not exclude them. For this reason, it can be preferable to use a log transform of time values, which are usually found to be normally distributed for human reaction times, and make the outlier values in the tail of the distribution more directly comparable to the rest of the population.

Subjective preference ratings are almost never normally distributed. In this case, a chi-square test, or a 'non-parametric' comparison of means must be used to test whether two conditions or groups are significantly different.

⇒ Chapter 6 of the research methods in HCI book: 'Using statistics in usability research' - provides a more thorough discussion of these issues

An alternative approach to the study of user interfaces is simply to 'evaluate' or 'explore' the usability of a system. The findings from **evaluation** or exploration studies can help inform the design of programming languages and environments, either in a 'formative' way (a study carried out early in the design process, in order to choose between or identify new design options) or a 'summative' way (identifying usability problems in a system you have already built). In conventional user-centred design processes, user studies are carried out within an iterative design process, allowing a system under development to be successively refined on the basis of evaluation or exploration results. However these kinds of study are relatively weak contributions to research literature, because they do not usually make any direct contribution to theory. The results can be of relevance to the specific product under development, but may not be more generally relevant to other research in future. Of course, the same considerations mean that these kinds of study are relatively popular in commercial contexts.

⇒ Chapter 1.5 of the PoP book discusses these issues further.

One important proviso for your future research careers is the significance of evaluation when publishing in broader technology research venues. There are many conferences for which the apparent purpose is to improve the efficiency or effectiveness of software development (e.g. ICSE, OOPSLA/SPLASH etc). Many presentations at these conferences propose new tools or methods that are claimed to result in improvements. Some of those presentations make their claims without any evidence to support them. This has been considered acceptable until recently, but it is increasingly common for papers at these conferences (especially the prize-winning papers at ICSE that are written by EUSES members) to include evidence from evaluation of the new tools. This trend is likely to increase, as the tendency at other conferences in the past has been for evidence-based research to drive out purely technical demonstrations. Sometimes, new conferences emerge to host research by people who do not wish to get involved in evaluation (e.g. the VLC conference that was created in reaction to VLHCC – they just removed the 'human-centred' part of the name!). However, an orientation toward claims without supporting evidence tends to result in such venues having a relatively poor reputation.

⇒ ICSE - http://www.icse-conferences.org/

⇒ OOPSLA/SPASH - http://splashcon.org/

Field study methods

If one wishes to study the organizational context of software development, or the way that software development teams interact with each other, or even realistic behaviour of individual programmers in the actual contexts where they work, it becomes necessary to go to them, rather than bring them to a laboratory. Field study methods are reasonably often encountered in PPIG research, possibly in combination with analysis of design documentation or source code repositories, for example in the work of the Microsoft HIP group. Field research can extend to interview studies (individual 'contextual inquiry' interviews, or structured 'focus group' discussions), 'case studies' of specific projects or organisations, or 'ethnographic' field work in which the researcher becomes immersed in the situation as a participant-observer for extended periods of time.

All of these methods result in the collection of qualitative data, often recorded and transcribed, and often analysed using a grounded theory approach. Chapter 2 of the research methods in HCI book provides an introduction to the use of interviews and focus groups. Detailed considerations of case study research and ethnographic field studies are beyond the scope of this course, as you will not have time to carry out studies of this kind.

Part 3: Special classes of programming language use

Educational Languages

There have always been close connections between the PPIG community, and the field of Computer Science Education, which aims to improve the syllabus, teaching methods, and tools that are used when teaching programming. Early PPIG and ESP meetings included contributions from well known CS educators such as Elliott Soloway. In the UK, a series of researchers at the Open University have reported experimental evaluations of OU course material, and special teaching languages developed for OU students. Sally Fincher at Kent is currently a UK and international leader in CSE.

However, there is also a long-standing tradition in the construction of special programming languages for use by children, not necessarily restricted to a formal educational context. Famous early examples include Papert's Logo and Kay's Smalltalk. Smalltalk rapidly grew beyond the scope of use by children, but Logo has been a longstanding focus of educational research, for example by Richard Noss and Celia Hoyles at the London Institute of Education. More recent languages developed for children have been Alexander Repenning's AgentSheets from Boulder, Allen Cypher and David Smith's Kidsim/Cocoa/StageCast Creator from Apple, Ken Kahn's ToonTalk (now at Oxford), Michael Kolling's Greenfoot (now at Kent, but developing his previous work on BlueJ for older students), the Alice project at Carnegie Mellon and the Scratch project at MIT. Most recently our very own Sonic Pi project, developed by Sam Aaron in collaboration with your lecturer, has been gaining broad attention.

⇒ http://www.agentsheets.com/
⇒ http://www.stagecast.com/
⇒ http://www.toontalk.com/
⇒ http://www.greenfoot.org/
⇒ http://www.alice.org/
⇒ http://scratch.mit.edu/
⇒ http://sonic-pi.net/

Many of these recent projects use visual language techniques, to overcome the problems with syntax that are often experienced by children. There is some debate over the educational consequences, with an argument that since syntax is one of the aspects of programming that seems to be hard to learn, it is either 'cheating' to avoid teaching it, or perhaps deferring problems until later. As a counter-argument, many of these systems are primarily concerned with motivating children to engage with programming, by making it easy for them to build programs that interest them (typically videogames, or animations).

⇒ Chapter 2.5 of the PoP book: 'Programming Languages in Education' describes these different perspectives as 'learning to program' versus 'programming to learn'.

This is a core debate in the design and evaluation of educational programming languages. On one side, a 'user-centred' design philosophy would focus on creating languages that allow children to achieve the things they want to do. On the other, a 'curriculum-centred' philosophy would concentrate on the principles that you want children to learn, and would focus on creating languages that illustrate those principles. The first is more typical of research in the USA, which tends to recruit participating children via after-school clubs or summer camps. The second is more typical of research in the UK and Europe, which tends to introduce experimental systems into classrooms within the context of a lesson. These contrasts are discussed in the following paper:

⇒ Rode, J.A., Stringer, M., Toye, E., Simpson, A.R. and Blackwell, A. (2003) Curriculum focused design. In Proceedings ACM Interaction Design and Children, pp. 119-126.

A further debate that has deep impact on the development of educational languages is the question of what theoretical principles are considered most important for the teaching curriculum. Often the academic advocates of particular programming paradigms are influenced by research trends at the time. At the time the PoP book was written, as can be seen from chapter 2.5, the popularity of AI research had led to advocates of Prolog as a first programming language. The ToonTalk language, although a purely educational language, is also influenced by Prolog-style models of AI research. In Cambridge and Edinburgh, there are strong advocates of ML as a first programming language. The educational goals of the BlueJ system are made quite clear in the title of the associated textbook, 'Objects First with BlueJ'. Debate continues at the time I am writing this, and probably will do for years to come – especially in Cambridge, where the success of the Raspberry Pi as a platform for educational computing has drawn us into the centre of national debates about how that education should be achieved. This was the original motivation for Sonic Pi.

In my own opinion, the design of environments whose goal is 'programming to learn' should be led by educational specialists, who have experience of teaching a range of subjects to children of the appropriate age (in Sonic Pi, we collaborate with Dr Pam Burnard, of the Cambridge Faculty of Education). Unfortunately educationalists can seldom find innovative computer science researchers who are sufficiently able to look beyond their own personal opinions and assumptions to create and evaluate good tools for use by educators. Creating programming languages that some children will enjoy, on the other hand, without necessarily requiring them to be educational, is great fun. But perhaps this has more in common with the next topic of end-user programming.

End-user Programming

The term 'end-user' comes from the Information Systems field, where there is a convention of referring to the business organization that has commissioned a software development project as the 'user'. However the people within that customer organization who become actively involved in the project are often IT professionals themselves (systems analysts, project managers or even programmers). This can cause problems for usability, because the judgments of those IT professionals with regard to what *they* consider usable often doesn't correspond to the experience of the person who eventually gets to operate the system every day. The phrase *end-user* therefore refers to the person who will actually use the program once it is finished. An 'end-user programmer' is thus a person who is not only writing the program, but who will *also* be the person that uses it.

The IS field also refers to 'end-user development' (EUD) and 'end-user customisation' (EUC), to refer to tools and strategies that allow end-users to become more involved in software development, and to have more control over the behaviour of their software. However 'end-user programming' is a particularly provocative term, because it implies a person who is actually doing programming, despite the fact that he or she is not a programmer (or any other kind of IT expert). From an IS perspective, that is almost a contradiction in terms.

There are three things that make EUP a topic of special interest for PPIG researchers. The first is that the years of research into 'novice' programmers gave us a reasonably good understanding of how their knowledge and strategies differ from experts. This means that we already know quite a lot about how to help this group of users. The second is that as computers become more ubiquitous, there are more and more people who would like to use computers for their own purposes. There simply aren't enough programmers to go around, so it is a good thing if ordinary users can look after their own programming needs. Finally, professional programmers are remarkably uncomplaining about the user interfaces that they have to use themselves. In a 'cobbler's children' scenario, programming tools are often the least usable among all software categories. Furthermore, programmers have become so accustomed to the usability shortcomings of their tools, that they even claim to like them that way (which could result either from Stockholm Syndrome, or professional protectionism, depending which side you look at it). Despite this general lack of interest in usability, if we focus on people who clearly have a usability problem, even professional programmers may benefit from the resulting improvements.

Bringing all of these concerns together, EUP is usually defined to refer to a person who has not trained as a programmer, is not primarily employed as a programmer, and does not program for its own sake, but as a means to an end. A regular example is a schoolteacher writing a spreadsheet to calculate grades from a mark-book. However, a successful end-user programmer may find that his or her programs start to be used by other people, in which case they are no longer an 'end-user' in the original sense. That situation, and situations in which people creating business-critical software while not having professional training in software development, has led to the recent research interest in end-user software engineering (EUSE), which is focused on tools to help improve the quality of software created by end-users, for example by assisting them with testing and debugging.

Some of the most successful end-user programming languages have been created for use by people who, although they may be 'novices' in programming, are really experts in their own field, and may be clever enough to acquire basic programming skills very quickly. These end-users can benefit greatly from languages that are designed specifically for use in their own problem domain – a class described naturally as 'domain-specific languages'. Good domain-specific languages, such as National Instruments LabView, can easily become popular well beyond the originally intended audience, for the simple reason that they have been designed with usability in mind. In future, it seems likely that domain-specific languages will become increasingly common. Just take a look at the specialized languages and paradigms in the Windows Presentation Framework – these are sufficiently complex to be a domain-specific language for the user interface development domain.

Creative mashups and composition

Early generations of digital technology were created for military, industrial and bureaucratic applications. These are all domains in which organisations are well structured, and there are ample resources. As a result, it has always been clear who should specify new technology, who should design and build it, who should use it, and who should tell them all what to do. However over the past 20 years, digital technologies have extended into all other areas of life, including leisure, media and the arts. People use these systems because they want to, not because someone is telling them what to do. It seems plausible that the development approaches for these 'discretionary-use' systems ought to be different to the bureaucratic and technocratic design processes of the past. Programming languages are now evolving to suit the new environment and broader applications of digital technology. Languages intended for use in agile development environments are one example, as are AJAX tools, that support applications with increased control and interactivity for website users.

⇒ James Noble and Robert Biddle (2002). Notes on postmodern programming. In Proc. OOPSLA 02, , pages 49-71. http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-02-9.abs.html

⇒ Why's (poignant) guide to Ruby. http://www.rubyinside.com/media/poignant-guide.pdf

If we consider these trends from the perspective of end-user programming, there are much wider audiences of interactive digital media creators who could benefit from the power of programming languages. These are often derived from the 'collaging' nature of digital media, where sampling and mash-ups allow new kinds of artist (possibly without conventional arts training) to make new works. Video mashups are a popular YouTube genre, and sampling in popular music is ubiquitous. Prize-winning open source documentary Rip! A Remix Manifesto illustrates these trends with the work of Girl Talk, a musician who does not use any original sound at all in his performances. All of these trends are informing programming styles that similarly rely on creative open source communities. The Scratch language was named after turntable scratching, with the intention that it should allow children to make creative digital mashups. Interactive web mashups require programming, but can be created with tools such as Yahoo! Pipes.

⇒ Rip! A Remix Manifesto - http://ripremix.com/

⇒ Yahoo! Pipes - http://pipes.yahoo.com/pipes/

At present, there are very few programming languages developed specifically for creative contexts. The most popular programming language for music is Max/MSP, a visual dataflow language (like Yahoo Pipes, though the resemblance probably ends there). The most popular language in visual arts is currently Processing. Max/MSP is regularly used for other time-based media, such as video (with the Jitter plug-in). More advanced musical capabilities are provided by the SuperCollider environment and programming language. Many of these systems are used within a broad context of sampling and mashup artworks, for example Nick Collin's BBcut library for SuperCollider, which uses sophisticated audio algorithms for beat-matching to help users extract breakbeats from music tracks.

⇒ Max/MSP - http://cycling74.com/

⇒ Processing - http://processing.org/

⇒ SuperCollider - http://supercollider.sourceforge.net/

Current research in the Computer Lab is exploring several relevant directions, including the creation of new programming languages for use in dance improvisation (with Wayne McGregor and Random Dance) and for musicians who compose directly to MIDI (Chris Nash), or who carry out live coding – writing sound synthesis software in front of an audience (Sam Aaron's Overtone, a predecessor of Sonic Pi). Your lecturer has previously collaborated with the BBC controller of research to express how end-user programming could be combined with open source and mashup principles to transform public engagement with broadcast media, and has also recently created a novel language for visual artists, that has been used in live performance with Sam Aaron

⇒ Random Dance research - http://www.randomdance.org/r_research

⇒ Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In Proceedings of the Psychology of Programming Interest Group Annual Conference. (PPIG 2012), pp. 163-174. http://www.cl.cam.ac.uk/~afb21/publications/PPIG-2012.pdf

⇒ Chris Nash's reViSiT - http://www.nashnet.co.uk/english/revisit/

⇒ Aaron, S., Blackwell, A.F., Hoadley, R. & Regan, T. (2011). A principled approach to developing new languages for live coding. In Proceedings of NIME'11.

⇒ Blackwell, A.F. and Postgate, M. (2006). Programming culture in the 2nd-generation attention economy. Presentation at CHI Workshop on Entertainment media at home - looking at the social aspects. http://www.cl.cam.ac.uk/~afb21/publications/BlackwellPostgate_CHI06.pdf

⇒ Blackwell, A.F. (2014). Palimpsest: A layered language for exploratory image processing. Journal of Visual Languages and Computing 25(5), pp. 545-571.

This is an expanding application area, and needs serious attention to programming language design. There are many iPad and Android apps that support personal creative media creation, but few of them benefit from a sophisticated understanding of the relationship between programming notations and compositional notations such as music and dance notation. Furthermore, the 'user experience' of programming in these discretionary and creative fields suggests that we need a set of programming tools that is very different to those for traditional bureaucratic systems. An active research topic for our group in Cambridge is how we can create tools that have the power of programming languages, while also supporting the psychological creative experiences of serendipity and 'flow'.

⇒ Church, L., Nash, C. and Blackwell, A.F. (2010). Liveness in notation use: From music to programming. In Proceedings of PPIG 2010, pp. 2-11.

⇒ Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In Proceedings of PPIG 2005, pp. 120-130. http://www.ppig.org/papers/17th-blackwell.pdf

Domestic automation

Digital home technologies are increasingly capable of exchanging data, raising the possibility that they might also exchange control information. However, facilities for programming home appliance behaviour have been notoriously poorly designed. Many people are unable to program the controls of their home heating systems. For many years, programming the VCR was the canonical example of a home task that was unfeasibly hard (HRH Prince Philip has expressed his frustration on this topic to design advocacy venues such as the RSA). Now the configuration of social media systems to optimize privacy or pricing, and monitoring and modification of energy usage patterns seem to be acquiring the same status. Perspectives from PPIG provide ways of addressing these problems that are challenging for conventional HCI.

However, the level of interest in this kind of technical engagement is very low among home-owners themselves. A new 'domestic economy' will be necessary, perhaps drawing on existing models of specialist trades (the 'software plumber'), or extending models of personal competence ('software DIY'). For those who have a significant interest in home automation as a hobby pursuit, technical standards such as X10 already provide the capability to create integrated control systems using power line communications, and have done since the 1970s. Yet these have hardly been popular among the general population. Despite increasingly ubiquitous home networking (e.g. WiFi, Zigbee), it seems unlikely that everybody will want to program integrated home controls. Nevertheless, there is huge room for expansion, whether we consider the relative size of the DIY market, or the need for tools to be used by relatively unskilled professionals.

The 'gentle slope' approach to end-user programming, by which programming languages allow simple things to be done with relatively low effort, but allow scalability to more complex applications with gradually increasing effort, seems to be ideally suited to the domestic automation domain. For this reason, applying the attention investment model to very simple home programming tasks seems to be an important first step. Investigations along these lines have been carried out in the Cambridge group, and a novel tangible programming language 'MediaCubes' was designed as an extension of the standard infrared remote control.

⇒ Blackwell, A.F., Rode, J.A. and Toye, E.F. (2009). How do we program the home? Gender, attention investment, and the psychology of programming at home. International Journal of Human Computer Studies 67, 324-341.

⇒ Rode, J.A., Toye, E.F. and Blackwell, A.F. (2005). The domestic economy: A broader unit of analysis for end user programming. In proceedings CHI'05 (extended abstracts), pp. 1757-1760

⇒ Blackwell, A.F. (2004). End user developers at home. Communications of the ACM 47(9), 65-66.

⇒ Rode, J.A., Toye, E.F. and Blackwell, A.F. (2004). The Fuzzy Felt Ethnography - understanding the programming patterns of domestic appliances. Personal and Ubiquitous Computing 8, 161-176.

⇒ Blackwell, A.F., Hewson, R.L. and Green, T.R.G. (2003) Product design to support user abstractions. In E. Hollnagel (Ed.) Handbook of Cognitive Task Design. Lawrence Erlbaum Associates. ISBN 0-8058-4003-6, pp. 525-545.

⇒ Blackwell, A.F. and Hague, R. (2001). AutoHAN: An Architecture for Programming the Home. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 150-157.

Part 4: Planning practical empirical studies.

The goal of this section is to prepare you for the design of your first experimental study. The lecture itself will follow the actual research interests of the class. Although the first three parts proceeded from theories of programming, to experimental methods, then specific users and programming technologies, our discussion will follow the reverse order, in order to establish connections to the rest of the MPhil curriculum.

Candidate programming languages/tools

We will discuss specific technical platforms and programming paradigms that are of interest to the class. These might be drawn from your own personal research (for example the topic of your MPhil dissertation), from other research that you have encountered while working with Cambridge research groups or in other lecture courses, from recent product releases of new programming systems, or from research prototypes that have been developed elsewhere. If you do not already have a system that you wish to investigate, a number of candidates are available from product announcements on the PPIG and Computing at School mailing lists, or prototypes developed by collaborators of the Rainbow Group.

In order to investigate usability, it is necessary to have an idea of who the intended user is – what is the target audience of the system that you are interested in? What will these users typically be trying to achieve by using the system?

Representative tasks and measures

You will need to identify what kind of user activities you plan to observe, whether these are tasks that you assign explicitly (in a controlled experiment) or that will arise from a user goal (in an observational study). Will these activities allow you to explore an interesting research question or experimental hypothesis that is relevant to your system? What measures are relevant to that question or hypothesis? Will qualitative data analysis be necessary, or is the question sufficiently simple that quantitative measures will suffice? Will this combination of task, measure and analysis result in a threat to external validity?

Review of study design options

Do you wish to carry out a comparison, an evaluation, or an open exploratory study? If you plan to conduct a controlled experiment, will it be possible to use a within-subjects design? What data analysis method will you use? What would you need to do in order to complete a pilot study? What ethical issues are raised by your planned research?

Theoretical goal

What do you expect to learn from conducting your study? What contribution will it make to the research literature relevant to usability of programming languages? Where would you publish the results?

Course structure

The remainder of the course follows the steps leading to a complete research contribution, building on the topics discussed in this lecture.

Assignment A: background to a proposed study, including description of the target language, paradigm, tool or environment, a review of the relevant theoretical literature and previous empirical studies.

Assignment B: structure of the experimental design, detailed protocol of the proposed study, and outline of analytic methods to be used.

Assignment C: full experimental report, building on final versions of assignments A and B, and presenting data analysis and findings in a format suitable for publication at a specialist research venue such as the psychology of programming interest group.

Session 3 and 4: presentation and feedback on study proposals.

Session 7 and 8: presentation and discussion of research study findings.