Software Engineering

Computer Science Tripos 1B
Michaelmas 2011

Richard Clayton

Lecture Five

Tools

- Homo sapiens uses tools when some parameter of a task exceeds our native capacity
  - heavy object: raise with lever
  - tough object: cut with axe
- Software engineering tools are designed to deal with complexity
- There are two types of complexity:
  - **incidental complexity** dominated programming in the early days, e.g. keeping track of stuff in machine-code programs
  - **intrinsic complexity** is the main problem today, e.g. complex system (such as a bank) with a big team. 'Solution': structured development, project management tools, ...
- We can aim to eliminate the incidental complexity, but the intrinsic complexity must be managed

Incidental complexity I

- The greatest single improvement was the invention of high-level languages like FORTRAN
  - 2000 LOC/year goes much farther than assembler
  - code easier to understand & maintain: appropriate abstraction, data structures, functions, objects rather than bits, registers, branches
  - structure lets many errors be found at compile time
  - code may be portable; at least, machine-specific details containable
  - performance gain: 5–10 times! But coding = 1/6 cost, so better languages give diminishing returns
- Most advances since early HLLs focus on helping programmers structure and maintain code
  - don't use 'goto' (Dijkstra 68), structured programming, Pascal (Wirth 71); info hiding plus proper control structures
  - OO: Simula (Nygaard, Dahl, 60s), Smalltalk (Xerox 70s), C++, Java, etc ... all well covered elsewhere

Incidental complexity II

- Don't forget the object of all this is to manage complexity!
- Early batch systems were very tedious for developer
- Time-sharing systems allowed online test – debug – fix – recompile – test – ...
  - this still needed plenty of scaffolding and a carefully thought out debugging plan
- Led on to integrated programming environments such as TSS, Turbo Pascal, Microsoft Visual Studio...
- Some of these started to support tools to deal with managing large projects – 'CASE' (computer aided software engineering)

Formal methods

- Pioneers such as Turing talked of proving programs correct
- Floyd (67), Hoare (71), … now a wide range:
  - Z for specifications
  - HOL for hardware
  - BAN for crypto protocols
- These are not infallible (a kind of multiversion programming) but can find a lot of bugs, especially in small, difficult tasks
- Not much use for big systems
- Much use of rule-based systems that look for suspicious code
  - lint
  - Coverity (see Comm ACM: "A few billion lines of code later")
  - in-house tools at Microsoft, Google, Yahoo! etc.

Programming philosophies I

- `**Chief programmer teams**' (IBM, 70–72)
  - capitalises on the wide productivity variance
  - team of chief programmer, apprentice, toolsmith, librarian, admin assistant etc., to get maximum productivity from your staff
  - can be effective during implementation
  - but each team can only do so much
  - why not just fire the less productive programmers?
- '**Egoless programming**' (Weinberg, 71) – code should be owned by the team, not by any individual. In direct opposition to 'chief programmer team'
  - but: groupthink entrenches bad stuff more deeply

Programming philosophies II

- '**Literate programming**' (Knuth et al)
  - code should be a work of art, aimed not just at machine but also future developers
  - but: creeping elegance is often a symptom of a project slipping out of control
- '**Extreme programming**' (Beck, 99)
  - aimed at small teams working on iterative development with automated tests and short build cycle
  - 'solve your worst problem, repeat'
  - focus on development episode: write the tests first, then the code. 'The tests are the documentation'
  - programmers work in pairs, at one keyboard and screen
  - new-age mantras: "embrace change" "travel light"

Capability maturity model

- Humphrey, 1989 (developed at CMU with DoD money)
  - important to keep teams together, as productivity grows over time
  - nurture the capability for repeatable, manageable performance, not outcomes that depend on individual heroics
- Identifies five levels of increasing maturity in a team or organisation, and a guide for moving up
  1. Initial (chaotic, ad hoc) – starting point for use of a new process
  2. Repeatable – the process is able to be used repeatedly, with roughly repeatable outcomes
  3. Defined – the process is defined/confirmed as a standard business process
  4. Managed – the process is managed according to the metrics described in the Defined stage
  5. Optimized – process management includes deliberate process optimization/improvement

Project management

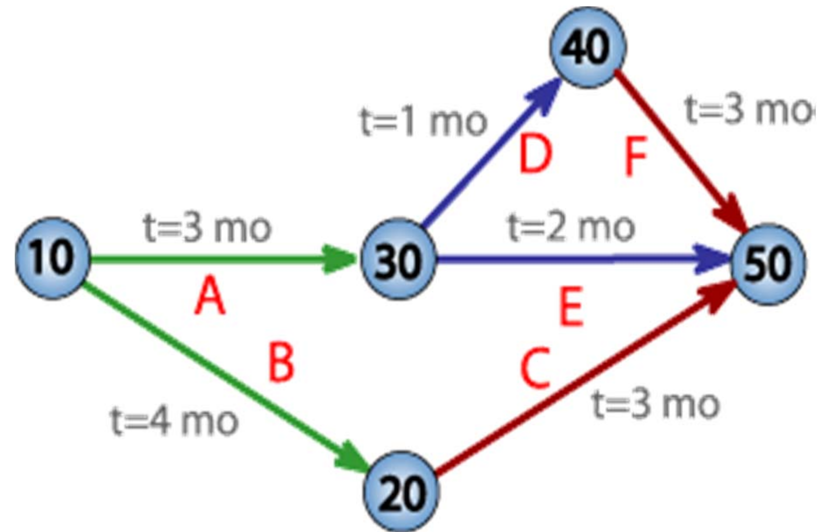
- A manager's job is to
  - plan
  - motivate
  - control
- The skills involved are interpersonal, not techie; but managers must retain respect of techie staff
- Growing software managers is a perpetual problem!
  - 'managing programmers is like herding cats'
- Nonetheless there are some tools that can help

Activity charts

ID	Protocol	Task Name	Start
1	HTTP	Usage survey	01/01/2010
2	HTTP	Error/misconfiguration analyis	01/03/2010
3	HTTP	Develop website for ongoing reporting of survey and analysis results	01/05/2010
4	HTTP	Draw conclusions from measured trends, assessing types of errors seen	01/07/2012
5	HTTP	Prototype tools to assist webmasters in avoiding errors	01/08/2012
6	DNS	Initial DNSSEC prevalence survey	01/07/2010
7	DNS	Identify how DNSSEC can be misconfigured & measure occurrence	01/09/2010
8	DNS	Develop website for ongoing reporting of DNSSEC status surveys	01/10/2010
9	DNS	Draw final conclusions, and evaluate existing error mitigation tools	01/01/2012
10	DNS	Develop prototype tools to prevent hostmasters making errors	01/02/2012
11	BGP	Identify types of BGP errors (and attacks) that can be monitored for	01/11/2010
12	BGP	Develop monitoring (& website) to measure BGP misconfig & attacks	01/12/2010
13	BGP	Develop specification for security oriented BGP solution for IXPs	01/02/2011
14	BGP	Develop initial IXP secured route server prototype	01/03/2011
15	BGP	Test prototype in live network	01/09/2011
16	BGP	Actively avoid usage of bad BGP data and assess impact on IXP stability	01/11/2011
17	BGP	Draw conclusions from measured trends from BGP survey	01/10/2012
18	BGP	Prototype tools to prevent network engineers from making errors	01/11/2012

- ‘Gantt’ chart (after inventor) shows tasks and milestones
- Problem: can be hard to visualize dependencies

Critical path analysis



- Project Evaluation and Review Technique (PERT): draw activity chart as graph with dependencies
- Gives the critical path (here, two) and shows slack
- Can help maintain 'hustle' in a project
- Also helps warn of approaching trouble

Keeping people motivated

- People can work less hard in groups than on their own projects
  - the ‘free rider’ or ‘social loafing’ effect
- Competition doesn’t invariably fix it: people who don’t think they’ll win stop trying
- Dan Rothwell’s ‘three C’s of motivation’:
  - collaboration – everyone has a specific task
  - content – everyone’s task clearly matters
  - choice – everyone has a say in what they do
- Many other factors
  - acknowledgement, attribution, equity, leadership
  - and ‘team building’ (shared food / drink / exercise / other bonding activities)

Agency issues

- Employees often optimise their own utility, not the projects; e.g. managers don't pass on bad news
- People prefer to avoid residual risk issues: risk reduction becomes due diligence
- Tort law reinforces herding behaviour: negligence judged 'by the standards of the industry'
- Cultural pressures in e.g. aviation, banking
- The result is: do the checklists, use the tools that will look good on your CV, hire the big consultants…

Testing I

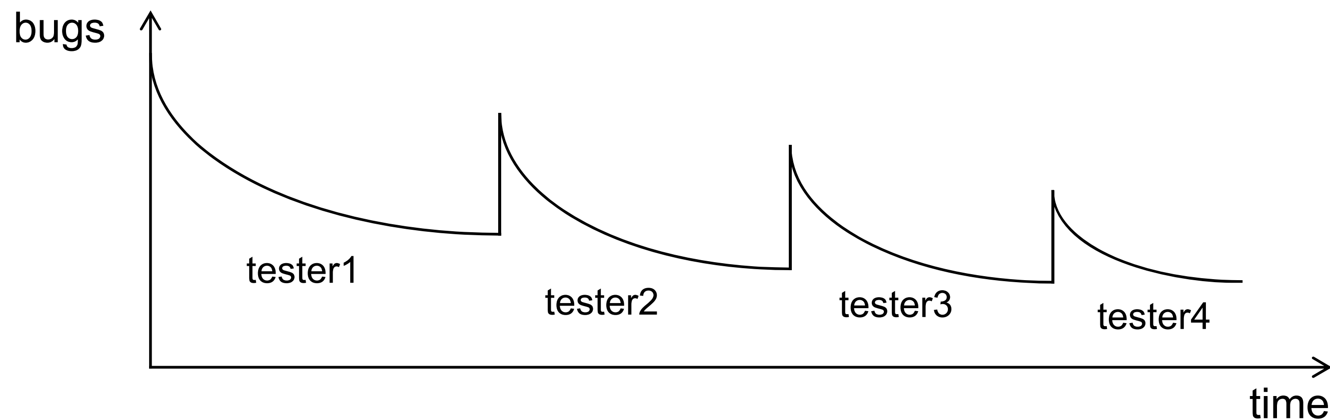
- Testing is often neglected in academia, but is the focus of industrial interest ... it's half the cost
- Important to make bugs reproducible & repeatable
- Bill Gates: "are we in the business of writing software, or test harnesses?"
- Happens at many levels
  - design validation
  - module test after coding
  - system test after integration
  - beta test / field trial
  - subsequent litigation
- Cost per bug rises dramatically as we go down this list!

Testing II

- Main advance in last 15 years is design for testability, plus automated regression tests
  - fuzzing now used to generate test cases
  - fuzzing works well with assertions…
- Regression tests check that new versions of the software give same answers as old version
  - customers more upset by failure of a familiar feature than at a new feature which doesn't work right
  - without regression testing, 20% of bug fixes reintroduce failures in already tested behaviour
  - reliability of software is relative to a set of inputs – best use the inputs that your users generate

Testing III

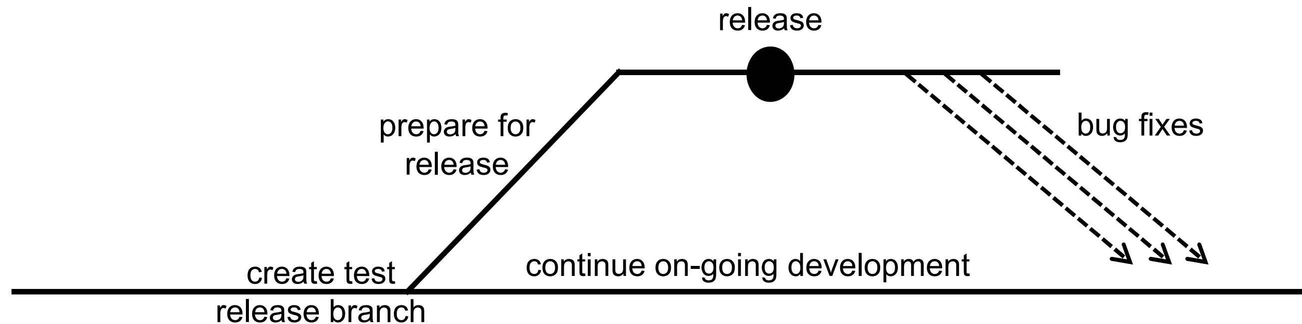
- Reliability growth models help us assess MTBF, number of bugs remaining, economics of further testing…
- Failure rate due to one bug is $e^{-k/T}$; (for T tests, where k expresses how much the bug affects test output)
- With many bugs this sums to k/T
- So for 109 hours MTBF, must test 109 hours
- But: changing testers brings new bugs to light



Testing IV

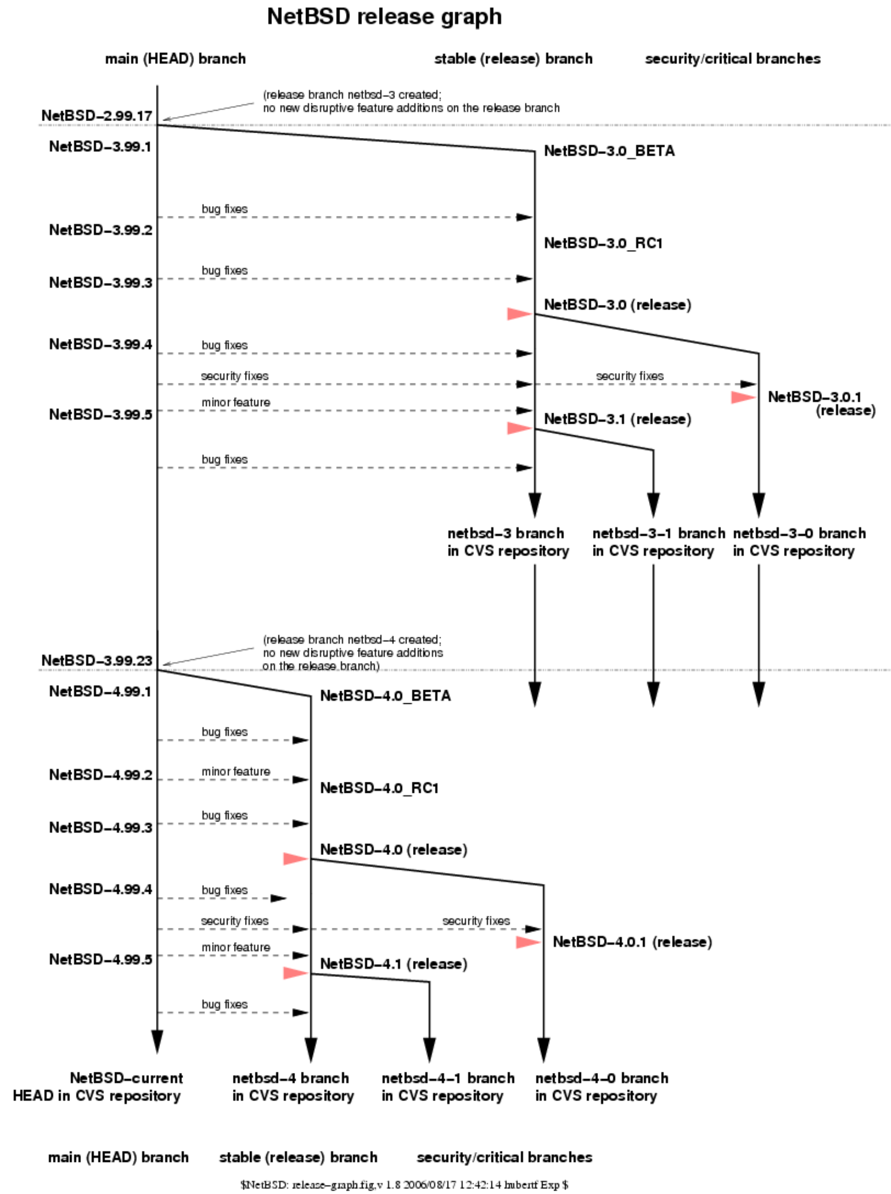
- The critical problem with testing is to exercise the conditions under which the system will actually be used
- Many failures result from unforeseen input / environment conditions (e.g. Patriot)
- Incentives matter hugely: commercial developers often look for friendly certifiers while the military arranges for a hostile review (ditto manned spaceflight, nuclear)
- Just as some people are good at programming, some are good at testing. Typical results from a beta test are that a handful of people will each identify 10 times as many bugs as any other tester – and there will be similar variations in the ability of testers to tell you what they were doing before the failure!

Release management



- Getting from development code to a production release can be nontrivial!
- Main focus is stability – check recently-evolved code, test with lots of hardware versions, etc.
- Add all the extras like copy protection, rights management
- Code freeze impacts work on future developments, so usual practice is to fork the source tree, but then you need to merge bug fixes back into the main branch...

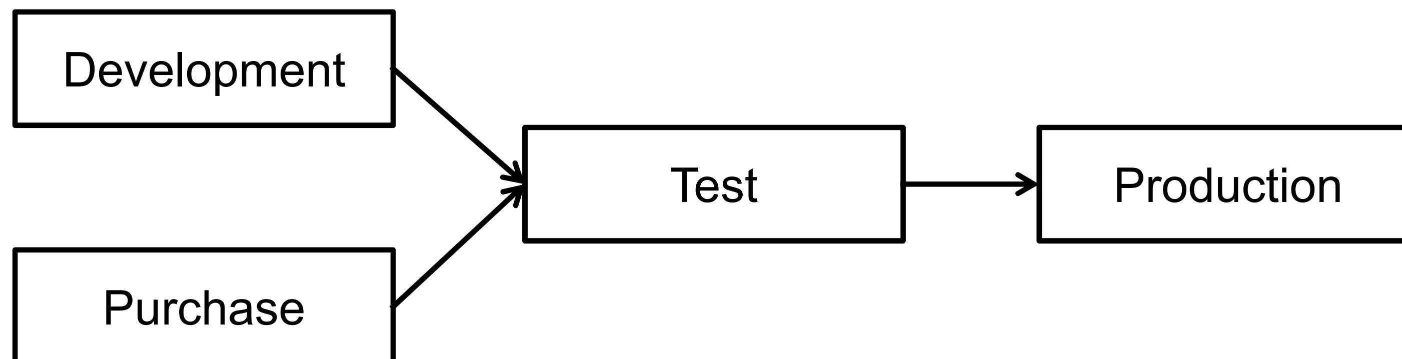
Example – NetBSD release



- Beta testing of release
- Then security fixes
- Then minor features
- Then more bug fixes …

Change control

- Change control and configuration management are critical yet often poor
- Commit messages (in revision control) matter! And they're not written for your benefit, but for others
- The objective is to control the process of testing and deploying software you've written, or bought, or got fixes for
- Someone must assess the risk and take responsibility for live running, and look after backup, recovery, rollback etc.



Documentation

- Think: how will you deal with management documents (budgets, PERT charts, staff schedules)
  - and engineering documents (requirements, hazard analyses, specifications, test plans, code)?
- CS tells us it's hard to keep stuff in synch!
- Possible partial solutions:
  - high tech: CASE tool
  - bureaucratic: plans and controls department
  - social consensus: style, comments, formatting
- Key issue is that code needs to be readable
  - not just executed (in fact that's relatively rare)
  - you have to have layout standards (or they will reformat it)
  - you have to comment! So they [and you] can work out if the bug is doing the wrong thing right, or doing the right thing wrong

```
;
;====================
!ROUTINE: WRITE LOCAL
;====================
;
; WRITE A STRING OF BYTES TO THE FILE FROM A LOCAL VAREA
;
!REQUIRE:   DXAX                         : FILE ADDRESS
!REQUIRE:         CX                     : BYTES TO WRITE
!REQUIRE:   DS.BX                        : BUFFER ADDRESS
!REQUIRE:                SS.BP           : STACK FRAME
!REQUIRE:                      DS        : VAR SEGMENT
!REQUIRE:                            SS  : STACK SEGMENT
;
!PRODUCE:              DI.SI             : -> STREAM
!CORRUPT: F AX BX CX DX             ES   : ALL OTHER REGISTERS PRESERVED
;
        LES,    SI, SF RED TAPE [BP]
;
        MOVW.ES, DI, RT STREAM + 2 [SI]
        MOVW.ES, SI, RT STREAM + 0 [SI]     ;-> STREAM
        CALL,   SEEK TO POS                 ;SET CURRENT POSITION
;
        PUSH    DS
        POP     ES                          ;ES.BX = BUFFER ADDRESS
!CONFIRM: * = WRITE HERE                    ;WRITE THE BYTES
;
;===================
!ROUTINE: WRITE HERE
;===================
;
; WRITE A STRING OF BYTES TO THE FILE AT THE CURRENT POSITION
;
!REQUIRE:   ES.BX                        : BUFFER ADDRESS
!REQUIRE:         CX                     : BYTES TO WRITE
!REQUIRE:              DI.SI             : -> STREAM
!REQUIRE:                      DS        : VAR SEGMENT
!REQUIRE:                            SS  : STACK SEGMENT
:
```

```
//                  ===============
FUNC BOOL           XFile::ReadLine                         PUBLIC
//                  ===============
//
//  read a line from the file
//  returns FALSE if an error
//  at EOF it sets the variable TRUE (and returns an empty line)
//  EOF is latched
//  note that the string has any trailing #0A or #0D stripped
//
(
    XString &       string,             // result string
    BOOL &          at_eof,             // set TRUE iff eof
    const BOOL      support_soft_eof
                        DEFAULT(TRUE)   // TRUE=> #1A is soft eof
)
{
    SingleLock lock(*this);

    ASSERT(isOpen);

    at_eof = seen_eof;                  // at eof already?

    WORD nRead = 0;
    WORD line_size = 128;               // pretty unlikely to need more
    BOOL ret_value = TRUE;

    TCHAR * pline = (TCHAR *)string.GetBufferSetLength(line_size);

    // at_sol TRUE first time round

    for (BOOL at_sol = TRUE; !seen_eof; at_sol = FALSE)
    {
        BYTE chara;
        VERIFY(btpack.Lock());
        ret_value = BU_ReadChar(info, chara);
        VERIFY(btpack.Unlock());
```

Problems of large systems I

- Study of failure of 17 large demanding systems, Curtis Krasner and Iscoe 1988. Causes were:
  1. thin spread of application domain knowledge
  2. fluctuating and conflicting requirements
  3. breakdown of communication, coordination
- Very often linked & typical progression to disaster was 1→2→3

#1: Thin spread of application domain knowledge

- how many people understand everything about running a phone service / bank / hospital ?
- many aspects are jealously guarded secrets
- some fields try hard, e.g. pilot training
- or with luck you might find a real ‘guru’
- but you can expect specification mistakes

Problems of large systems II

#2 The spec may change in midstream anyway

- competing products, new standards, fashion
- changing environment (takeover, election, ...)
- new customers (e.g. overseas) with new needs

#3 Communications problems inevitable

- N people means N(N-1)/2 channels and 2^N subgroups
- traditional way of coping is hierarchy; but if info flows via 'least common manager', bandwidth inadequate
- so you proliferate committees, staff departments
- this causes politicking, blame shifting
- management attempts to gain control, results in restricting many interfaces, e.g. to the customer

Conclusions

- Software engineering is hard, because it is about managing complexity
  - we can remove much incidental complexity using modern tools
  - but the intrinsic complexity remains: you just have to try to manage it by getting early commitment to requirements, partitioning the problem, using project management tools
- Top-down approaches can help where relevant, but really large systems necessarily evolve
- Things are made harder by the fact that complex systems are usually socio-technical
- People come into play as users, and also as members of development and other teams

Conclusions

- About 30% of big commercial projects fail, and about 30% of big government projects succeed. This has been stable for years, despite better tools!
- Better tools let people climb a bit higher up the complexity mountain before they fall off
- But the limiting factors are generally human