Implementation and Testing

OMSE 535 Lecture 1: Introduction
Bart Massey and Dick Hamlet
September 27, 1999

Bart Massey is an experienced implementer and computer scientist.
Dick Hamlet is an expert in testing and implementation.
Today’s Materials

• These notes
• Syllabus
• Academic Honesty document
• This week’s paper
  • *Hints On Programming Language Design*
    C.A.R. Hoare 1973

You will receive the lecture notes weekly, and they will be posted on the course web page.

This packet contains the papers for the first half of the course. The second half of the course will be taught much more from readings.
About the Instructors

- PSU Computer Scientists
  - Bart Massey (bart@cs.pdx.edu) /me/
  - Dick Hamlet (hamlet@cs.pdx.edu)
- Oregon Master of Software Engineering
  (http://www.omse.org)

I am easiest to reach by e-mail; you can also try to contact me at my office at PSU (725–5393).

The OMSE web pages contain much useful information about the program.
The pilot offering of an advanced graduate course is bound to be marked by a need for hard work on everyone’s part. The fact that implementation and testing could easily each be taught separately for 10 weeks will make this problem that much worse.

I expect everyone to arrive every class period prepared to discuss and evaluate the material for the week.
Prerequisites

- Industry SE experience or SE coursework
- Programming experience (C, C++, etc.)
- CS undergraduate degree or equivalent
- Ability to read CS papers (or to learn same)

I assume that you know how to program, know how to design and analyze algorithms, and are comfortable with discrete mathematics. Further, I assume that you are familiar with the problems of, and basic tools for, construction of large (>20KLOC) computer programs.
What You Must Do

• Work
  • Attend class
  • Read texts: McConnell, Hetzel (Myers)
  • Read papers (many)
  • Do project (4 parts)
  • Take quizzes on readings
  • Take final exam

• Academic Honesty

You have a copy of John McHugh’s excellent academic honesty policy document.

I encourage you to work on your own on these assignments. If you do work with other students, it is extremely important to explicitly credit them for contributions. The code, writeups, etc., that you turn in must be entirely your own.
Requirements

- Course textbooks
- E-mail access
- Web access
- In class: pencil & paper
- Be in class on time!

You should be able to get e-mail and web access through your institution, if you don’t have it already. We can talk about exceptions after class, but in 1999 it’s kind of silly to try to be a practicing computerist of any stripe without them.

I suggest using a mail redirector (e.g., iname.com, netscape.com) for your e-mail address for the course. This way if your address changes you will continue to receive course e-mail.
Electronic Classroom

- http://www.omse.org/omse535
- omse535@cs.pdx.edu
- Homework is
  - assigned, graded via e-mail
  - negotiated format
- HW1: Due 1999/10/3
  - send e-mail to omse535@cs.pdx.edu
    - your background
    - course expectations
    - important assignment!

The web page is a focal point of activity for this course. Take a look at it ASAP.

I may drop your name from the course rolls if I have not received HW1 by the due date (and it’s not my fault from mailer screwups. :-}
Textbooks

- *Code Complete*
  Steve McConnell, Microsoft Press 1993
  ISBN 1–55615–484–4

  Bill Hetzel, Wiley/QED 1984
  ISBN 0–89435–242–3

- from www.total-info.com:
  (800) 876–4636

The Hetzel text is old and expensive. Unfortunately, there are really no good, cheap, modern books on testing.

The 800–number ordering will get you the books overnighted; web ordering will take 1.5 weeks.
I am aware of nothing quite like this book out there. It is an invaluable resource on the coding state−of−the−art. You will find me very critical of this text in this course: a) It is impossible for any human being to master this much material, and McConnell is no exception. b) Coding is a “religious” topic. c) The state of the art is not what this course is about (and I mean that in a good way). Within its limitations, however, this is a great book.

By C−style I really mean Algol−style: the examples are drawn mostly from C, Pascal, and BASIC. This book might be somewhat less useful as a cookbook for a Scheme, Haskell, or Prolog programmer.
Neither Dick Hamlet nor myself is terribly fond of this book. We are actually both much fonder of the Myers book, but it has its own problems...

Other portions of this book may also be used in other OMSE courses.
This is in some ways a better text than Hetzel. But it is even older, much more expensive, and lacks a few important topics. If you already have a copy of Myers, you might consider arranging to share texts with someone who has Hetzel rather than purchasing it.
Questions Or Problems?

- If you have questions or problems with
  - course rules or expectations
  - assignments and grading
  - course content
  - etc.
- Contact me! (bart@cs.pdx.edu)

I believe that you will find me very accessible and responsive to e-mail. If you would like to talk on the phone or in person, send me e-mail and we’ll arrange a time. I will keep 3PM−5PM Mondays free on my schedule as office hours, but please make an appointment in advance.

You can also talk to Dr. Hamlet, but he’ll have to speak for himself on this one.
Where This Course Fits

- Software–Intensive Business
- Software Project Management
- Software Engineering (OMSE)
- **Software Craftsmanship (OMSE 535)**
- HW/SW Interface Engineering

The OMSE program actually concentrates on all of the top levels listed here. This course, however, is the main focus of OMSE instruction in what I call “Software Craftsmanship”.

In most areas of engineering, actual implementation is done by skilled (or semi–skilled) craftspeople, rather than by the engineers themselves. It is usually assumed that the engineers have thorough understanding of, and often have experience in, the craft activities required to construct their project.

In SE, things are much more muddled, for a variety of reasons. This course will attempt to remedy this situation somewhat.
Lecture Overview

- About this course
- The “Program Life Cycle”
- Programming languages
- Defensive programming
- About the readings

Together these topics should give a good overview of the first part of this course. We will discuss specific topics from the reading in class each week as time permits.
Program Life Cycle: Products

- Requirements, Design (coding)
- Source Code (compilation)
- Object Code (link)
- Executable (run)
- Execution

Note that this is a very different view from other OMSE courses: It is “code–centric” and probably corresponds more closely to the view of most programmers than better models do.

An execution is a different kind of product than the others: it is dynamic rather than being an artifact. Still, thinking of each execution as an object has some benefits...
Program Life Cycle: Phases

• Coding
• Compilation
  • lexical analysis (tokens)
  • parsing (syntax)
  • datatyping (assignment, checking)
  • code generation
• Linking
  • static linking
  • dynamic linking (DLLs, shared libs)

Of course, there are languages where compilation proceeds differently, as well as interpreted languages (and stranger things). But this model is pretty common.

Datatyping has dual uses: in defect detection and in code generation. We will consider largely the former.
The Execution Phase

• Run time typing
• Late dynamic linking
• Self−checking (assertions)
• Algorithm execution

Note that these things are largely independent: we are concerned with algorithm execution only inasmuch as it is ‘‘correct’’/‘‘incorrect’’. Slightly wrong results are horrible...
Programming Languages

• You can write FORTRAN in any language
• You can write quality code in any language
• Things that help
  • readable syntax
  • precise semantics
  • early error detection
  • ‘‘as simple as possible, but not simpler’’

As you will see from the Hoare paper (we are about to discuss), many of the lessons learned about coding and PL design have been steadfastly ignored by HLL designers for 20 years.
Hoare’s Purposes Of HLLs

- Design
  - should guide design ()
- Documentation
  - program should explain itself
- Debugging
  - should remove accidental drudgery
  - should catch defects early

Obviously programming as well, but most HLLs are good for this. “Debugging” is a placeholder for concepts we will refer to as inspection, testing, defect removal (in construction or maintenance phases). Indeed, some would argue that if a program needs traditional “debugging” of any but the most trivial sort, it has been engineered or crafted improperly!
Hoare’s Principles Of PL Design

- Simplicity
- “Security”
  - precise portable semantics
  - early error detection
- Fast Translation
- Efficient Object Code
- Readability

Simplicity yields transparency, the ability to see the model underlying a PL. What Hoare calls “security” is something we still have no very good term for, although we now usually use Hoare’s term to mean security against deliberate intrusion.

How many of Hoare’s points are valid in 1999?
Does PL Quality Matter?

- Yes and no (big surprise)
- Poor quality PLs place bigger burden on front-end SE activities:
  - Architecture
  - Detailed Design
- In truly bad PLs detailed design should be done to statement level *before* coding!

The practice of avoiding flowcharts and low-level pseudocode is more acceptable in industry each year. I suspect that part of this is due to the existence of better and better languages...
The Fundamental PL Tradeoff

- Simplicity vs. expressiveness
- From an SE POV, simplicity *must* win!
  - where SE errors are introduced vs. cost of correction *(e.g. Boehm)*
  - similar is true for implementation lifecycle:
    catch errors/defects during
    - coding (best)
    - statically (good)
    - startup (ok)
    - at run−time (weak, but better than...)
    - in testing (or after testing!)

One of the many reasons I hated original C++ so much is that it’s very difficult to tell what a statement or routine is going to do when coding it, and very difficult for the compiler to statically catch defects introduced through the misapprehensions. Template−based typing helps somewhat with this, but...
“Modules”

• I will try to use McConnell’s terminology:
  • routine: procedure or function
  • module: collection of routines and data
• Modules are protected by interfaces

Note that some (usually older) authors use the word “‘module’” to refer to routines, which sort of begs the question of what a “‘module’” should be called. “‘Package’” and “‘unit’” (but c.f. “‘unit testing’”) are popular choices here.

Most modern languages provide some support for modules, although C’s in particular is not terribly good. The traditional remedy in languages lacking these features is clever naming conventions.
Modularity and Invariants

- Modules: structured code
- Invariants: structured data
- Together: ADTs allow programs to be
  - tested
    - isolated routines = tractable testing
    - controlled structures = reliable testing
  - maintained
    - Hoare’s “programs as languages”
    - readability of code and data

We will talk about ADTs in great detail next week.
Defensive Programming

- General defensive engineering
  - load tolerance
  - fault tolerance
  - repairability and upgradability
- Defensive software craftsmanship
  - literate programming
  - error handling
- Defensive programming and performance

Anybody with a modicum of technical training can write a computer program that sort of works right under ideal conditions. A well-crafted program is free of internal faults and robust against external disturbances. Some of the techniques for achieving this are common to all engineering disciplines. Some are unique tools of the software craft.

There are currently many competing views of how a well-crafted program should trade robustness and performance. I will discuss some of these in a moment.
Defensive Engineering

• Load Tolerance: software should handle
  • large data sets
  • long runs
  • slow/small platforms
• Fault Tolerance: software must
  • detect as many software faults as possible
  • recover from as many faults as possible

GNU software has a goal of ‘‘no limits’’. Most older UNIX software has ‘‘huge limits’’. Both approaches have advantages: use one. The slow/small platform thing can be carried to extremes. Build for the smallest/slowest platform which will reasonably be encountered. Degree of fault tolerance is a function of risk tolerance.
Repairability and Upgradability

- Repairability and upgradability are linked
- Foundations:
  - modular structure
  - easy access
  - documentation
- Support infrastructure:
  - assertions, tracing support, check code
  - leave in as much as possible

The same techniques that make software repairable tend to make it upgradable. By “easy access”, I mean enough access to internals to do maintenance: e.g., source code when possible and necessary.

We can distinguish between support infrastructure for fault tolerance, and support infrastructure for diagnosis. Fault tolerance code should be largely invisible, and thus left in. Diagnostic code is of necessity visible: it should be left in, but with appropriate UI.
Performance and Support Infrastructure

- "Production versions vs. Debugging versions"
- The worst failure mode
- My rule of thumb: if it can’t stay in, it shouldn’t go in.

There has been much controversy over the question of building separate debugging and production versions of programs. My (extreme) take on this is based on the old observation that the worst failure mode a program can have is to silently produce slightly wrong answers with severe consequences; I take the view that any piece of code which hurts performance enough to be removed from production versions probably also carries enough problems with it that it was a bad idea to write it in the first place.

Performance studies indicate that generation of code to do such things as check array indices in a HLL is fairly cheap (<20% penalty). It is harder to do these studies for programmer-inserted code, but my experience has been similar.
Defensive Software Craftsmanship

• “Literate Programming”
  • Knuth Web style largely failed
    • too hard to write, to read
  • Programs can still be self-documenting

• Error Handling
  • software is incredibly complex
  • must have strategy for recovery from
    • internal faults
    • unexpected external influences

Knuth’s Web (no relation to the WWW) literate programming system was designed to process mixed-mode (code + NL) source into documents plus HLL programs. A better approach is to make the HLL and the HLL program human-readable in the first place.

No other kind of engineered artifact has as much ability to recover from internal and external errors as software. Unfortunately, no other engineered artifact needs them as badly.
McConnell On Metaphors

- I dislike text’s treatment
- Topic deserves serious discussion
  - Why metaphors/analogies?
  - Metalevels
  - Heuristics
  - Idioms

IMHO McConnell gets this material wrong all down the line, but he deserves full credit anyway for actually confronting the issue, and early in the book, too. Tom Duff’s description of ‘‘analogy wars’’ on Usenet has always been dear to my heart: I think that misunderstanding of how metaphors/analogies/similes can be profitably used in science has led to some of the biggest CS/SE problems.

An idiom is a stylized or accepted way of doing/saying something: there are idioms for almost every software activity, at every level. They are extremely important, and much neglected.
Development Metaphors

- Writing Code
- Growing/Accreting A System
- Building A Product
- Software ICs
- **Software Craftsmanship**

The first three metaphors are listed by McConnell: I personally think that he is very hard on writing code”. Software ICs is a metaphor that used to be very popular; it is instructive to note that it has been largely abandoned, partly as IC vendors/integrators have started to run into the same problems SE ran into 15 years ago!
Heuristics

- From Greek *Heuriskein*: to discover
  “...of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance <a heuristic computer program>” —Merriam Webster
  WWWebster Dictionary

- Technique for successful trial-and-error
- Meta- vs. program

Note how different this description is from McConnell’s view, in which heuristics are somehow a *substitute* for algorithms, instead of a way to improve them.
We will discuss naming issues in detail two weeks from now. McConnel’s discussion of functions is a mess (e.g., note the \(=/==\)). The fact of the matter is that in languages without exceptions it is difficult to get around returning boolean status. In general, it is nice if a function returns a value dependent only on the arguments, but this is not always reasonable. He gets it right in the section on naming: a function should return a value which answers a question. McConnell’s discussion of macros is totally out of place here as well, and fairly primitive.
McConnell On Modules

- Coupling and Cohesion
- "Information Hiding"
- PL issues
  - hiding data
  - hiding functions
  - nesting
  - naming

The notion of modules as ADTs is central to modern understanding: we will take it up in detail next week. The Parnas paper meshes very well with this chapter.
Conclusions

• Software should be crafted
• Defensive programming is important
• Modularity is key

Or something. The problem with a high−level survey like this is that it is difficult to summarize any simple conclusions that are not “obvious” and trivial. The really important point is to get a feel for the interaction between all these things.
Next Week’s Papers

- D. Parnas, *On the criteria to be used in decomposing systems into modules*, CACM December, 1972.

In case you haven’t figured it out by now, Parnas is one of the most famous names in SE. He’s known for having a bit of an attitude, as well, and I think the paper on Software Aging shows this up quite nicely. The Modules paper is a classic that you must read. The chapter by Hamlet and Maybee is still a bit rough, but provides the clearest introduction I’ve ever seen to ADTs.