

Software Development Tools

S1

IDEs

Revision Control

Using GitLab and Git

Integrated Development Environments

- A rich context for software development
 - Examples: Eclipse, IntelliJ, VisualStudio, XCode
- Syntax highlighting, continuous compilation, testing, debugging, packaging
- Powerful refactoring capabilities

Revision Control



- Indispensible software engineering tool
- Solitary work
 - Personal audit trail and time machine
 - Establish when bug was introduced
 - Fearlessly explore new ideas (roll back if no good)
- Teamwork
 - Concurrently develop
 - Share work coherently

Git

- Distributed version control system
 - hg, git, others (conceptually very similar)
- Contrast with centralized version control
 - cvs, svn, others

We will focus on distributed version control systems and not discuss centralized version control any further.

Git & GitLab

comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this lab in the lab environment.

Tasks


1. **Set up your GitLab account.**

You will use GitLab throughout the semester to manage all of your coursework.

First you need to set up your GitLab account. Log in to a lab computer, open a browser, and go to GitLab <http://gitlab.cocos.anu.edu.au>. Log in to GitLab using the LDAP tab of the Sign in section of the front page. You should type your student ID and your normal password.

Go to Profile Settings, which is accessible via a gear icon at top right. Feel free to update your GitLab personal profile if you wish.

This completes your GitLab setup.

comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this lab in the lab environment.

Tasks

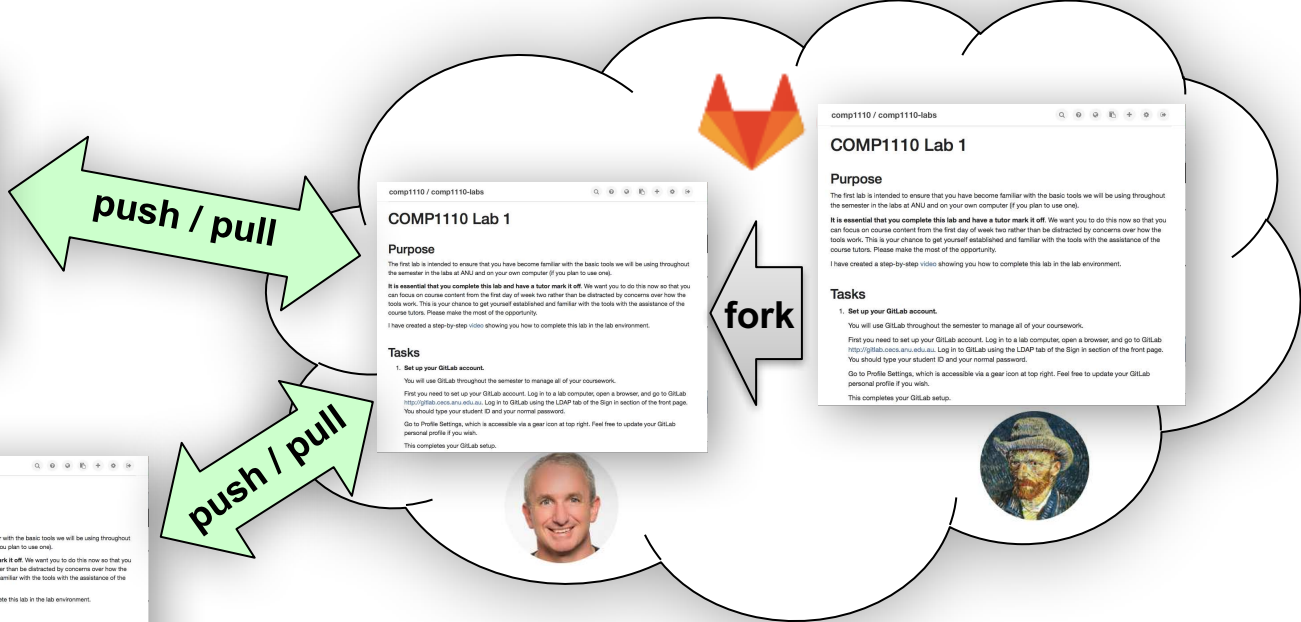
1. **Set up your GitLab account.**

You will use GitLab throughout the semester to manage all of your coursework.

First you need to set up your GitLab account. Log in to a lab computer, open a browser, and go to GitLab <http://gitlab.cocos.anu.edu.au>. Log in to GitLab using the LDAP tab of the Sign in section of the front page. You should type your student ID and your normal password.

Go to Profile Settings, which is accessible via a gear icon at top right. Feel free to update your GitLab personal profile if you wish.

This completes your GitLab setup.



IntelliJ Git Integration

- Create a new repository:
 - VCS->Import into Version Control->Create Git Repository...
- Clone an existing repository:
 - VCS->Checkout from Version Control->Git...
- Other operations:
 - VCS
 - VCS->Git
 - right mouse click -> Git

Revision Control

Git

S2

Git Concepts

- Commit (noun)
- Staging (IntelliJ allows you to more or less ignore this, so we will)
- ✓ Commit (atomically commit changes to your local repo)
- ✓ Push (push outstanding local changes to a remote repo)
- ✓ Pull (pull new changes from a remote repo)
- ✓ Update (update your working version)
- Merge
- Reset and Revert

Git Commits

Captures a set of changes (including modifications, additions and deletions) that may span multiple files.

- Globally unique commit ID (large hexadecimal number)
- Parent – child relationship (based on changeset ID)
 - Single parent, single child is simple case
 - Multiple children indicates a *branch*
 - Multiple parents indicates a *merge*
- A push sends commits, a pull gets commits
- Commits are usually never deleted



A Little More on Update

Update will by default take you to the “HEAD” (the most recent known commit).

You can, however, “update” to any particular revision, moving yourself back and forward in time. To do this, you need to specify the revision.

In IntelliJ you can do this by double-clicking on the revision (VCS -> Git -> Show History, then select the revision)

Branches and Merging

A **branch** occurs when a commit has more than one *child*.

A **merge** is special commit with two *parents* (thus uniting branches).

If branches are *conflicting* (changes to the same file), those conflicts must be **resolved** before merging.

Amend Reset and Revert

You can amend a commit message with **amend**

You can reset your local state to a particular commit (throwing away un-pushed changes whether committed or) with **reset**.

You can also **revert** any particular commit. This amounts to applying a counteracting commit.

When All Else Fails



Monday: S2, S3, J9, O4, O5, B5
Labs: L3
Deliverables: DXC, D2A
Homework: J9, O4
Friday: X1, X2, B6

COMP1110

- Schedule
- Learning Objectives
- Deadlines
- Email
- Code of Conduct**
- Academic Integrity

RELATED SITES

- Piazza

» Code of Conduct

Code of Conduct

You have two primary responsibilities:

- **Promote** an inclusive, collaborative learning environment.
- **Take action** when others do not.

Professionally, we adhere to ACM's Code of Ethics. More broadly, a course like COMP1110 involves reflection, collaboration, and communication. Computer science has a checkered history with respect to inclusion – in corporate environments, in our classrooms, and in the products we create. We strive to promote characteristics of transparency and inclusivity that reflect what we hope our field becomes (and not necessarily what it has been or is now).

Above all, **be kind.**

We reject behaviour that strays into harassment, no matter how mild. Harassment refers to offensive verbal or written comments in reference to gender, sexual orientation, disability, physical appearance, race, or religion; sexual images in public spaces; deliberate intimidation, stalking, following, harassing photography or recording, sustained disruption of class meetings, inappropriate physical contact, and unwelcome sexual attention.

If you feel someone is violating these principles (for example, with a joke that could be interpreted as sexist, racist, or exclusionary), **it is your responsibility to speak up!** If the behaviour persists, send a private message to your course convener to explain the situation. We will preserve your anonymity.

(This code of conduct was developed by Evan Peck of Bucknell University. Portions of this code of conduct are adapted from Dr. Lorena A. Barba)

Test Driven Development

S4

Test-Driven Development (TDD)
JUnit

Test Driven Development (TDD)

TDD “red, green, refactor”

1. Create test that defines new requirements
2. Ensure test **fails**
3. Write code to support new requirement
4. Run tests to ensure code is **correct**
5. Then **refactor** and improve
6. Repeat

Key element of agile programming

JUnit

Unit testing for Java

- Developed by Kent Beck
 - Father of extreme programming movement
- Integrated into IntelliJ
- Useful for:
 - TDD (Test driven development)
 - Bug isolation and regression testing
 - Precisely identify the bug with a unit test
 - Use test to ensure that the bug is not reintroduced

JUnit

- Methods marked with `@Test` will be tested
- When JUnit is called on a class, all tests are run and a report is generated (*a failed test does not stop execution of subsequent tests*).
- JUnit has a rich set of annotations that can be used to configure the testing environment, including:
 - `@Test`, `@Ignore`, `@Before`, `@BeforeClass`, `@After`,
`@AfterClass`
- JUnit can check that an exception is thrown if that is expected in a certain case
 - `@Test(expected = ArithmeticException)`

COMP1510

Software Engineering

\$5

What is Software Engineering?
Does Software Engineering Matter?

Software Engineering

Very roughly:

Software engineering is concerned with the efficient and timely delivery of software that meets stated requirements.

Software Engineering

Software project success rates:

Success 10-30%

Challenged 50-75%

Failed 10-30%

Source: Dan Galorath, *Software Project Failure Costs Billions. Better Estimation & Planning Can Help*, 2008

Software Engineering: Financial Costs

Ariane 5 Failure, ~\$500M, 1996



The error which ultimately led to the destruction of the Ariane 5 launcher about 40 seconds after lift off on its maiden flight was clearly identified in the report of the investigating committee [1]: a program segment for converting a floating point number to a signed 16 bit integer was executed with an input data value outside the range representable by a signed 16 bit integer. This run time error (out of range, overflow), which arose in both the active and the backup computers at about the same time, was detected and both computers shut themselves down. This resulted in the total loss of attitude control. The Ariane 5 turned uncontrollably and aerodynamic forces broke the vehicle apart. This breakup was detected by an on-board monitor which ignited the explosive charges to destroy the vehicle in the air. Ironically, the result of this format conversion was no longer needed after lift off.

Robert L. Baber 2002

Software Engineering: Financial Costs

Queensland Health Payroll, ~\$500M, 2008-2012



IEEE SPECTRUM INSIDE TECHNOLOGY
MAGAZINE MULTIMEDIA

AEROSPACE BIOMEDICAL COMPUTING CONSUMER ELECTRONICS ENERGY

risk factor The views expressed here do not represent those of IEEE Spectrum.

BLOGS // THE RISK FACTOR

A Mismanaged Australian Payroll System Is One of the Worst IT Projects Ever
POSTED BY: ROBERT N. CHARETTE / WED, JUNE 13, 2012

I have blogged about some spectacularly mismanaged government IT development projects, such as the [UK FireControl](#) fiasco, the [US Secure Border Initiative](#) debacle, and [New York City's CityTime](#) scandal. But one that continues to fascinate me is the saga of the [Queensland Health payroll system](#), which will likely play out for at least five more years.

According to Australian news stories like these at the [Delimitter](#) and the [Australian](#), an [audit report](#) (PDF) by the consulting company KPMG into the status of the payroll system indicates that it will cost another A\$220.5 million—on top of the A\$311 million already spent—to fix nine priority items that prevent the payroll of the 85,000 or so Queensland Health employees from being calculated without massive manual intervention. Currently, the audit report states, “1,010 payroll staff are [still] required to perform over 200,000 manual processes on an average of 82,000 forms to deliver

[...] an audit report by the consulting company KPMG into the status of the payroll system indicates that it will cost another A\$220.5 million—on top of the A\$311 million already spent—to fix nine priority items that prevent the payroll of the 85,000 or so Queensland Health employees from being calculated without massive manual intervention.

[...] back in 2008, the original cost of the payroll system development was pegged at A\$6.19 million (fixed price), which has steadily grown as problems such as the massive overpayment or underpayment of employee salaries ran rampant.

Robert Charette, IEEE Spectrum 2012

Software Engineering: Financial Costs

Facebook IPO, ~\$?B, 2012

Bloomberg Our Company | Professional | Anywhere

HOME QUICK NEWS OPINION MARKET DATA PERSONAL FINANCE **TECH** POLITICS SUST

LIVE NOW Bloomberg View's Paul Dwyer and Francis Wilkinson blog convention speeches. Watch live TV coverage. [Tweet](#) 79

Nasdaq Chief Blames Software For Delayed Facebook Debut

By Nina Mehta - May 22, 2012 8:23 AM ET

[f](#) [t](#) [in](#) [+](#) 43 COMMENTS [+](#) QUEUE [+](#)

Nasdaq OMX Group Inc. (NDAQ), under scrutiny after shares of Facebook Inc. were hit by delays and mishandled orders on its first day, blamed "poor design" in the software it uses for driving auctions in initial public offerings.

Computer systems used to establish the opening price were overwhelmed by order cancellations and updates during the "biggest IPO cross in the history of mankind," Nasdaq Chief Executive Officer Robert Greifeld, 54, said yesterday in a conference call with reporters. Nasdaq's systems fell into a "loop" that kept the second-largest U.S. stock venue operator from opening the shares on time following the \$16 billion deal.



The Facebook Inc. logo is displayed with price valuations on monitors during trading at the Nasdaq MarketSite in New York, U.S. Photographer: Scott Elio/Bloomberg



While the errors were resolved and Facebook completed its offering, the day was another setback for equity exchanges trying to erase the memory of the botched IPO in March by Bats Global Markets Inc., another bourse owner. Nasdaq's issues contributed to disappointment among investors as Facebook (FB)'s stock plunged as much as 14 percent today.

"It's amazing that both Bats and Nasdaq unfortunately failed in an inglorious way," William Karsh, the former chief operating officer at Direct Edge Holdings LLC, an exchange operator that competes with Nasdaq, said in a telephone interview yesterday. "It proves that technology isn't infallible. There are so many moving parts that things can go wrong. That's the lesson we learn."

Nasdaq OMX Group Inc [...] blamed "poor design" in the software it uses for driving auctions in initial public offerings. Computer systems used to establish the opening price were overwhelmed by order cancellations and updates [...]. Nasdaq's systems fell into a "loop" that kept the second-largest U.S. stock venue operator from opening the shares on time following the \$16 billion deal.

"It's amazing that both Bats and Nasdaq unfortunately failed in an inglorious way," William Karsh, the former chief operating officer at Direct Edge Holdings LLC, an exchange operator that competes with Nasdaq, said in a telephone interview yesterday. "It proves that technology isn't infallible. There are so many moving parts that things can go wrong. That's the lesson we learn."

Bloomberg 22/5/2012

Software Engineering: Human Costs

Missile Defense Failure, 25/2/91, 28 Dead



[...] the Patriot's failure was at least in part caused by a software flaw. Hitting the incoming Iraqi Scud missile was within the capability of the Patriot system, yet it missed. [...] Yet to characterize this failure as a bug or programming lapse misses a larger point.

This failure can be seen as boneheaded software management. The case can be made that the problem is better traced to a framework flaw. [...] More to the point, a suitable test framework would have detected the flaw, using the same compilers used in the Patriot and without impugning the skills of the Patriot developer team, who may we have excelled in other aspects of that complex software project.

Mark Underwood, Technorati 1/11/2009

Software Engineering: Human Costs

AF447, 1/6/2009, 228 Dead



```
02:12:27 PNF You are climbing,
VS Stall Stall
PNF You are descending, descending descending
02:12:30 PF I am descending?
PNF Descend!
02:12:32 PIC No, you are climbing
02:12:33 PF Here, I am climbing, okay, right so lets descend (or okay we are de
02:12:42 PF OK, we are in TOGA
02:12:42 PF On the altitude where are we?
02:12:44 PIC this is not possible
02:12:45 PF On alti(tude) we are where?
02:12:45 PNF What do you mean on altitude?
PF Yes, yes, yes, I am descending there, no?
PNF Yes, you are descending.
PIC Hey, you are in.... put the wings level,
PNF Put the wings level!
PF That is what I am trying to do
PIC Put the wings level
02:12:59 PF I am at the limit of, with the warping
PIC The rudder
02:13:25 PF What, how is it that we are continuing to descend at the limit there?
02:13:28 PNF Try to find what you can do with the controls up there, The primary
02:13:32 PF At level 100
02:13:36 PF 9000 ft
02:13:38 PIC Carefull with the rudder!
```

Transcripts from BAE, via avherald

Software Engineering: Human Costs

AF447, 1/6/2009, 228 Dead



02:13:38	PIC	Carefull with the rudder!
02:13:39	PNF	Climb, climb. Climb, climb
02:13:40	PF	But I am at the limit of the nose since a while
	PIC	No, no, no, don't climb
	PNF	So descend
02:13:45	PNF	So, give me the controls, to me the controls.
	PF	Go ahead, you have the controls, we are still on TOGA
02:14:05	PIC	Careful, you are nose high (cabres?)
	PNF	I am nose high?
	PF	Well, we need to, we are at 4000 ft
02:14:18	PIC	Go, Pull
	PF	Go, Pull pullpull
02:14:26	PIC	Ten degrees pitch

Transcripts from BAE, via avherald

- *The lack of a clear display in the cockpit of the airspeed inconsistencies identified by the computers;*
- *The crew not taking into account the stall warning, which could have been due to:*
 - *A failure to identify the aural warning, due to low exposure time in training to stall phenomena, stall warnings and buffet,*
 - *The appearance at the beginning of the event of transient warnings that could be considered as spurious,*
 - *The absence of any visual information to confirm the approach-to-stall after the loss of the limit speeds,*
 - *The possible confusion with an overspeed situation in which buffet is also considered as a symptom,*
 - *Flight Director indications that may led the crew to believe that their actions were appropriate, even though they were not,*
 - *The difficulty in recognizing and understanding the implications of a reconfiguration in alternate law with no angle of attack protection.*

Findings from BAE, via avherald

Key Facets of Software Engineering

- Requirements
- Design
- Implementation
- Testing
- Quality
- Maintenance
- Configuration Management

COMP1510

Software Development Models

S6

Big Design Up Front

Waterfall

Spiral

Agile

Formal Methods

The Waterfall Model and “Big Design Up Front”

Benington 1956, Royce 1979, et al

These emphasize getting design absolutely right before progressing the development. Waterfall applies this to all phases: each each must be finalized before moving to the next.

Waterfall stages:

- Requirements
- Design
- Implementation
- Verification
- Maintenance

The Spiral Model

Barry Bohem, 1986

This model is iterative, unlike the waterfall model.

Each iteration includes steps like those in the waterfall model. The spiral model is based on prototyping and iterative refinement.

Agile Development

Beck et al 2001

value-driven

rather than

plan-driven

dynamic

rather than

static

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Beck et al 2001

Formal Methods

Rigorous mathematical approach to verifying correctness of implementation.

Requires

- Formal specification
- Verification of implementation
- Theorem proving assistance (interactive theorem provers).

seL4: Formal Verification of an Operating-System Kernel

DOI:10.1145/1743546.1743574

By Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Phillip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood

Abstract

We report on the formal, machine-checked verification of the seL4 microkernel from an abstract specification down to its C implementation. We assume correctness of compiler, assembly code, hardware, and boot code. seL4 is a third-generation microkernel of L4 provenance, comprising 8700 lines of C and 600 lines of assembler. Its performance is comparable to other high-performance L4 kernels.

We prove that the implementation always strictly follows our high-level abstract specification of kernel behavior. This encompasses traditional design and implementation safety properties such as that the kernel will never crash, and it will never perform an unsafe operation. It also implies much more: we can predict precisely how the kernel will behave in every possible situation.

1. INTRODUCTION

Almost every paper on formal verification starts with the observation that software complexity is increasing, that this leads to errors, and that this is a problem for mission and safety critical software. We agree, as do most.

Here, we report on the full formal verification of a critical system from a high-level model down to very low-level C code. We do not pretend that this solves all of the software complexity or error problems. We do think that our approach will work for similar systems. The main message we wish to convey is that a formally verified commercial-grade, general-purpose microkernel now exists, and that formal verification is possible and feasible on code sizes of about 10,000 lines of C. It is not cheap; we spent significant effort on the verification, but it appears cost-effective and more affordable than other methods that achieve lower degrees of trustworthiness.

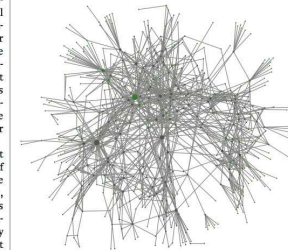
To build a truly trustworthy system, one needs to start at the operating system (OS) and the most critical part of the OS is its kernel. The kernel is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The kernel is a mandatory part of a system's *trusted computing base* (TCB)—the part of the system that can bypass security.¹⁰ Minimizing this TCB is the core concept behind *microkernels*, an idea that goes back 40 years.

A microkernel, as opposed to the more traditional *monolithic* design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping

hardware mechanisms and needing to run in privileged mode. All OS services are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. Previous implementations of microkernels resulted in communication overheads that made them unattractive compared to monolithic kernels. Modern design and implementation techniques have managed to reduce this overhead to very competitive limits.

A microkernel makes the trustworthiness problem more tractable. A well-designed high-performance microkernel, such as the various representatives of the L4 microkernel family, consists of the order of 10,000 lines of code (10 kloc). This radical reduction to a bare minimum comes with a price in complexity. It results in a high degree of interdependency between different parts of the kernel, as indicated in Figure 1. Despite this increased complexity in low-level code, we have demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification.

Figure 1. Call graph of the seL4 microkernel. Vertices represent functions, and edges invocations.



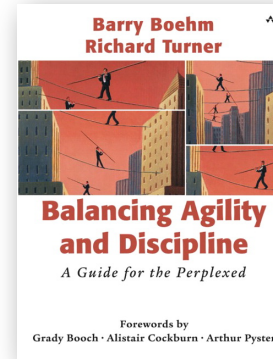
The original version of this paper was published in the *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, Oct. 2009.

JUNE 2010 | VOL. 53 | NO. 6 | COMMUNICATIONS OF THE ACM 107

Which Approach?

Often viewed as a religious question...

Here's Boehm & Turner's take:



Agile home ground	Plan-driven home ground	Formal methods
Low criticality	High criticality	Extreme criticality
Senior developers	Junior developers	Senior developers
Requirements change often	Requirements do not change often	Limited requirements, limited features
Small number of developers	Large number of developers	Requirements that can be modeled
Culture that responds to change	Culture that demands order	Extreme quality

COMP1510

Software Engineering Landmarks

S7

Landmark Publications on Software Engineering

The Mythical Man Month,

Fred Brooks, 1975

Brooks' law: "Adding manpower to a late project makes it later."

Brooks' experience leading the development of IBM's OS/360.

Much of what Brooks describes are what we now call '*anti-patterns*'.

- Adding manpower to a late project makes it later
 - Large complex projects are communications-intensive
 - Adding new people is very costly in terms of communications
 - The communications overhead will eventually dominate
- Second system effect
 - Second implementation is dangerous
 - Tend to want to incorporate all the ideas discarded as impractical
- Scheduling
 - "Q: How does a project get one year late? A: One day at a time!"

No Silver Bullet

Fred Brooks, 1986

“building software will always be hard. There is inherently no silver bullet.”

“Software entities are more complex for their size than perhaps any other human construct.”

“Despite progress in restricting and simplifying software structures, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools”

No Silver Bullet,

Fred Brooks, 1986

Accidental Complexity: artifacts of production of s/w

Essential Complexity: *inherent* in the nature of the s/w

- difficulty in communicating among team: *product flaws etc*
- difficulty in enumerating and understanding all states: *unreliability*
- difficulty of invoking function: *software is hard to use*
- difficulty of extending programs: *unanticipated states, security flaws*

Design Patterns: Elements of Reusable OO Software

Gamma, Helm, Johnson, Vlissides, 1994 (aka “The Gang of Four”)

A long history of using *patterns* in engineering.

The GoF identify 23 *software design patterns*:

- 5 **creational** (create objects)
 - *Prototype* creates an object by cloning an existing one.
- 7 **Structural** (describe object composition)
 - *Proxy* functions as an interface to something else.
- 11 **Behavioral** (describe communication between objects)
 - *Visitor* separates algorithm from structure.

The Five Orders of Ignorance

Phillip Armour, 2000

“the hard part of building systems is not building them, it’s knowing what to build”

“If we view systems development as the acquisition of knowledge, we can also view it as the reduction or elimination of ignorance.”

The Five Orders of Ignorance

Phillip Armour, 2000

0th Order Ignorance: *Lack of Ignorance. I have 0OI when I (probably) know something.*

1st Order Ignorance: *Lack of Knowledge. I have 1OI when I don't know something. With 1OI we have the question in a well-factored form.*

2nd Order Ignorance: *Lack of Awareness. I have 2OI when I don't know that I don't know something.*

3rd Order Ignorance: *Lack of Process. I have 3OI when I don't know a suitably efficient way to find out I don't know that I don't know something.*

4th Order Ignorance: *Meta-ignorance. I have 4OI when I don't know about the Five Orders of Ignorance.*