Software Engineering

Lecture 1
Introduction, principles & architecture

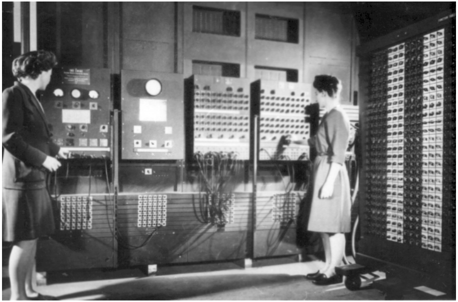
David Baelde
baelde@lsv.fr

MPRI

September 20, 2019

Introduction

Prehistory



Main control panel of ENIAC (1946)

First Turing-complete computers

- Huge and expensive (30 tons, 167m^2, 150kW, 6M$)
- One-off, built for specific purposes (military computations)
- Focus on making hardware reliable

Industrialization

IBM System/360 (1964)

Mainframe computers

- Wide range of applications, scientific to commercial
- Separation of architecture and implementation
- Software complexity rises

Birth of software engineering

1960' software crisis

- Spectacular failures: bugs, cost, overtime, cancellation

Birth of software engineering

1960' software crisis

- Spectacular failures: bugs, cost, overtime, cancellation
- Frederick P. Brooks about OS/360:

  > *The flaws in design and execution pervade especialy the control program. [...] The product was late, it took more memory than planned, the costs were several times the estimate, and it did not perform very well until several releases after the first.*

Birth of software engineering

1960' software crisis

- Spectacular failures: bugs, cost, overtime, cancellation
- Frederick P. Brooks about OS/360:

  The flaws in design and execution pervade especialy the control program. [...] The product was late, it took more memory than planned, the costs were several times the estimate, and it did not perform very well until several releases after the first.

1968 NATO conference on Software Engineering

Need for software manufacturers to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering.

Software Engineering

Problem	Goal
Complexity	Correctness

Software Engineering

Problem	Goal
Complexity	Correctness
Change	Evolutivity

Software Engineering

Problem	Goal
Complexity	Correctness
Change	Evolutivity

Approach : social sciences $\leftrightarrow$ computer science $\leftrightarrow$ hacking

- Principles behing good software products and processes.
- Methodologies that apply and promote those principles.
- Tools to implement and help follow methodologies.

Software Engineering

Problem	Goal
Complexity	Correctness
Change	Evolutivity

Approach : social sciences ↔ computer science ↔ hacking

- Principles behing good software products and processes.
- Methodologies that apply and promote those principles.
- Tools to implement and help follow methodologies.

Scope

Activities	spec.	design	implem.	validation	evolution
Products	doc	doc	code	tests	history

Rigor

Rigor

How to ensure correctness?

- Ideally, formal methods!
- In practice, mostly through rigorous methodologies.

Rigor

How to ensure correctness?

- Ideally, formal methods!
- In practice, mostly through rigorous methodologies.

Correctness is meaningless without a spec!

- Always specify precisely what you need, and no more
- Informal specs (*i.e.*, doc) are *much* better than nothing
- Make sure the spec is visible to the implementer

Rigor

How to ensure correctness?

- Ideally, formal methods!
- In practice, mostly through rigorous methodologies.

Correctness is meaningless without a spec!

- Always specify precisely what you need, and no more
- Informal specs (*i.e.*, doc) are *much* better than nothing
- Make sure the spec is visible to the implementer

"There are two ways to write error-free programs; only the third works." (Alan J. Perlis)

- Be paranoid, seek to detect anomalies early on
- Design precise tests, run them after each change

Change

Anticipation of change

Code *will* evolve

- Bugs will have to be fixed
- Requirements and the environment may change
- Components could be re-used in a (slightly different) context

Anticipation of change

Code *will* evolve

- Bugs will have to be fixed
- Requirements and the environment may change
- Components could be re-used in a (slightly different) context

Anticipation of change

Code *will* evolve

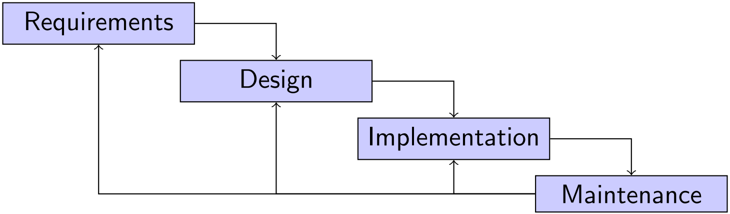
- Bugs will have to be fixed
- Requirements and the environment may change
- Components could be re-used in a (slightly different) context

Brace yourself

- Actively work to identify potential changes
- Design code so that change and re-use is facilitated
- Use tools that help to keep track of change
- Organize work around upcoming changes

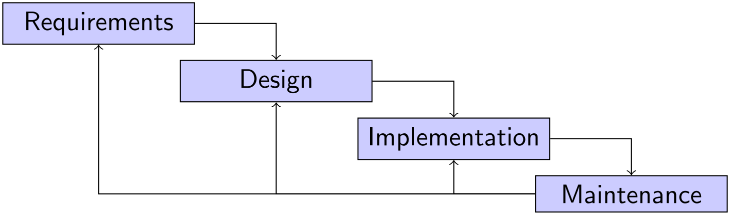
Software development processes

Waterfall model



Software development processes

Waterfall model



- Prevalent at least until 70'
- Probably inspired from other engineering fields
- DoD guidelines for military software: mandatory until 88
  remains reference after that (until recently?)

Incrementality

Proceed step by step to get early feedback and adjust.

- Start by implementing a subset of features.
- Start with functional correctness only.

Incremental development model

Incrementality

Proceed step by step to get early feedback and adjust.

- Start by implementing a subset of features.
- Start with functional correctness only.

Incremental development model

Pros/cons

- ⊕ Early feedback. Opportunity to fix requirements and design. May be necessary if requirements are not initially clear.
- ⊕ Good for the morale of developers and clients!
- ⊖ Requires refactoring to maintain good structure.
- ⊖ Hard to keep track of change in large projects.

The Linux kernel

The main invention in Linux is . . .

The Linux kernel

The main invention in Linux is its development model.

- Wide distribution and invitation to contribute,
  thanks to personal computers and the internet.
- Active integration of patches and frequent releases,
  initially by hand, then with dedicated tools.
- Pre-requisites in the code:
  - precise documentation
  - extensibility through modules for drivers, file system, etc.

E. S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 1999.

More development models

Collaborative software development

Incremental with collaboration and involvement of the public

Main model for open-source software:

- More testers $\rightarrow$ earlier bug reports
- Massive peer review (?)

More development models

Collaborative software development

Incremental with collaboration and involvement of the public

Main model for open-source software:

- More testers $\rightarrow$ earlier bug reports
- Massive peer review (?)

Agile software development

Incremental process + focus on collaboration & self-organization

`http://agilemanifesto.org/principles.html`

Various methodologies (XP, SCRUM...)

Modularity

Modularity

Segment project in modules with clearly defined interfaces.

Modularity

Segment project in modules with clearly defined interfaces.

A slogan: High Cohesion, Low Coupling

- Maximize modularity:
  parallelizability of the software process, chances of re-use
- Minimize interactions:
  separately test, modify... understand, then integrate

Example (types of cohesion)

- Coincidental: no (good) reason
- Temporal: executed around the same time, *e.g.*, init
- Functional: realize a task, *e.g.*, convert file
- ...

Modularity

Segment project in modules with clearly defined interfaces.

A slogan: High Cohesion, Low Coupling

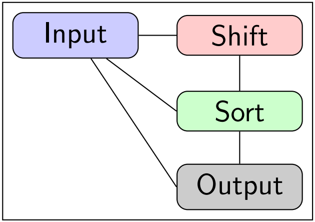
- Maximize modularity:
  parallelizability of the software process, chances of re-use
- Minimize interactions:
  separately test, modify... understand, then integrate

Example (types of cohesion)

- Coincidental: no (good) reason
- Temporal: executed around the same time, *e.g.*, init
- Functional: realize a task, *e.g.*, convert file
- ...

What is a good modularization?

Modularity



Modules export arrays

No text in shifted/sorted arrays

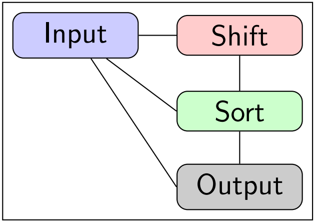
A possible modularization for a *KWIC index generator*:

Input: lines of text

Output:
all permutations of those lines, sorted alphabetically

David Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 1972.

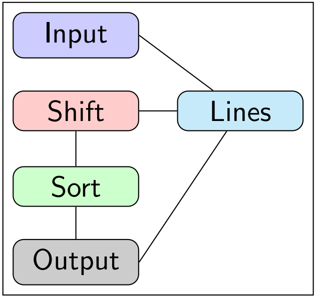
Modularity



Modules export arrays

No text in shifted/sorted arrays

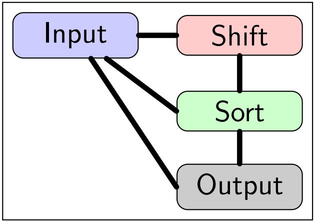
vs.



Modules export `get/set()`

David Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 1972.

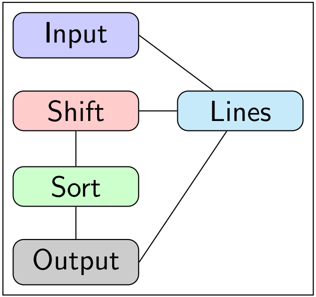
Modularity



Modules export arrays

No text in shifted/sorted arrays

vs.



Modules export `get/set()`

David Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 1972.

Modularity

Segment project in modules with clearly defined interfaces.

A slogan: High Cohesion, Low Coupling

- Maximize modularity:
  parallelizability of the software process, chances of re-use
- Minimize interactions:
  separately test, modify... understand, then integrate

Example (types of cohesion)

- Coincidental: no (good) reason
- Temporal: executed around the same time, *e.g.*, init
- Functional: realize a task, *e.g.*, convert file
- Informational: independent operations on same data, *e.g.*, list

Modularization goes hand in hand with information hiding, aka ...

Abstraction

Ignoring details

Design

- Do not specify implementation *details*.

Ignoring details

Design

- Do not specify implementation *details*.
- Details are things that can easily change:
  max. waiting time, password length, graphics library, etc.

Code

- Code in a high-level language, far from the machine.
- Code for correctness first, then optimize if needed.
  "Premature optimization is the root of all evil." – Knuth
- Don't hardcode:
  no magic numbers, any constant should be justified.

Modularity + Abstraction

Segment project in modules with clearly defined interfaces.

Maximize information hiding in interfaces:

- Minimize coupling.
- Plan for evolution.

Language support

More or less constraining/helpful

- Modules and abstract types in ML-like languages
- Classes in object oriented programming
- Separate compilation units in other languages
- Procedures in *structured programming languages*!

Proof assistants

Concerns of computer-aided theorem proving

- Soundness: the whole point is to have trustworthy proofs!
- Usability: undo, notations, automation, efficiency, user extensions, etc.

Proof assistants

Concerns of computer-aided theorem proving

- Soundness: the whole point is to have trustworthy proofs!
- Usability: undo, notations, automation, efficiency, user extensions, etc.

Edinburgh LCF (70's)

- Proof objects cannot be maintained for performance reasons
- Small trusted kernel provides sound manipulations of abstract datatype `theorem`
- Tactics and tacticals built on top of this sound kernel
- By-product: ML language and module system!

M. J. C. Gordon, *From LCF to HOL: a short history*, 2000.

Proof assistants

Concerns of computer-aided theorem proving

- Soundness: the whole point is to have trustworthy proofs!
- Usability: undo, notations, automation, efficiency, user extensions, etc.

Coq v7 (2000)

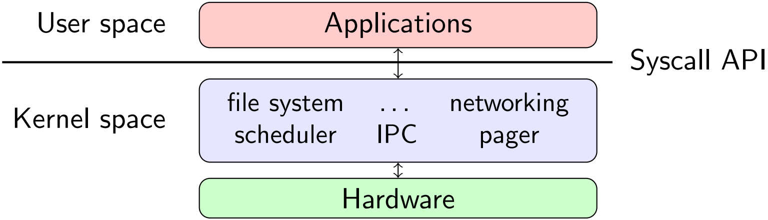
- Proof objects are maintained: relevant, non-local checks
- Isolated kernel: breaking dependency on undo-able objects
- (OCa)ML modules still used: abstraction ensures safety
- Kernel is purely functional, 1/3 of the code
- 2013, `v8.4pl2`: same design, impure kernel, 1/10 of the code

J-C. Filliâtre, *Design of a proof assistant: Coq version 7*, 2000.

Software architecture
examples / success stories

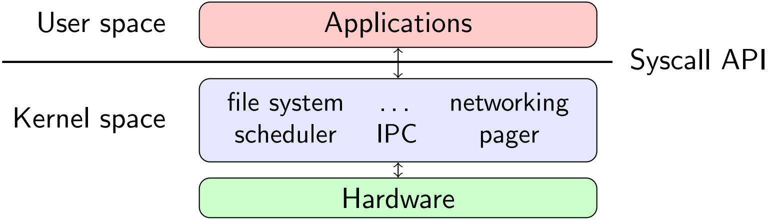
Layers

Monolithic kernel architecture



Layers

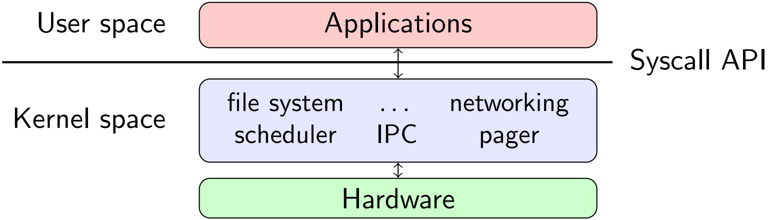
Monolithic kernel architecture



Modern kernels not strictly layered, for performance.

Layers

Monolithic kernel architecture



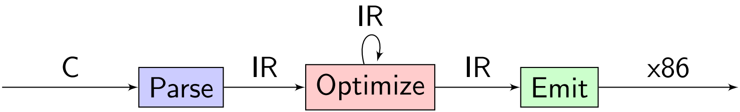
Unix

Powerful abstractions such as processes and file descriptors.

> *The success of Unix lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas.*

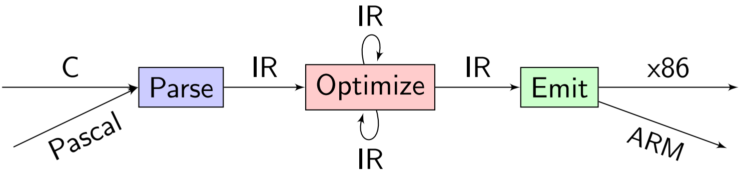
N. Gordon, *Ghosts of the UNIX past: a historical search for design patterns*, LWN, 2010.

Pipes and filters



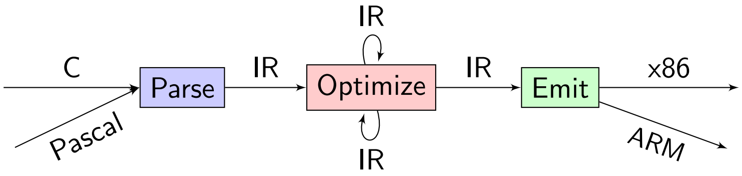
- Parser generators are engineering pearls in themselves

Pipes and filters



- Parser generators are engineering pearls in themselves
- Reason separately about individual "filters" (cf. CompCert)
- Easy extension with new front-ends, back-ends or optimizers?

Pipes and filters



- Parser generators are engineering pearls in themselves
- Reason separately about individual “filters” (cf. CompCert)
- Easy extension with new front-ends, back-ends or optimizers?
- LLVM took this architecture seriously: truly decoupled phases, documented interfaces, ships as library, provides dynamic configuration tools
  $\rightsquigarrow$ maximum re-use, huge community, lots of features

Chris Lattner, *The Architecture of Open Source Applications, volume I*, chapter 11: *LLVM*, 2012.

That's all for now!

Today

We've seen the main principles, aka. the rules of the game:

- Rigor, Adaptability
- Modularity, Abstraction

Next

- Methods:
  - rigorous software development, notably through testing
  - software modelling to guide design
- Tools:
  - git, basic and advanced
  - during project, or on demand: documentation generators, debuggers, profilers...
- Experience through the project

References

- Frederick P. Brooks, *The Mythical Man-Month (20th anniversary edition)*, Addison-Wesley, Prentice Hall, 1995.
- Ian Sommerville, *Software Engineering (9th edition)*, Addison-Wesley, 2011.
- C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.
- A. Hunt, D. Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

...and many others cited in the slides.