Software Engineering

Lecture 1
Introduction, principles & architecture

David Baelde
baelde@lsv.ens-cachan.fr

MPRI

September 14, 2015
Introduction
Prehistory

First Turing-complete computers

- Huge and expensive (30 tons, 167m², 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

Frederick P. Brooks about OS/360:

The flaws in design and execution pervade especially 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.
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 especially 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 especially 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.
A lot of parameters:

- **Product**: is it critical? clearly defined? meant to evolve?
- **Economical aspects**: cost of machines and workers, competition and time pressure.
Software Engineering

A lot of parameters:

- **Product**: is it critical? clearly defined? meant to evolve?
- **Economical aspects**: cost of machines and workers, competition and time pressure.
- **Technology**: languages, RCS, communication means.
- **Science**: PL theory, verification, static analysis, etc.
Software Engineering

A lot of parameters:

- **Product**: is it critical? clearly defined? meant to evolve?
- **Economical aspects**: cost of machines and workers, competition and time pressure.
- **Technology**: languages, RCS, communication means.
- **Science**: PL theory, verification, static analysis, etc.
- **Humans**: client, users, developers, managers, etc.
- **Ideology**: more or less hierarchy, secret, etc.
Software Engineering

A lot of parameters:

- **Product**: is it critical? clearly defined? meant to evolve?
- **Economical aspects**: cost of machines and workers, competition and time pressure.
- **Technology**: languages, RCS, communication means.
- **Science**: PL theory, verification, static analysis, etc.
- **Humans**: client, users, developpers, managers, etc.
- **Ideology**: more or less hierarchy, secret, etc.

Relies on common sense, computer science but also social sciences: psycho, ethno, experiments, etc.
No best approach.
## Software Engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>Correctness</td>
</tr>
<tr>
<td>Change</td>
<td>Evolutivity</td>
</tr>
</tbody>
</table>

**Approach**
- Principles behind 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, code, tests, history
Software Engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>Correctness</td>
</tr>
<tr>
<td>Change</td>
<td>Evolutivity</td>
</tr>
</tbody>
</table>

**Approach**

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

Problems | Goal
---|---
Complexity | Correctness
Change | Evolutivity

Approach

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

Scope

<table>
<thead>
<tr>
<th>Activities</th>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>implem.</td>
<td>code</td>
</tr>
</tbody>
</table>
Software Engineering

<table>
<thead>
<tr>
<th>Problems</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>Correctness</td>
</tr>
<tr>
<td>Change</td>
<td>Evolutivity</td>
</tr>
</tbody>
</table>

Approach

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

Scope

<table>
<thead>
<tr>
<th>Activities</th>
<th>spec.</th>
<th>design</th>
<th>implem.</th>
<th>validation</th>
<th>evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Products</td>
<td>doc</td>
<td>doc</td>
<td>code</td>
<td>tests</td>
<td>history</td>
</tr>
</tbody>
</table>
Rigor
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., docs) are much better than nothing
- Make sure the spec is visible to the implementer

There will be bugs!

- Be paranoid, seek to detect anomalies early on
- Design precise tests, run them after each change
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 *will* be bugs!

- 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

**Be ready**
- 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

- Requirements
- Design
- Implementation

Maintenance

- 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?)
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

```
Specification ➔ Development ➔ Validation
```

Pros/Cons

⊕ Early feedback. Opportunity to fix requirements and design.
   May be necessary if requirements are not initially clear.

⊖ Requires refactoring to maintain good structure.
   Hard to keep track of change in large projects.
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 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.

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 → 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.

The 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
Modularity

- Segment project in modules with clearly defined interfaces.

The 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

What is a good modularization?
A possible modularization for a *KWIC* index generator:

**Input:** lines of text

**Output:** all permutations of those lines, sorted alphabetically

---

Modularity

Modules export arrays
No text in shifted/sorted arrays

vs.

Modules export get/set()

Modularity

Modules export arrays
No text in shifted/sorted arrays

vs.

Modules export get/set()

Abstraction
Ignoring details

Design

- Do not specify implementation details.
- Details are things that can easily change: maximum waiting time, password length, 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!

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, v8.4pp12: same design, impure kernel, 1/10 of the code

Exercises
Discuss the following Java function:

```java
public void showRemaining(User u, Conference c) {
    TimeZone tz = u.getLocation().getTimeZone();
    Date d = c.getPaperDeadline();
    ... // something involving only tz and d
}
```
Pairs

(Based on 2013 MPRI project “Geriatric Terrorist Anarchy”)

Two C++ classes use pairs:

- The UI performs drawing using SFML’s `Vector2f` class.
- The simulator moves characters around the world, also using 2D floating point coordinates.

Alternatives to discuss:

- Use `Vector2f` for the simulator code.
- Create a new class for pairs of floats.
- (Use `std::pair`.)
char *strtok(char *str, const char *delim);

The `strtok()` function parses a string into a sequence of tokens. On the first call to `strtok()` the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` should be NULL.

Discuss
- What’s wrong with this spec?
- Give examples of when the function is unusable.
- Propose other designs, not necessarily in C.
Software architecture examples
Layers

Monolithic kernel architecture

User space

Kernel space

Applications

file system
scheduler
IPC
networking
pager

Hardware

Syscall API
Layers

Monolithic kernel architecture

User space

Applications

Kernel space

file system
scheduler
IPC
networking
pager

Hardware

Syscall API

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.*

Layers

Monolithic kernel architecture

User space

Applications

Kernel space

file system
scheduler
IPC
networking
pager

Hardware

Syscall API

Exercise

Does Unix follow a strict layered architecture?
Reflect the abstraction level of memory or file descriptors.
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
  \( \Rightarrow \) maximum re-use, huge community, lots of features

Plugins

Plugins are **dynamically loaded** software components:

- Core software defines **interfaces**
  for acquiring data, converting file formats, building UIs, etc.
- Core software loads plugins dynamically
- Plugins register new **implementations** of interfaces
- User may explicitly trigger new feature
- In case of implicit use, a selection mechanism is needed

Remarks

- Full exploitation of modularity and abstraction
- Eases extension and configuration of software
- Enables external contributions
- Simple “static plugins” already useful for configurable builds
Plugins

Plugins are dynamically loaded software components:

- Core software defines interfaces for acquiring data, converting file formats, building UIs, etc.
- Core software loads plugins dynamically
- Plugins register new implementations of interfaces
- User may explicitly trigger new feature
- In case of implicit use, a selection mechanism is needed

Remarks

- Full exploitation of modularity and abstraction
- Eases extension and configuration of software
- Enables external contributions
- Simple “static plugins” already useful for configurable builds
Events

When things are so decoupled, interactions seem quite limited...
Events

When things are so decoupled, interactions seem quite limited... 

“Please call me whenever event E occurs.”

- Sender and receiver don’t need to know each other
- Central event manager or peer-to-peer system
- Set of events may or may not be fixed
Events

When things are so decoupled, interactions seem quite limited...

“Please call me whenever event E occurs.”
- Sender and receiver don’t need to know each other
- Central event manager or peer-to-peer system
- Set of events may or may not be fixed

Limitations
- Some form of “runtime coupling”
- Isolated tests of limited use
- Abuse may lead to messy control flow
Scripting

Video game (Battle for Wesnoth)

- **Separate** game logic (campaigns, characters...) from engine, graphics, etc.
- **Simpler script language** for high-level stuff
  ➞ more contributors

---

Scripting

Video game (Battle for Wesnoth)

- Separate game logic (campaigns, characters . . . ) from engine, graphics, etc.
- Simpler script language for high-level stuff → more contributors

Configuring characters

How to handle combinations such as elvish warriors, drunken ogres, invisible men, and so on?

Scripting

Video game (Battle for Wesnoth)

- Separate game logic (campaigns, characters...) from engine, graphics, etc.
- Simpler script language for high-level stuff → more contributors

Configuring characters

How to handle combinations such as elvish warriors, drunken ogres, invisible men, and so on?

- Theorician’s approach: new language to describe these traits
- WML approach: \textit{keep it simple}, fixed set of possible features

References


...and many others cited in the slides.