Software maintenance and Evolution

Prof. Romain Robbes

Outline

About this class

A brief history of Software Engineering

Essential and accidental difficulties in Software Engineering

Maintenance & evolution: facts and figures

The software change process

Change initiation

About this class

Course Format

Frontal lectures

- ~~Monday 14:00 – 16:00~~
- ~~Wednesday 14:00 – 16:00~~

Labs

- ~~Monday 16:00 – 18:00~~

Office hours

Course Format

Frontal lectures

~~Wednesday 14:00 – 16:00~~ Thursday, 12:00–14:00

Labs

~~Monday 14:00 – 18:00~~ Tuesday, 08:00–12:00
(Today we end at 10:00)

Office hours

By email appointment (rrobbes@unibz.it)

Administrative information about the course

Contact Details

rrobbes@unibz.it

Piazza Domenicani 3 - office 1.16

(First floor to the right)

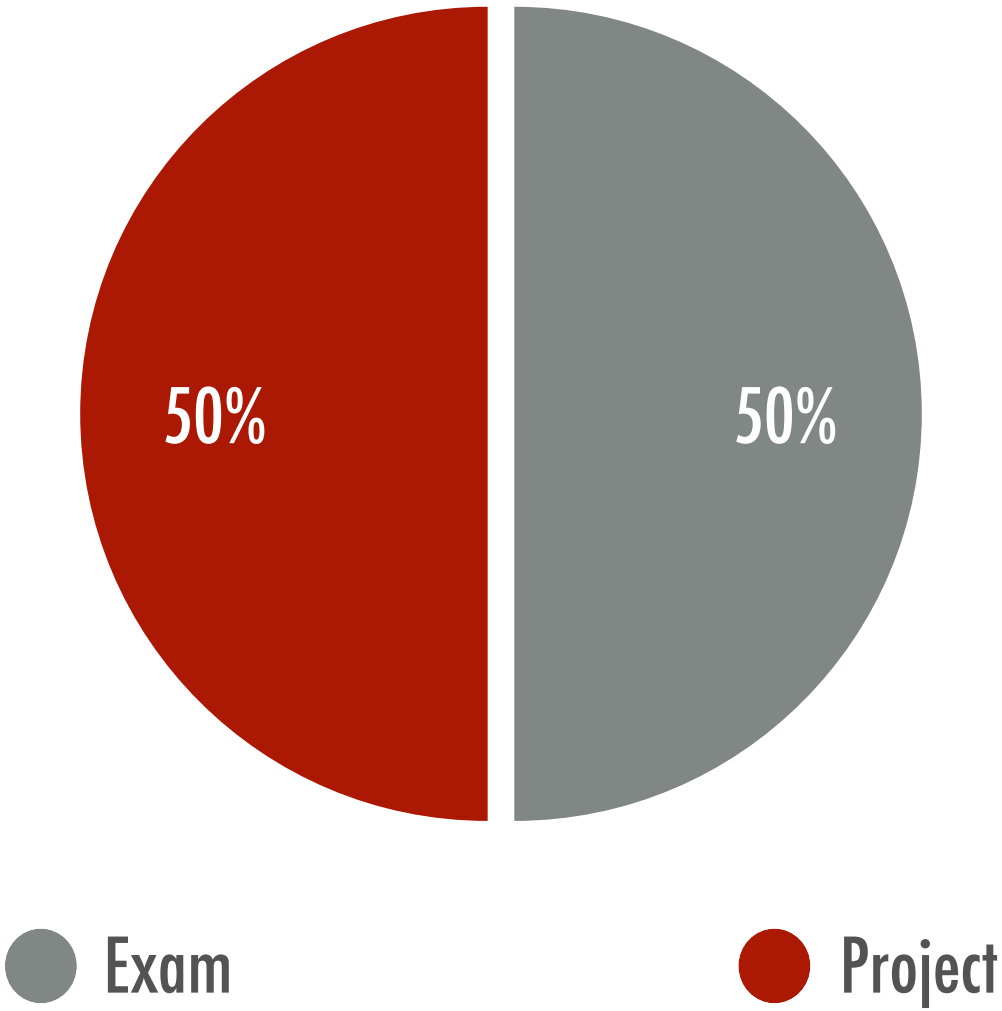
Administrative information about the course

Teaching Material

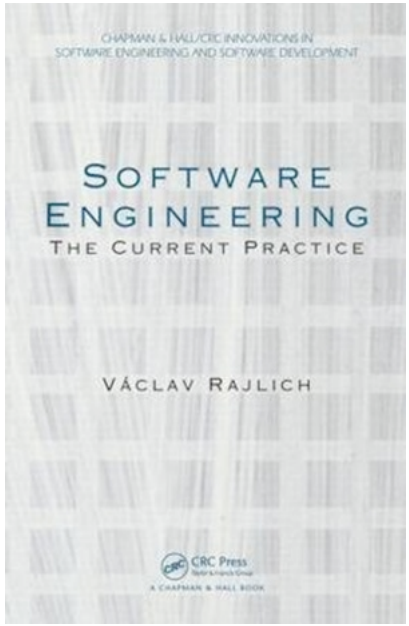
https://ole.unibz.it/course/view.php?id=4112

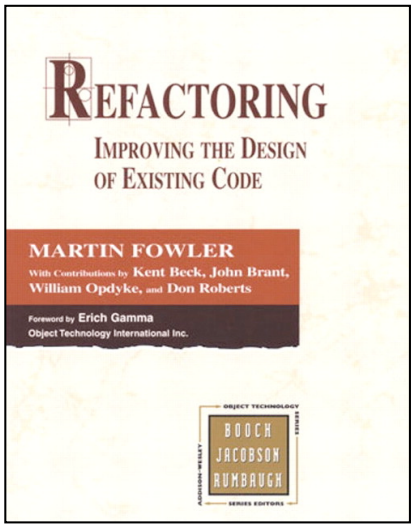
Other references will be added to the course web site

Evaluation



Material for this course





Software Engineering: The Current Practice – V. Rajlich
Refactoring: Improving the Design of Existing Code – M. Fowler
Code Reading: The Open-Source Perspective – D. Spinellis
Working Effectively with Legacy Code – M. Feathers
Your Code as a Crime Scene – A. Thornhill

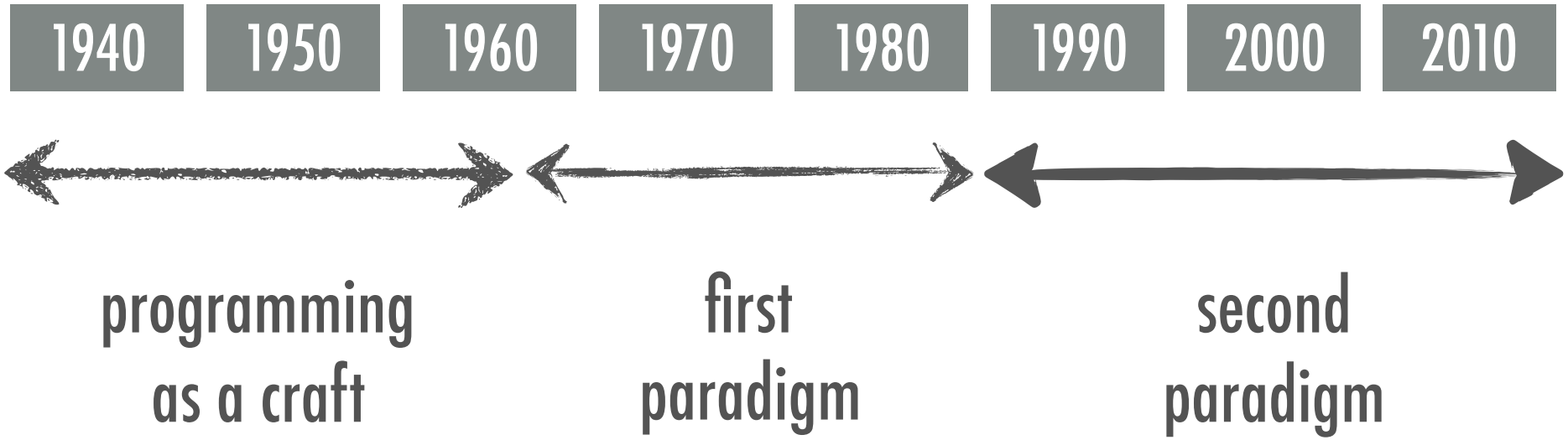
Check library for online access

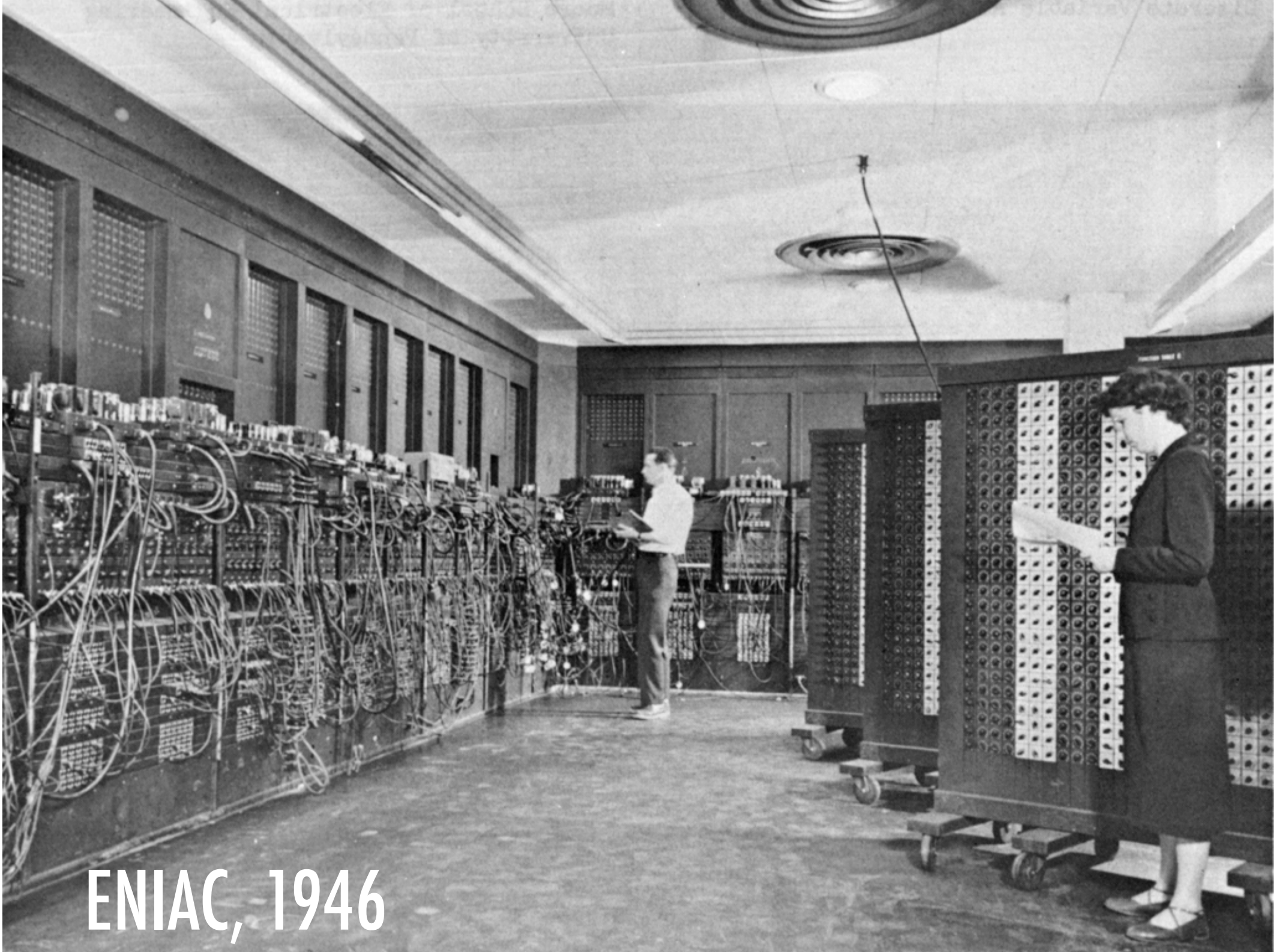
Topics of this class: How to evolve software systems

- Basic techniques
  - Concept location
  - Impact analysis
  - Change propagation
  - Refactoring
  - Verification
- Advanced techniques
  - Mining Software Repositories
  - Software Metrics
  - Text Analysis
  - Defect Prediction
  - Search-based Techniques

A brief history of software engineering (Or, why we need all this)

Software engineering timeline





ENIAC, 1946

When a bug was a bug ...

Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 Arctan started

1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 ?95 correct

13"uc (032) MP - MC ~~1.5821447000~~ ~~2.13047641 5 (-3)~~ 4.615925059(-2)

(033) PRO 2 2.13047615

correct 2.13067615

Relays 6-2 in 033 failed special speed test

In relay " 11.00 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545 

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~1700~~ 1630 Arctangent started.

1700 closed down.

Relay 2145
Relay 3370

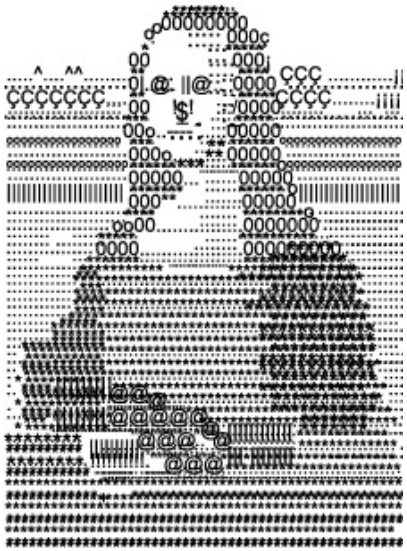
When programs were “simple”

Programs: differential equations, ballistics, ...

No “programmers”: physicists, mathematicians

Hardware cost dominates: ENIAC cost 6,000,000 $ (U.S.)

Programming is an art



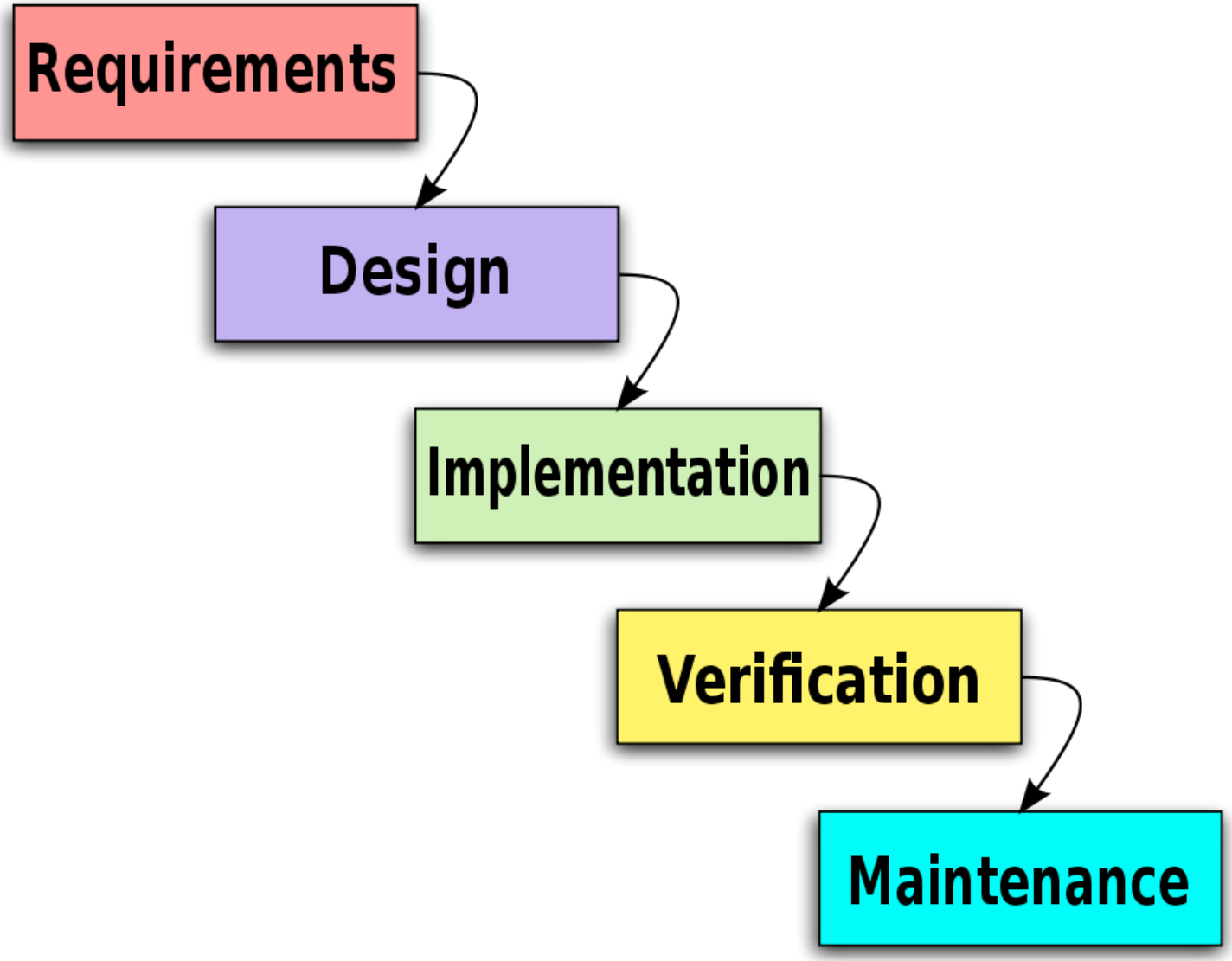
Programming is an art ... until the 60's

IBM's OS/360 proves that this approach has limits

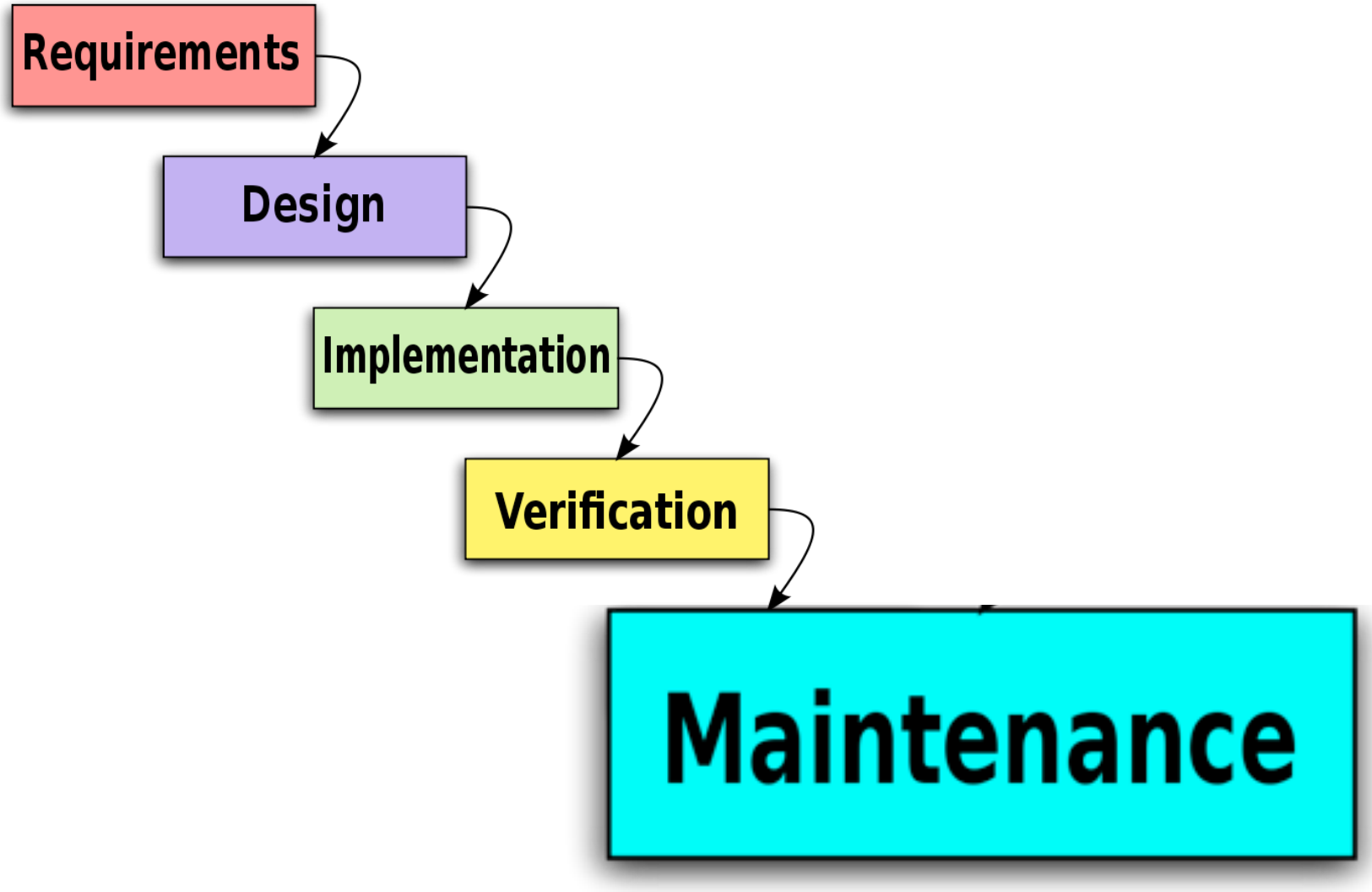
Discovery: adding people to a late project slows it down!

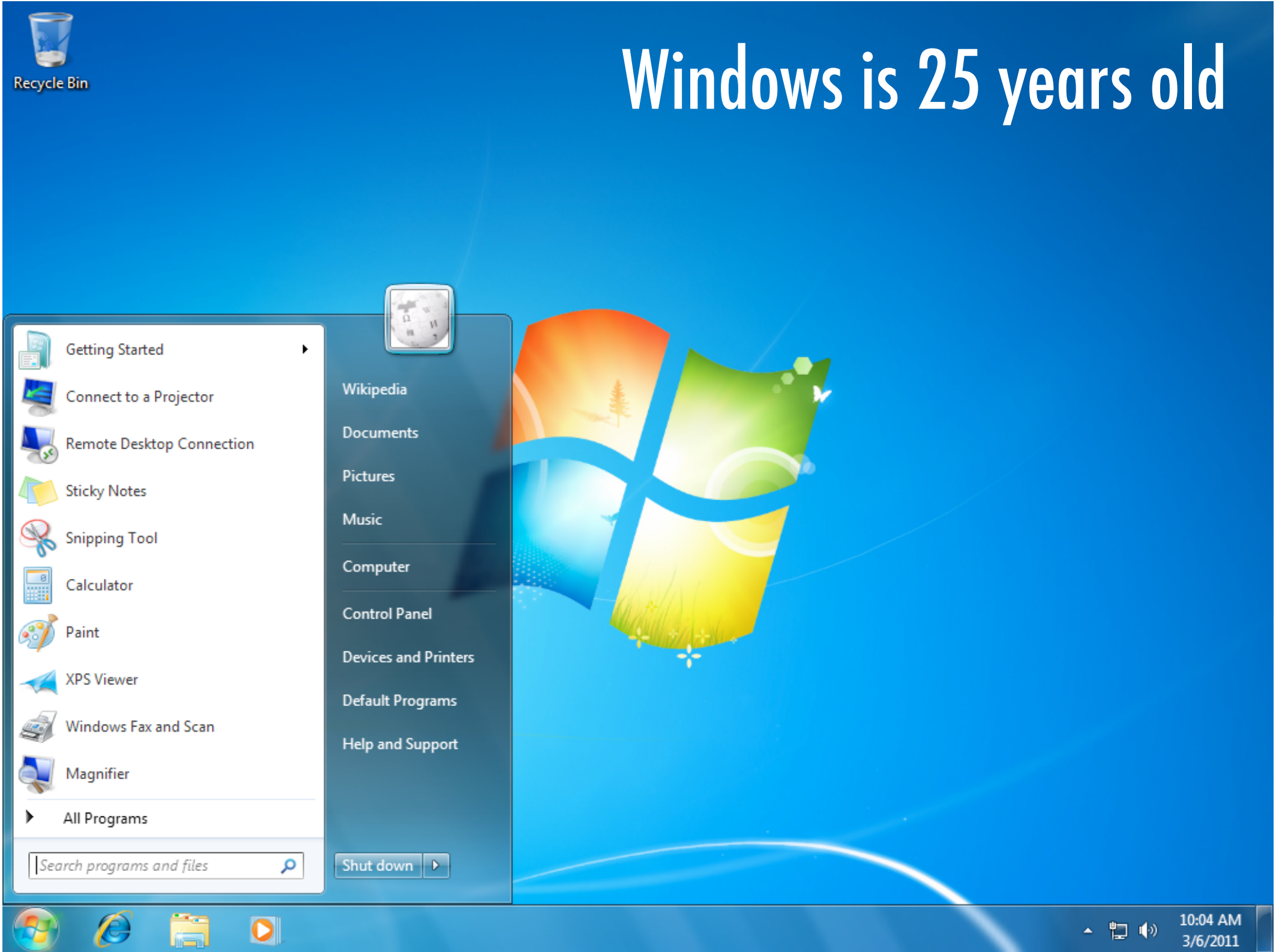
First software engineering conferences, 1968 and 1969

The first standards activities and processes emerge, e.g. waterfall



During the 80's and 90's, people realize that software systems live very long!



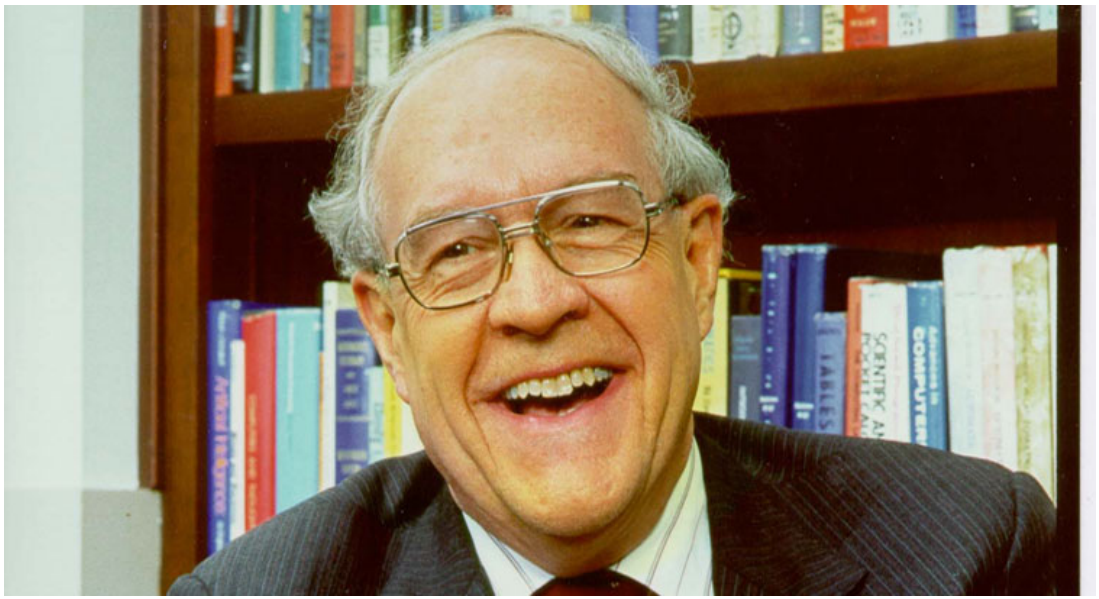

Windows is 25 years old
Recycle Bin
Getting Started
Connect to a Projector
Remote Desktop Connection
Sticky Notes
Snipping Tool
Calculator
Paint
XPS Viewer
Windows Fax and Scan
Magnifier
All Programs
Search programs and files
Wikipedia
Documents
Pictures
Music
Computer
Control Panel
Devices and Printers
Default Programs
Help and Support
Shut down
10:04 AM
3/6/2011

Scratch that, make it 30

The difficulties of software development

What do you think?

Essential vs. accidental difficulties



Essential difficulties:
properties specific to software that are always present.

Accidental difficulties:
properties that may vary from one system to another.

The 5 essential difficulties of software

Complexity

Invisibility

Changeability

Conformity

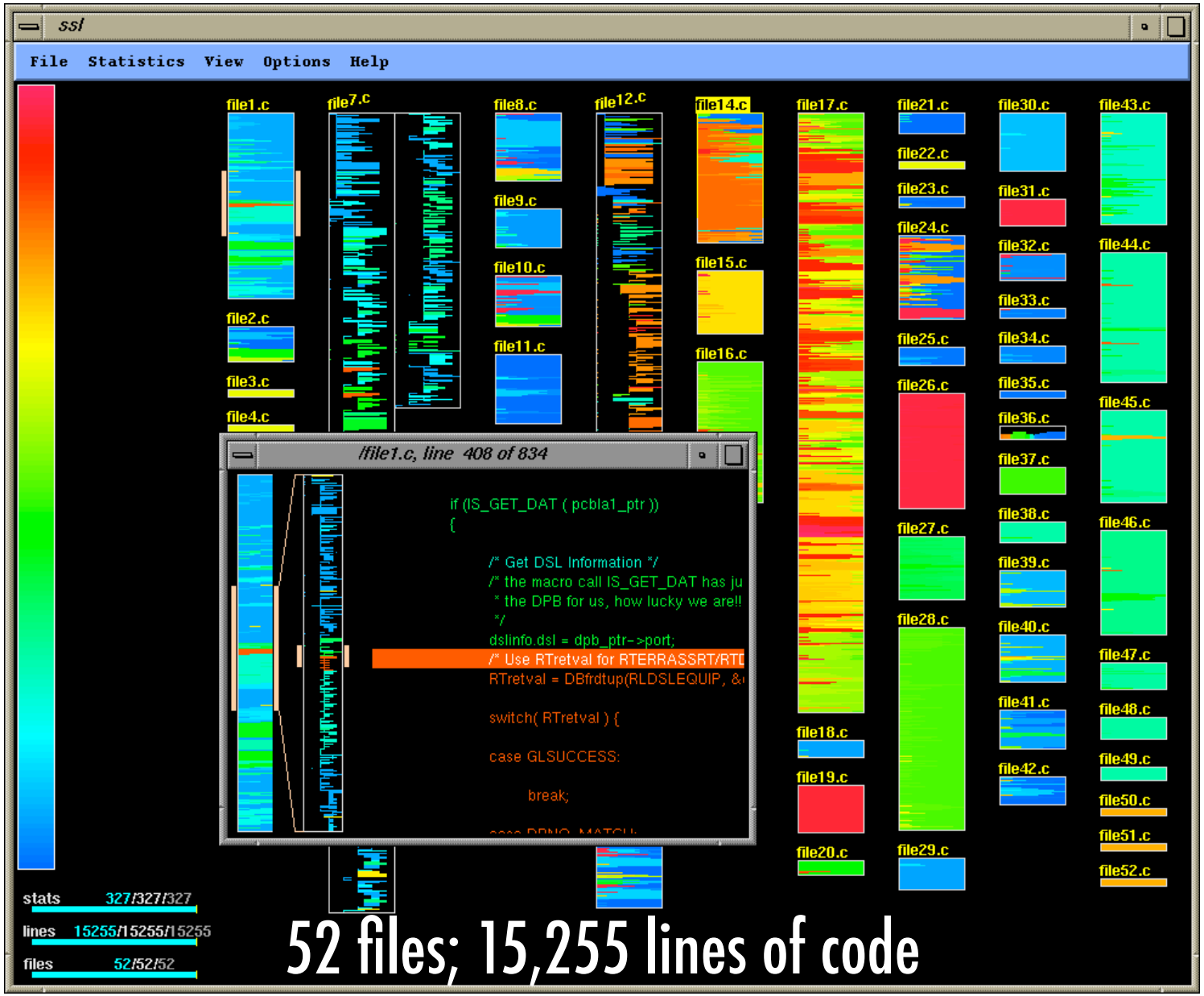
Discontinuity

Software is complex

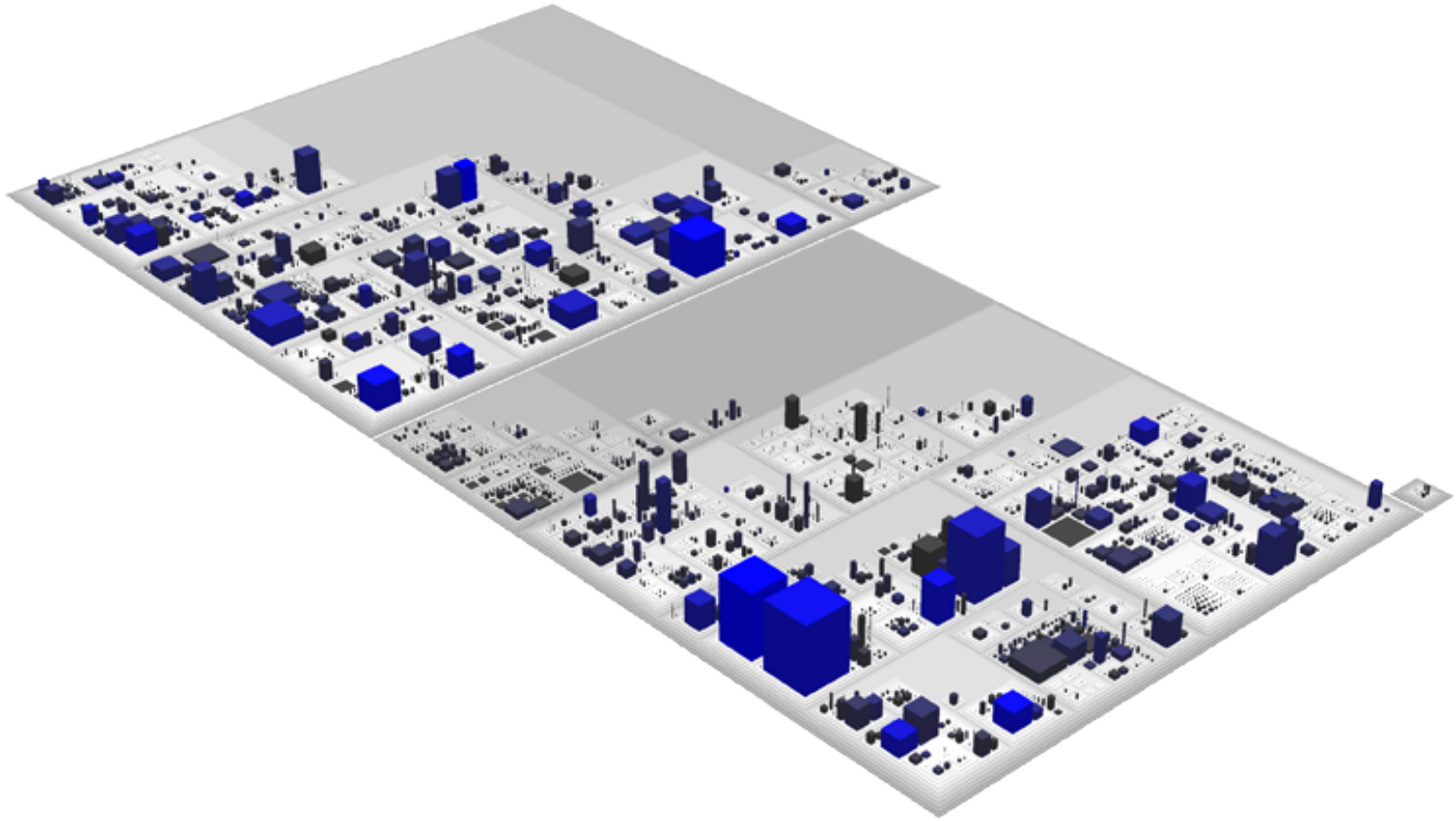
```
#include <stdio.h>
#include <stdlib.h>

main()
{
int i;
for (i=0; i<10; i=i+1);
  printf("i is %d\n",i);
}
```

main.c: 1 file,9 lines of code

Now consider the following code ...



Now consider the following code ...



Azureus: 4,656 classes; 454,387 lines of code

Now consider the following code ...

Windows Vista: 50,000,000+ lines of code

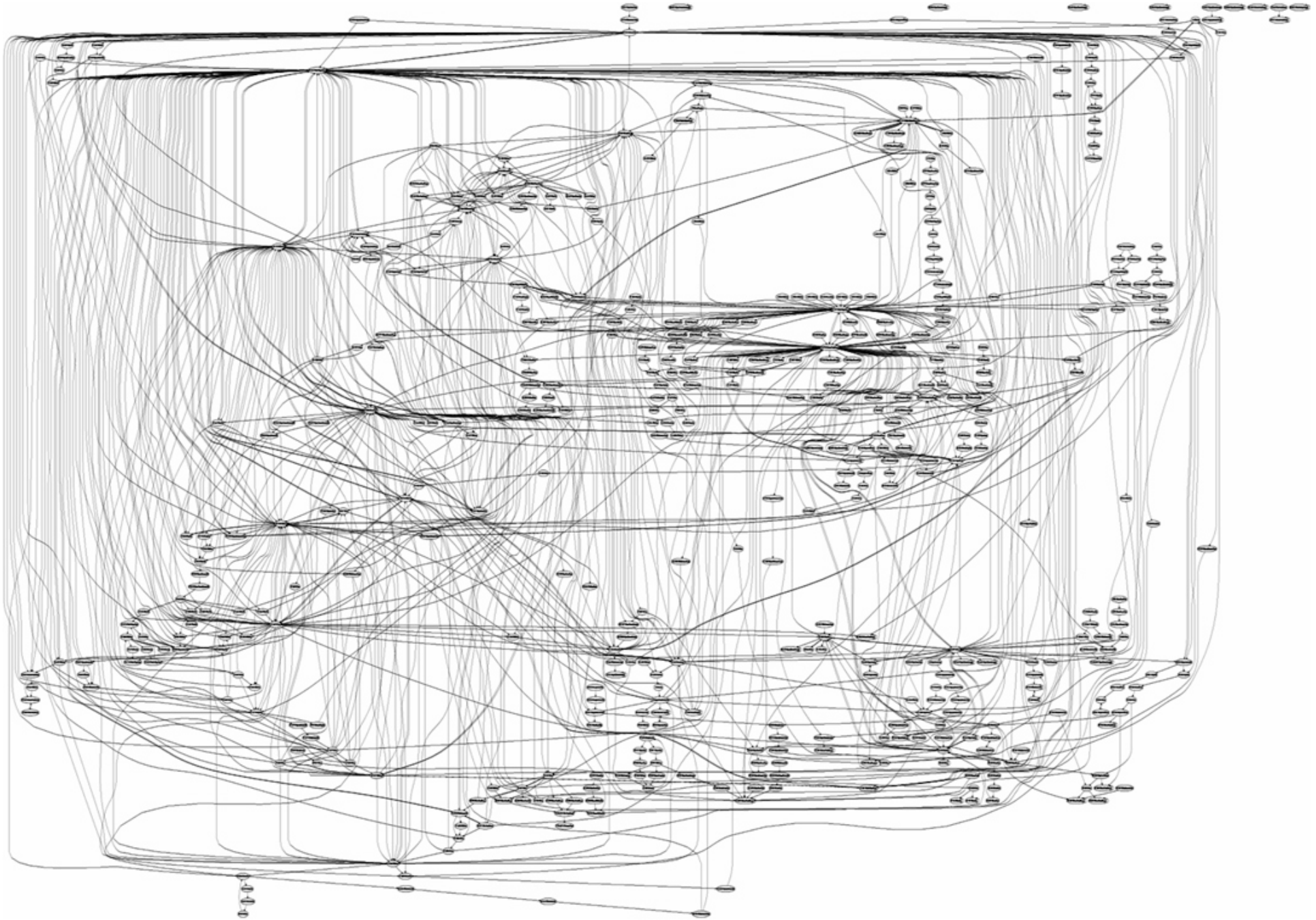
main.c ➡ .

Azureus

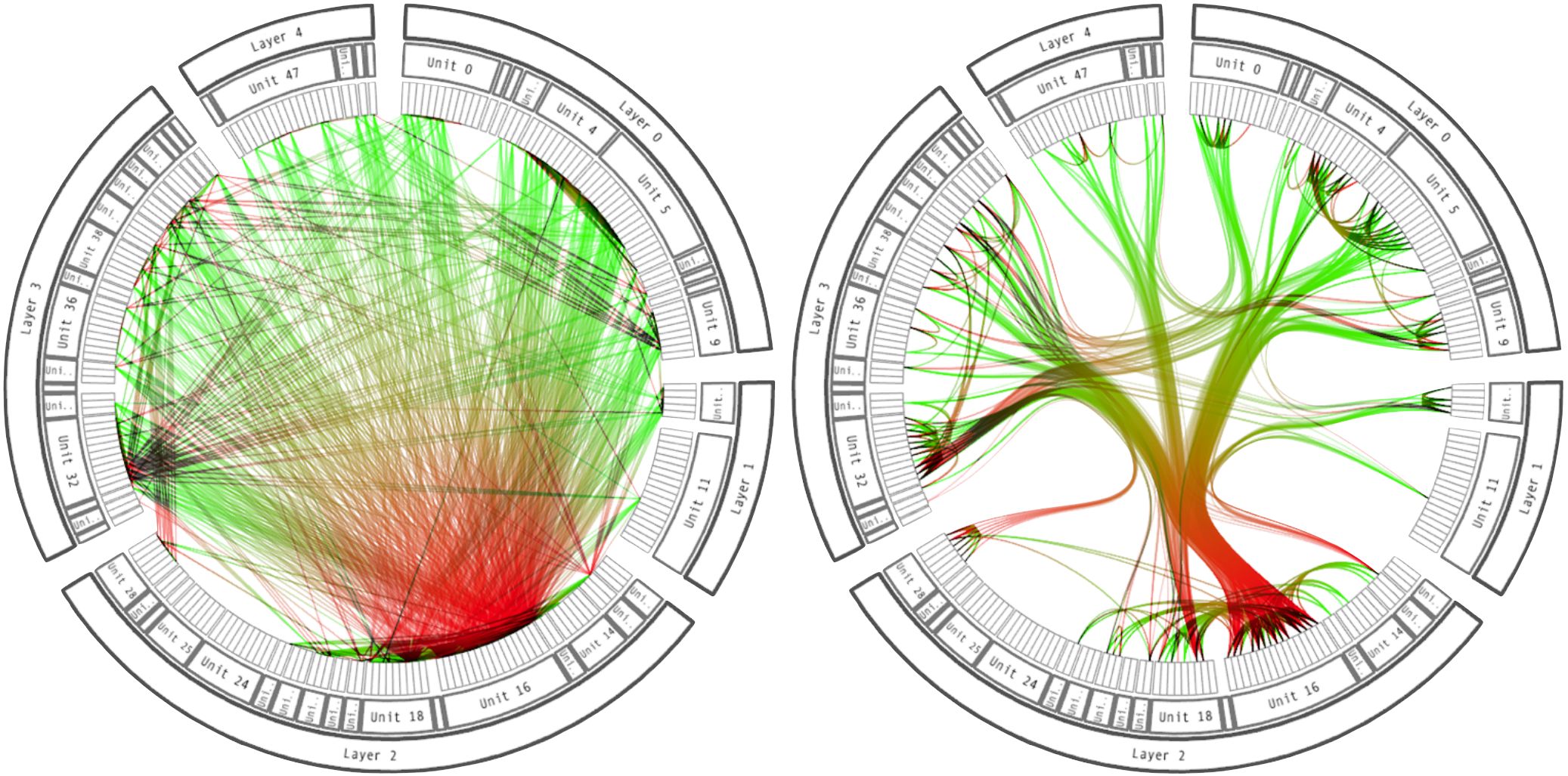
Azureus ➡ 

Windows

The relationships are complex too!



Software is invisible



There is still a long way to go ...

Software is malleable

It is easy to change software ... too easy?

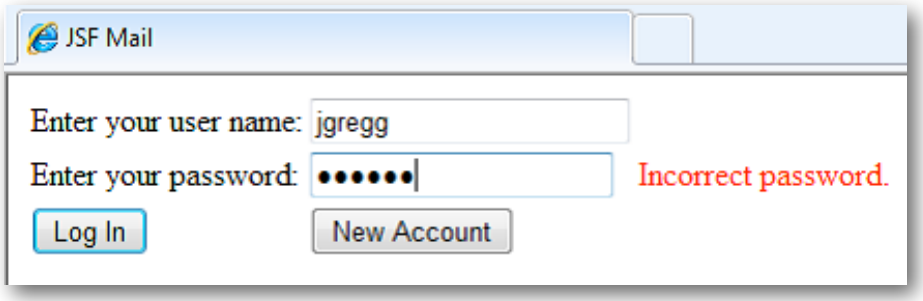
Ease of change is expected.
Hasty changes hamper maintainability.

Software must conform to its domain

Changes in the domain cause changes in the software.
Software engineers must possess some domain knowledge.

Software is discontinuous

People understand linear systems easily.



Some accidental difficulties

Choices of technology

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
DO ,1 SUB #3 <- #112
DO ,1 SUB #4 <- #0
DO ,1 SUB #5 <- #64
DO ,1 SUB #6 <- #194
DO ,1 SUB #7 <- #48
PLEASE DO ,1 SUB #8 <- #22
DO ,1 SUB #9 <- #248
DO ,1 SUB #10 <- #168
DO ,1 SUB #11 <- #24
DO ,1 SUB #12 <- #16
DO ,1 SUB #13 <- #162
PLEASE READ OUT ,1
PLEASE GIVE UP
```

```
;
```

```
World!");
```

Some domain aspects: mission-critical vs non-critical software

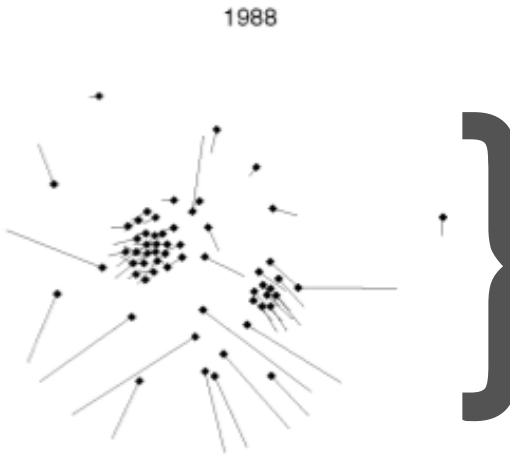
Software maintenance & evolution

Maintenance & evolution: facts and figures

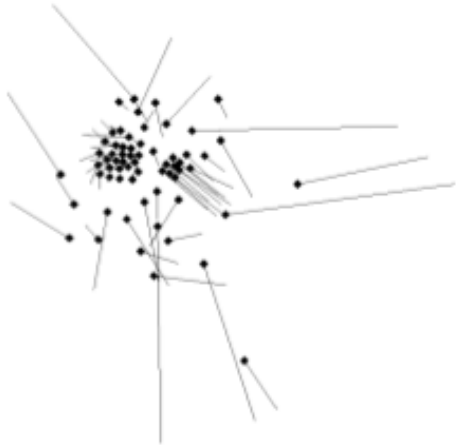
Lehman's laws of software evolution, first observed on IBM's OS/360

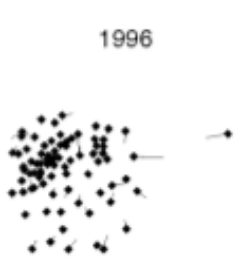
Continuing Change – Software systems must be continually adapted or they become progressively less satisfactory.
Increasing Complexity – As a system evolves, its complexity increases unless work is done to maintain or reduce it.
Self Regulation – The evolution is self regulating with distribution of product and process measures close to normal.
Conservation of Organisational Stability (invariant work rate) - The average effective global activity rate in an evolving system is invariant over product lifetime.
Conservation of Familiarity – As a system evolves, all associated with it, developers, users, ..., must maintain mastery of its content to achieve satisfactory evolution. Excessive growth diminishes that mastery, hence growth is invariant.
Continuing Growth – The functional content of a system must be continually increased to maintain user satisfaction.
Declining Quality – The quality of systems declines unless they are rigorously maintained and adapted to changes.
Feedback System – The evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Some visual examples of code decay

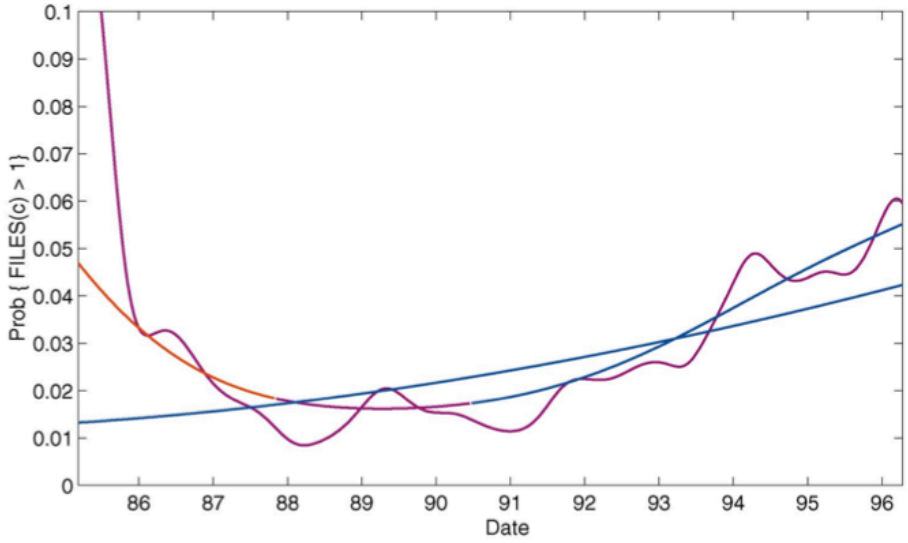


Two clusters





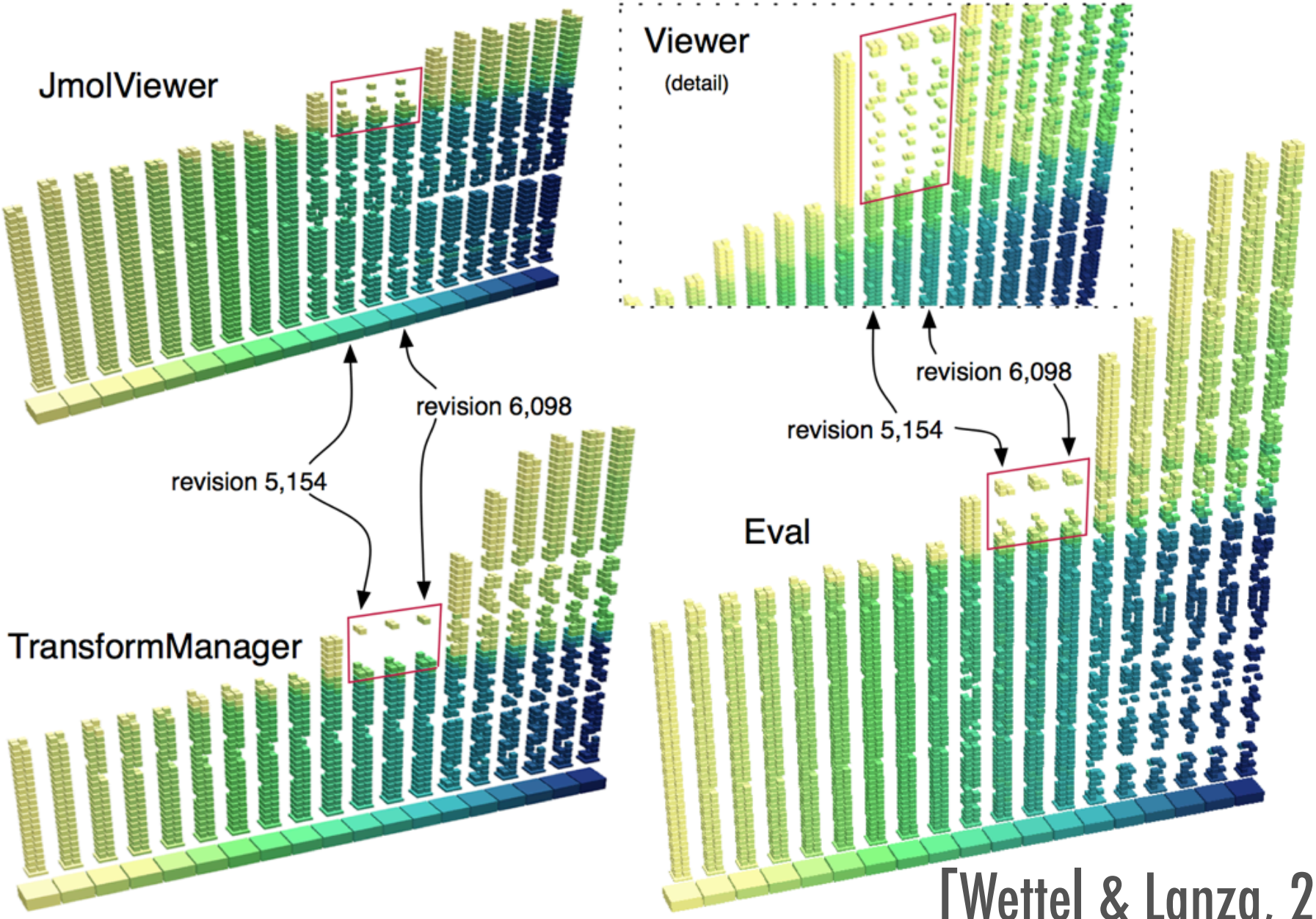
No clusters



#files changed for each MR

[Eick et al., 2001]

The evolution of several classes of Jmol



[Wettel & Lanza, 2008]

Estimates of the cost of software put maintenance at

90 %

[Erlikh, 2000]
40 % or more, according to [Brooks, 1975]

Y2K

Table I

Magnitude of the Y2K Mess

	Number of applications	Applications with Y2K problems (%)	Number of files	Files modified (%)	Screens and reports modified (%)	Hardware upgraded or replaced (%)	System software and utilities upgraded or replaced (%)
Average	1,090	65	9,900	33	31	27	49
Median	163	75	2,500	20	20	15	38

"I had the opportunity to track the progress and practices of hundreds of Y2K projects since 1996. [...] On average, these projects consumed more than 50% of one year's IS operating budget. This puts Y2K's total global cost between $375 and $750 billion. [...] About 45% of all applications were modified because of Y2K and another 20% replaced entirely."

[Kappelman, 2000]

Some other studies:

60% of the time is spent **understanding** existing code

[Corbi, 1989]

40% of bug fixes introduce **new** bugs

[Purushotaman and Perry, 2005]
20 to 50%, according to [Brooks, 1975]

Definition of a maintenance programmer

A programmer whose job mainly consists of changing code that someone else has written.

Categories of software maintenance

Perfective: introduce new functionality in the system
Adaptive: adapt the system to new circumstances (new OS, etc)
Corrective: fix software defects
Preventive: in prevision of future changes (refactorings, etc)

More than half of the changes are perfective. Introducing new functionality is also called software **evolution**.

[Swanson, 1975]

Categories of changes, by impact on functionality

Incremental changes: add new functionality, increase value.
Contraction changes (code pruning): remove obsolete functionality; no increase in value, but reduces bloat.
Replacement changes: replace a functionality with a newer one (bug fix); often an incremental change as well.
Refactoring: no change in functionality, but structural improvement; no increase in value, but increases maintenability.

Categories of changes, by impact on the system

Localized changes: Impact a single of few modules
(small bug fixes, change anticipated in the design)
Non-localized changes: Impacts a dozen (or more) modules
(significant new functionality)
Massively delocalized changes: Impacts a large part of the system
(fundamental design change, changes in the OS, Y2K)

Change strategies

"Quick fixes" done in urgency degrade the structure (copy-paste, etc...), and have long-term costs.

Changes should be well-designed and improve (through refactoring), rather than degrade, the structure of the software.

The software change process

Unlike design, maintenance work is on the actual software

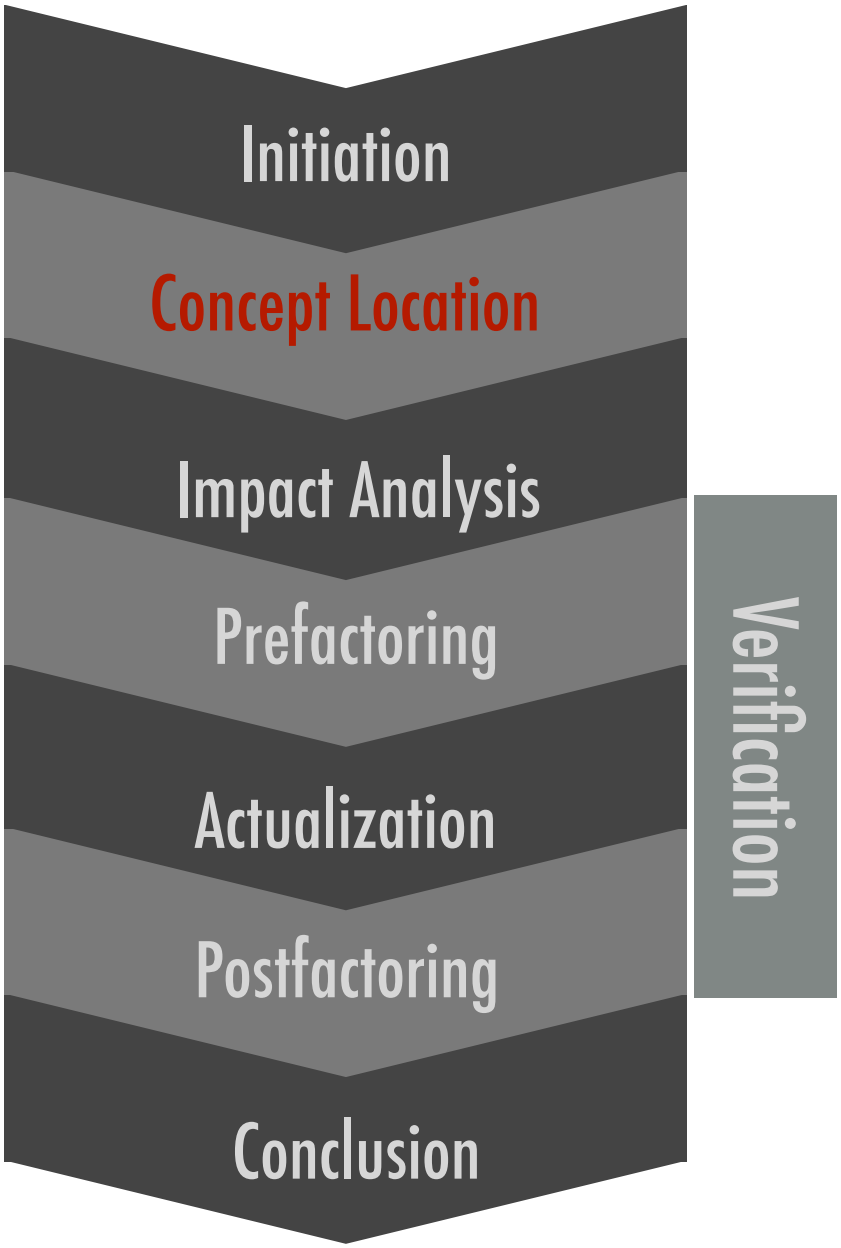
We need a process that works even if the quality of the code is not optimal

A typical software change process

Initiation marks the start of the process

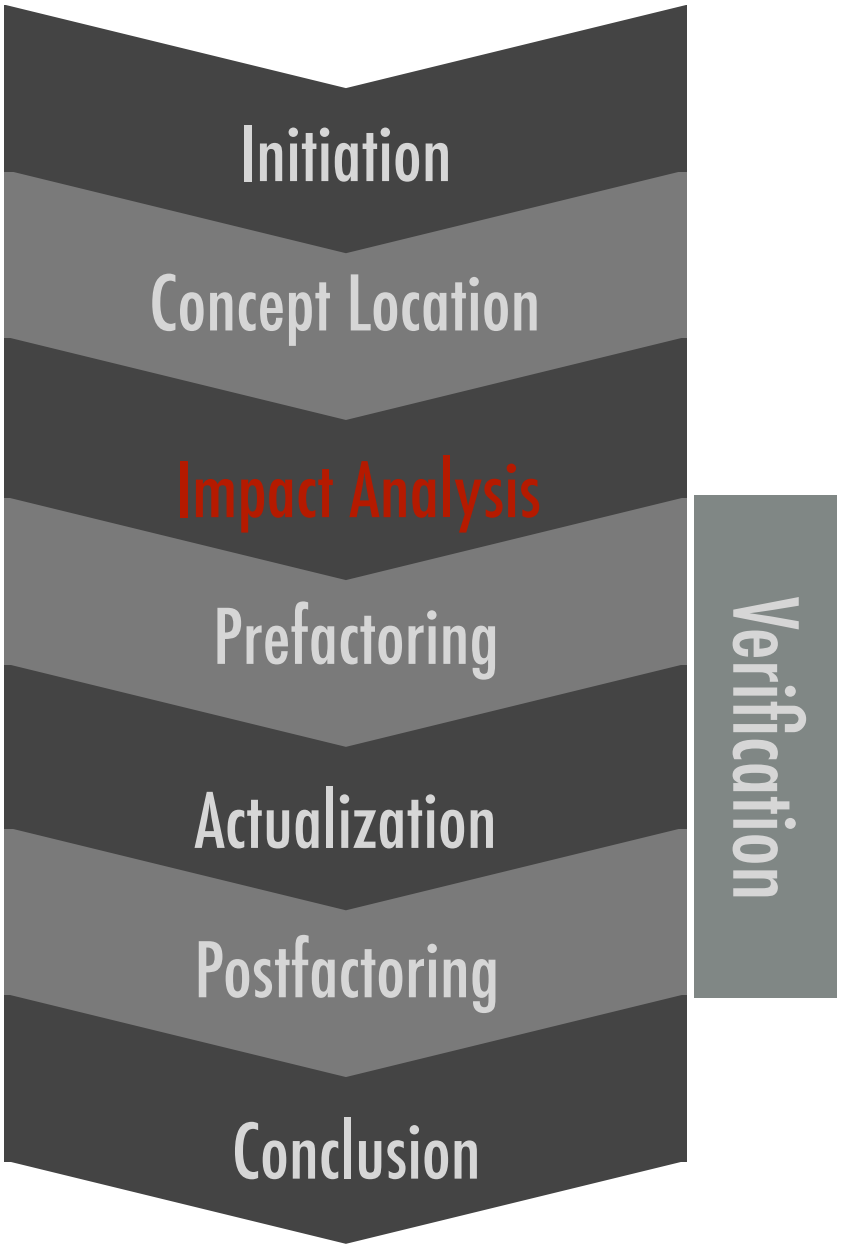
A new change request arrives, or is selected from the product backlog

Concept location finds a starting point



The programmer looks for the module that needs to be changed to implement the change request

Impact analysis estimates actual changes

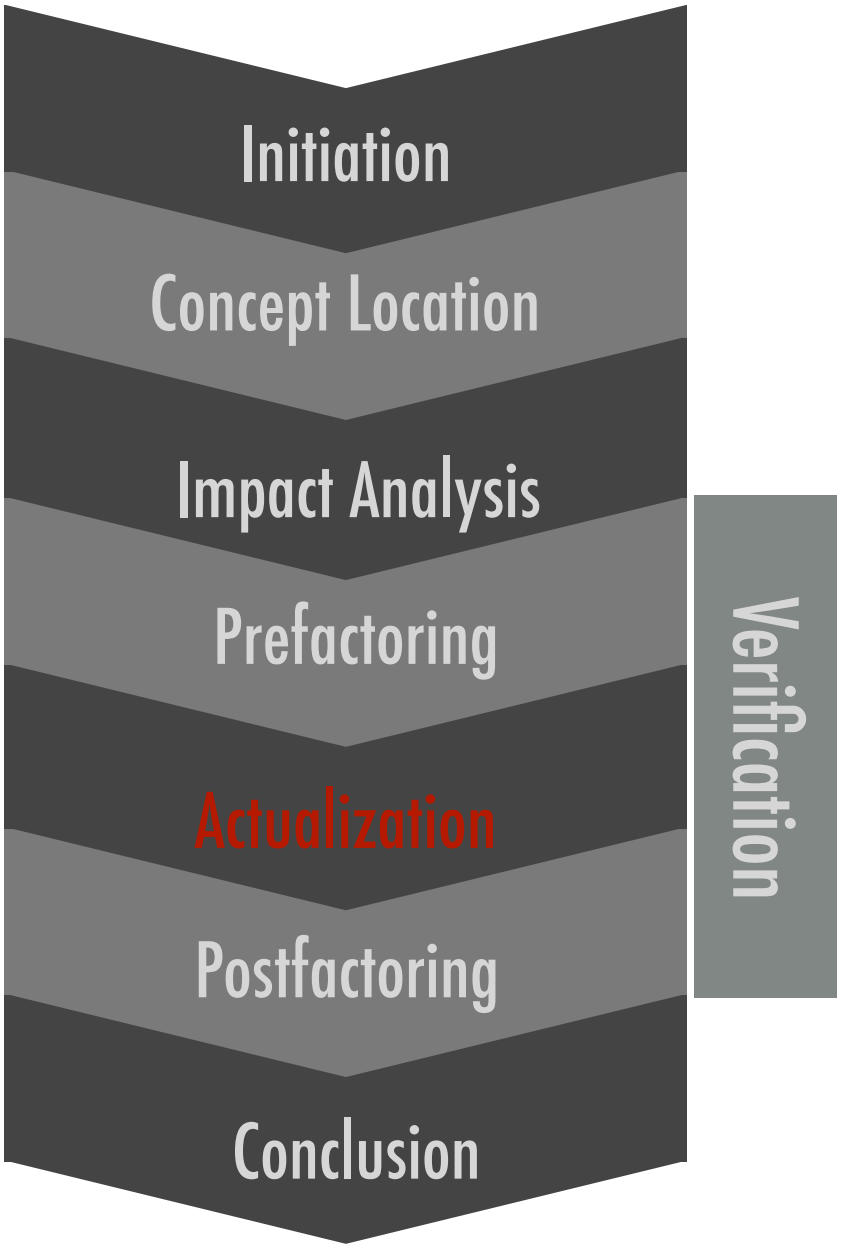


If the change is not localized, the programmer determines which other modules are affected by the change.

Then the change is implemented

The programmer decides of a strategy for the change and changes the source code

Actualization adds new functionality



The programmer updates modules, adds new modules, and propagate changes to affected modules identified by IA.

Refactoring improves the structure of code

The programmer makes the necessary changes to make actualization easier (prefactoring), and to prevent the decay caused by the change (postfactoring).

Verification ensures changes are correct

The programmer tests that the new functionality works correctly, and that the old functionality still works too!

Conclusion updates the project

The programmer commits the changes, creates a new baseline, updates the documentation.

Change initiation

Initiation marks the start of the process

A new change request arrives, or is selected from the product backlog.

What constitutes a change request?

- Change in requirements
- Bug reports
- Competing functionality
- Preventive refactorings
- ...

New functionality can be described in plain english, use cases, user stories ...
Large functionality should be split in several requests.
Change requests are managed in a product backlog.

The product backlog is a queue of requests, sorted by priority

New functionality are sorted by how much value they provide. Bugs are sorted by their severity:

- Fatal application error (showstopper)
- Serious bug affecting functionality (no workaround)
- Bug affecting functionality (with a workaround)
- Minor issue not involving primary functionality

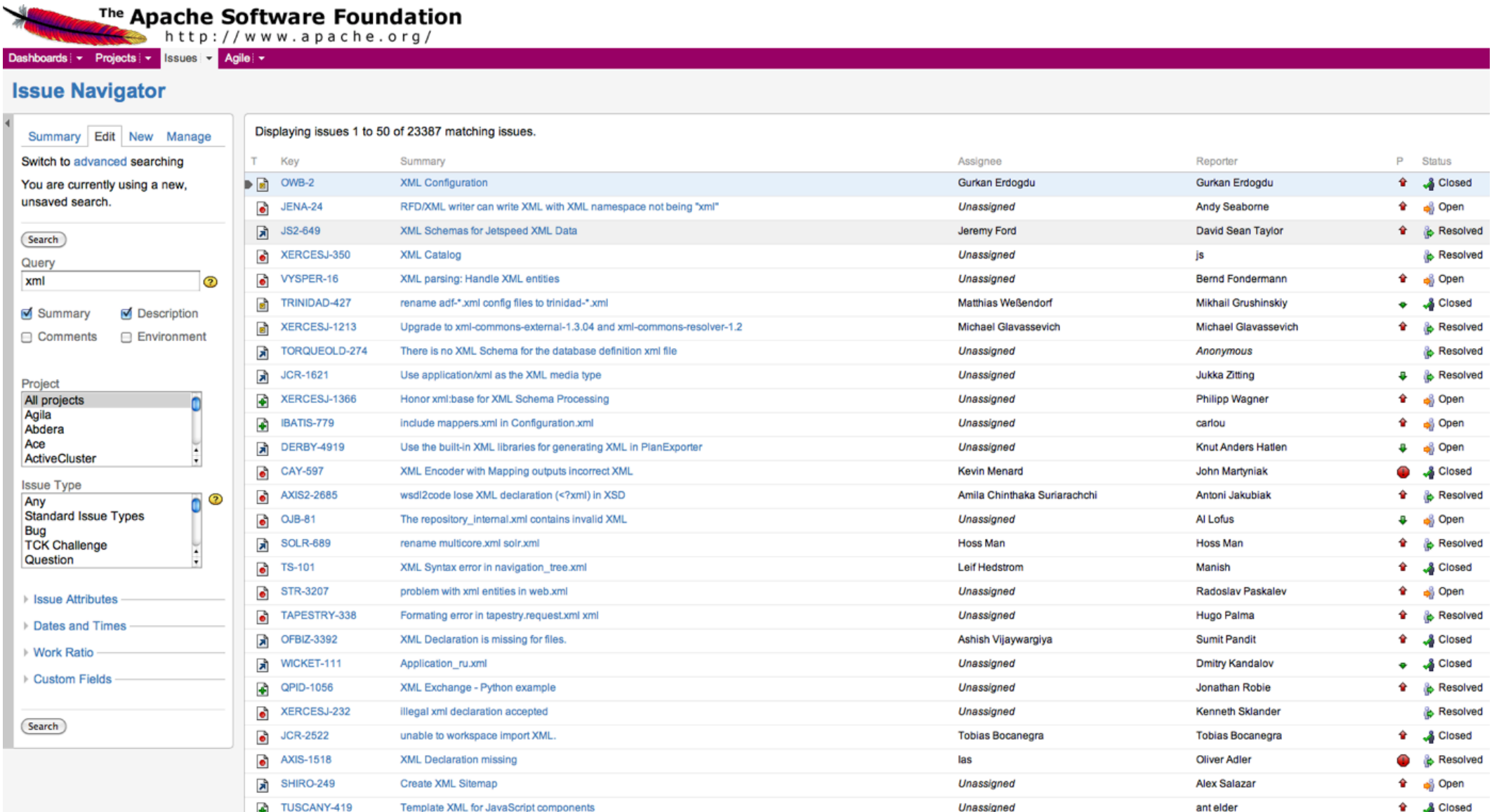
The first two levels are urgent. The next two are evaluated as any other request.

Expertise is another factor in choosing requests

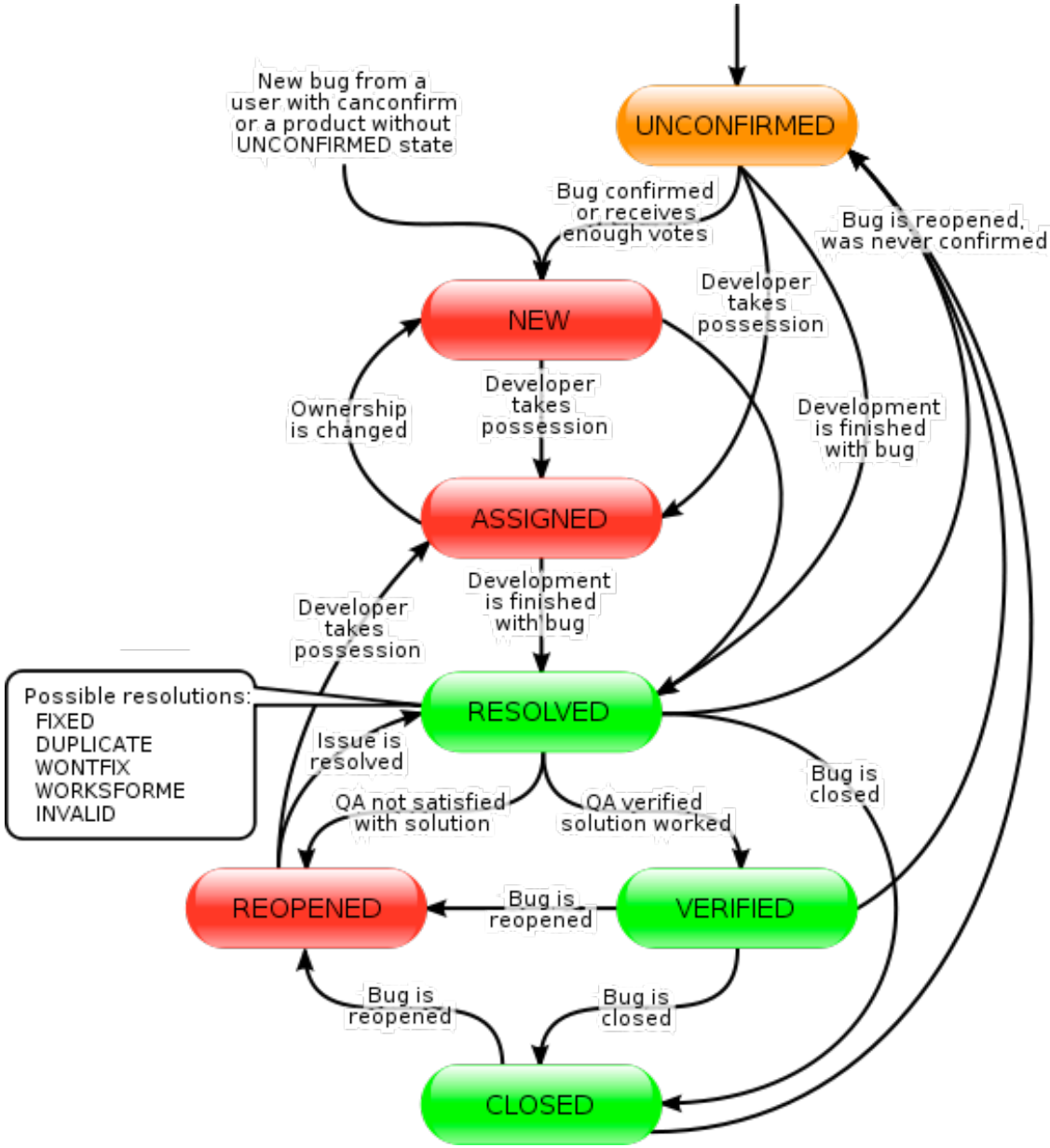
An expert in the domain or the affected code should be more efficient than a novice.

A cost/benefit analysis can be made (e.g., is this simple to fix for me, and providing enough value?)

The product backlog should be supported by a tool (Bugzilla, Jira, ...)

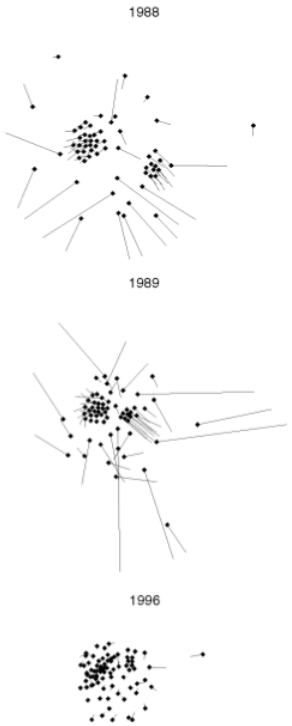


Statuses of a change request in Bugzilla



Conclusion

Software maintenance & evolution is the costliest phase in the software lifespan

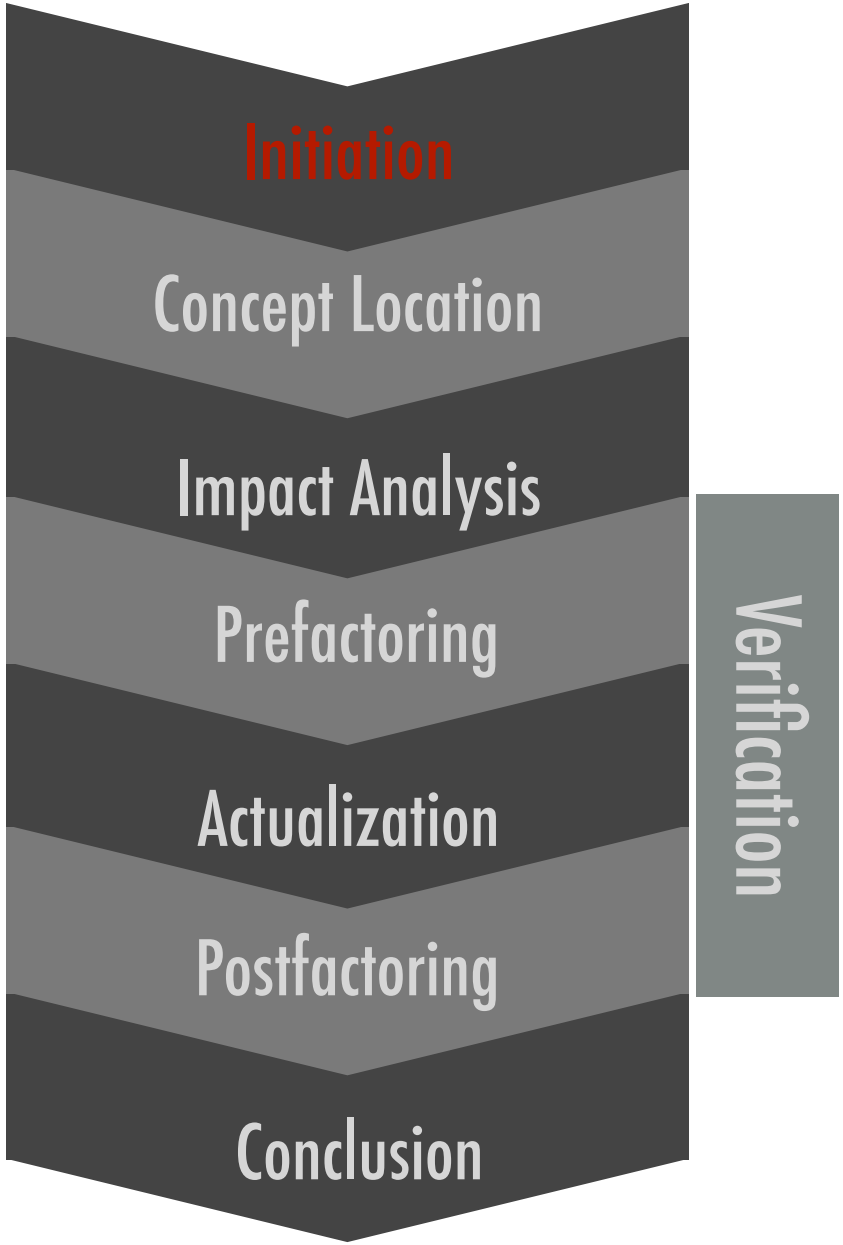


Over time, even the best thought-out design may decay, or be confronted to unanticipated changes (refactoring improves the situation).

Maintainers often have to understand unfamiliar code.

Software maintenance seeks to keep the complexity under control in the presence of necessary changes.

Change process supports maintainers



Initiation is the first activity of the process, where a change request is selected for implementation.

We will then talk about the next phases.