

Gestion de Projet et Génie Logiciel Master 1, Lyon 1

Matthieu Moy

UCBL

2018-2019



Objectives

- Software engineering (Génie Logiciel): the art of making “good” software, at reasonable price
- Project management: what's left, other than development? (need analysis & specification, team management, ...)
- Approach:
 - ▶ Good practices: tests, design patterns, ...
 - ▶ Tooling: continuous integration, version control, unit testing, ...
 - ▶ Organization: project lifecycle, agile methods
 - ▶ Speakers from several external companies
- Many notions usable directly ⇒ apply in your school project, your company (apprentices), ...



Practical Aspects

- Course material:
 - <https://forge.univ-lyon1.fr/matthieu.moy/mlif01>
 - ▶ In your web browser
 - ▶ git clone
 - <https://forge.univ-lyon1.fr/matthieu.moy/mlif01>
 - once, git pull periodically.
 - ▶ Slides, exercises, labs, examples of code
- Evaluation:
 - ▶ Final exam
 - ▶ Mini-project: deadline = Oct 6th 2018, 20:00 (TOMUSS)
- Schedule: see ADE



Assumed to be Previously Learnt

- Object-Oriented Programming basics:
 - http://tabard.fr/courses/2015/mif17/2015/MIF17_Rappel_objet.pdf
- UML Modeling basics:
 - ▶ Class diagrams (classes, object, package, etc.):
 - <http://tabard.fr/courses/2015/mif17/2015/UML-Statique.pdf>
 - ▶ Dynamic diagrams (sequence, state-machine):
 - <http://tabard.fr/courses/2015/mif17/2015/UML-Dynamique.pdf>



Outline

- Generalities (today, tomorrow)
- Tools for code management
- Use-cases
- Agile methods
- Design-patterns
- Tests
- Ethics
- Project management



Distribution

- 8×2h Lecture (CM)
- 2×2h Tutorial (TD)
- 5×2h Labs (TP)



Quality Criteria for Software

- Some criteria for the user
 - Reliable** Gives the expected result,
 - Robust** Doesn't crash, behaves reasonably in unexpected conditions,
 - Efficient** Gives the result quickly,
 - User-friendly** Easy to use,
 - Secure** Ability to resist to malicious uses.



Quality Criteria for Software

- Criteria for the developer
 - Readable** Easy to read, to understand. Well documented,
 - Maintainable** Easy to modify, to fix,
 - Portable** Runs on various systems,
 - Extensible** Easy to improve,
 - Reusable** Can be adapted to other applications.



Software Crisis

Cost of software is always important and higher than expected.

- Success of projects (Standish group, CHAOS Summary 2014):

	Number of projects
Abandoned, never used	31%
Late, over budget, incomplete	52%
On time and on budget	16%

- Usage of features (Waste In Software Projects,

<http://thecriticalpath.info/2011/07/07/waste-in-software-projects/>):

Use of features	Number of features
Never used	45%
Rarely used	19%
Sometimes	16%
Often	13%
Always	7%



Why is it so hard?

Software development is easy:



<https://www.flickr.com/photos/jacobavanzato/16152519186>



Why is it so hard?

A first prototype is easy to get ...



<https://www.flickr.com/photos/78044378@N00/364003706>



Why is it so hard?

... but how do we extend it? How do we scale?



<https://www.flickr.com/photos/10402746@N04/7165270428>



Scale? How big?

- Typical lab work = 100 LOC
- Typical school project = 1000 LOC
- 100 developers, 100 LOC/day¹, 10 years = 20 MLOC
- Linux Kernel 4.13: 20 MLOC
- Facebook: 60 MLOC
- Windows: 3 million files
- Google's full codebase: 2 Billion LOC

Nice visualization:

<https://informationisbeautiful.net/visualizations/million-lines-of-code/>

↪ Not all programs are that big, but most programs are orders of magnitude bigger than what you've experienced so far.

¹Optimistic estimate, e.g. COCOMO model estimates to 10-20 LOC/day
⇒ > 10 \$/LOC ...



How buggy is it?

"Industry Average: about 15 – 50 errors per 1000 lines of delivered code."

"Microsoft Applications: about 10 – 20 defects per 1000 lines of code during in-house testing, and 0.5 defect per KLOC in production."

(Source: Steve McConnell book, "Code Complete")



A few Famous Bugs

- Therac-25: radiotherapy machine **killed** at least 6 persons
<http://cr4.globalspec.com/blogentry/19025/Failure-of-the-Therac-25-Medical-Linear-Accelerator>
- Ariane 5 crash: **arithmetic overflow** ⇒ self-destruction of the shuttle ⇒ most expensive firework ever (≈ 500 M\$). Ironically: well-tested software (re-used from Ariane 4), hardware redundancy (both computers crashed).
<http://www-users.math.umn.edu/~arnold/disasters/ariane.html>
- Mont Saint-Odile's crash: "the pilots inadvertently left the autopilot set in Vertical Speed mode (instead of Flight Path Angle mode) then entered "33" for "3.3° descent angle", which for the autopilot meant a descent rate of 3,300 feet (1,000 m) per minute." ⇒ a **UI bug** that costed 87 lives.
https://en.wikipedia.org/wiki/Air_Inter_Flight_148
- Mars Climate Orbiter: "software which produced output in non-SI units of **pound-force** seconds (lbf-s) instead of the SI units of **newton**-seconds (N-s) specified in the contract between NASA and Lockheed" ⇒ Incorrect interpretation of specification = ≈ 300M\$ crash
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter



OK, these were critical systems. I just write ERPs/Accounting software

Well, you're in the game too!



Louvois: payment system for french militaries

- “Pendant près de 7 ans, de nombreux soldats français ont vécu l'enfer. Non pas sur le terrain mais à cause de leur fiche de paie”
- Started in 2011, affected 120,000 employees.
- Over and under-payments.
- More complex than it seems: 174 different kinds of bonus
- “C'était courru d'avance”, “De 150 points de vérification, on est passé à 15 pour tenir les délais”
- ⇒ complete rewrite decided. No fix available/possible.
- “Un logiciel nouveau comme ça, il faut au moins trois ans pour l'installer”, “Il y aura donc bien un an de retard, et peut-être plus si l'on en croit d'autres sources.”, “On prévoit le lancement au 1er janvier 2019”

~> Not just “one unfortunate bug”: a 7-years failure due to initial bad management choices.

<https://www.franceinter.fr/emissions/secrets-d-info/secrets-d-info-27-janvier-2018>

Failure of “Agile” Contract at Macif

Alors qu'elle était plaignante, la Macif vient d'être condamnée à payer à un éditeur de logiciel 1.45 millions d'euros” (et 4 ans de développement).

<https://www.linkedin.com/pulse/saffranchir-du-cycle-en-v-agile-canada-dry-ou-comment-maxime-blanc>

Keep in Mind ...

- 1 Computer systems are complex and require good methods
- 2 Methods must combine rigor and adaptation to unknown and to change

UML Modeling?

- Modeling:
 - Helps informal discussions between developers (e.g. quick and dirty diagrams on white board)
 - Helps rigorous specifications
 - Helps deriving implementation (manually or automatic)
- Modeling is not sufficient:
 - Need for design and programming techniques
 - Write readable and reusable code

Software Lifecycle

- Requirement analysis and definition
- Analysis and design
- Coding/Debugging
- Validation
- Evolution and Maintenance

Software Lifecycle

- Requirement analysis and definition
 - specifications
 - feasibility study (may involve a prototype)

Software Lifecycle

- Analysis and design
 - Specification
 - Architecture (= hard-to-change decisions)
 - Detailed design (algorithms, data-structures)

Software Lifecycle

- Coding

Software Lifecycle

- Validation: make sure the program “works”
 - Static analysis and proof
 - Code review (very efficient)
 - Tests (essential)



Software Lifecycle

- Evolution and maintenance:
 - Corrective maintenance (Bug fixing)
 - Adaptive maintenance (Porting, ...)
 - Evolutionary maintenance (New features, ...)

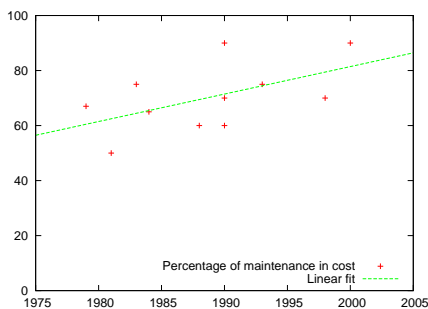
“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”



Effort distribution

<http://users.jyu.fi/~koskinen/smcosts.htm>

- Part of Maintenance in Total Cost:



⇒ better optimize for maintainability than for initial development

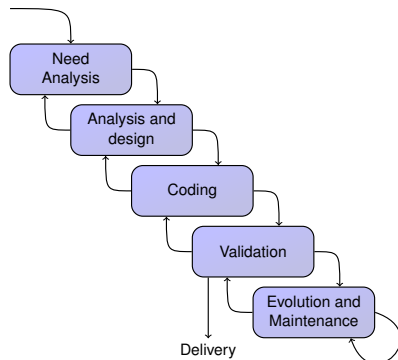
Effort distribution in Initial Development

As a rule of thumb ...

- Initial development:
 - Requirement analysis, architectural design: 40%
 - Coding, debugging: 20%
 - Validation: 40%

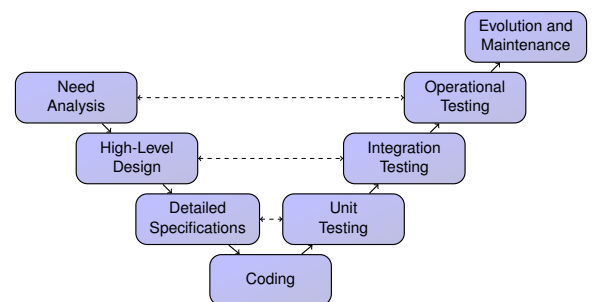


Waterfall Lifecycle



V Lifecycle

Variant of the Waterfall Lifecycle



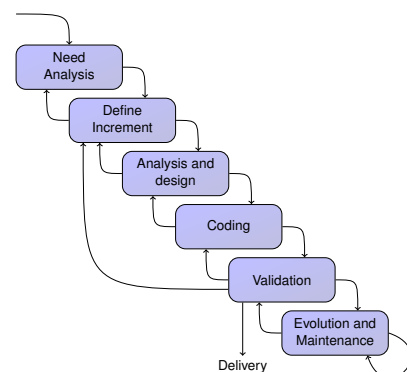
Waterfall/V Lifecycle

- Guiding principle: Interactions occur only between two successive states.
- Advantages:
 - Clean design (hopefully), no evolution within initial development.
 - Contractualization: specifications and effort estimates are known (hopefully) early
- Drawbacks:
 - Defect in first steps can have catastrophic consequences
 - Validation of specification late in the design
 - Integration late in the cycle ⇒ most risks eliminated late
 - Hardly parallelizable
 - Perfect in theory, but not adapted to humans?
- Variant: de-risk with “W”-lifecycle

“The management question, therefore, is not whether to build a pilot system and throw it away. You *will* do that. [...] Hence *plan to throw one away; you will, anyhow.*” (The Mythical Man-Month, 1975, Brooks)

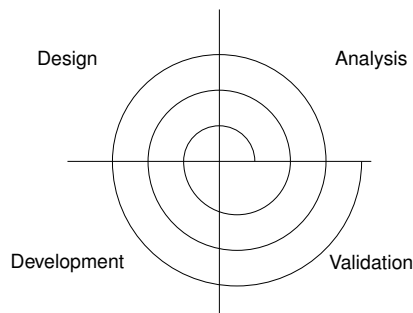


Incremental Lifecycle



Spiral Lifecycle

Another view of the Incremental Lifecycle



Incremental Lifecycle

- Guiding principle: divide the program in small amounts of analyzed, coded, and tested features.
- Advantages:
 - ▶ Early discovery of problems,
 - ▶ Early availability of prototypes (essential to get feedback from the client),
 - ▶ Helps continuous validation,
 - ▶ Allows time-based releases, as opposed to feature-based releases.

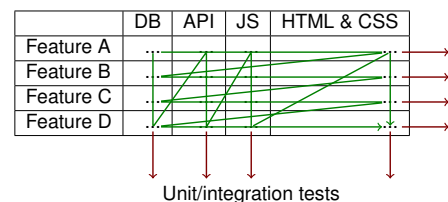
Specifying the increment

- Informally
- With a subset of the specification (if it exists)
- With a use-case (or "user story")
- With a set of tests
 - ⇒ **Test Driven Development**
 - while true loop
 - write tests
 - make sure they don't pass
 - implement feature
 - debug until test pass
 - commit and push
 - end loop



Planning and Architecture with Iterative Development

Example with a typical web application: each feature may need entries in the database, modification to an internal REST API, JavaScript for the client-side logic and HTML&CSS for rendering.



Layer-by-layer ~ First usable prototype after $\approx 90\%$ work is done
 Feature-by-feature ~ Prototypes/releases available all along development



Organize the Work

- **Principles:** general guideline. Examples:
 - ▶ "Our highest priority is to satisfy the customer[...]" (Agile Manifesto)
 - ▶ "every module or class should have responsibility over a single part of the functionality provided by the software" (Single Responsibility Principle)
 - ▶ "The process of developing software consists of a number of phases." (Software Development Life Cycle common principles)
- **Practices:** concrete things one can do. Examples:
 - ▶ Pair Programming
 - ▶ Test Driven Development
 - ▶ Code Review
 - ▶ Refactoring
 - ▶ DevOps
- **Methods/methodologies:** set of practices and how they are organized. Examples:
 - ▶ Waterfall
 - ▶ Merise (\approx ancestor of UML)
 - ▶ Scrum



Principles? Practices? Methods?

How to (not) Get the Best of it

- You may agree with principles and dislike the associated method
- Good practices of one method may apply to other methods
- Applying a method without understanding its principles is doomed²
- No silver bullet: a method that works in a context may fail in another
- A successful method needs/attracts consultants to train people on this method \Rightarrow creates business \Rightarrow creates marketing.

²Remember the Macif example above?



Methods in Software Engineering

- Main classes of methods:
 - ▶ Strategic management methods
 - ▶ Development methods
 - ▶ Project management methods
 - ▶ Quality assurance and control methods
- Development methods to:
 - ▶ Build operational systems
 - ▶ Organize the work
 - ▶ Manage the project's lifecycle
 - ▶ Manage costs
 - ▶ Manage risks
 - ▶ Get repeatable results



Evolution of methods

4 successive trends:

- Modeling by functions
- Modeling by data
- Object-oriented modeling
- Agile methods



1st generation: function-based modeling

- Decompose a problem into sub-problems: functions, sub-functions, ...
- Each function defines inputs and outputs
- Examples: IDEF0, SADT
- When one function changes, everything may change



2nd generation: data-based modeling

- “Systemic” approaches
- Information system = structure + behavior
- Model the data, how they are organized (e.g. UML entity-relationship diagrams, MERISE).
- Data-modeling important with (R)DBMS.
- Behavior modeled separately
- Strengths:
 - Data consistency, abstraction levels well-defined
- Weaknesses:
 - Lack of consistency between data and behavior
 - Pushes towards long lifecycles (V)



3rd generation: object-oriented modeling

- “Systemic” approaches with data/processing consistency
- Set of objects that collaborate, considered statically (what the system is: data) and dynamically (what the system does: functions).
- Functional evolution possible without changing the data
- Modularity through abstraction and encapsulation



Current trend: agile/adaptive (≠ predictive) methods

- Short iterations (e.g. demonstrate new features to the client every week)
- Strong and continuous interaction with client
- Value responding to change over following a plan
- Self-organized teams
- Adaptive: retrospective and adaptation periodically



Tools ...

- Necessary for development (compiler, text editor)
- Catch mistakes early (tests, code analysis)
- Automate stuff (I'm lazy, too)



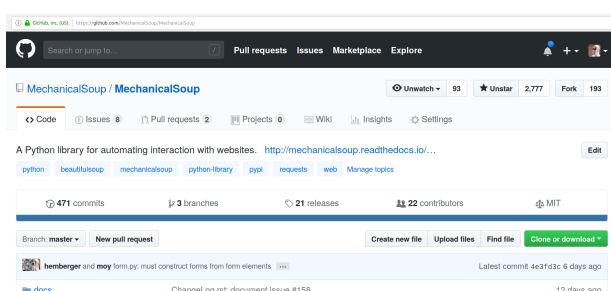
A Small Example: MechanicalSoup (Python library)

Disclaimer: I'm one of the authors

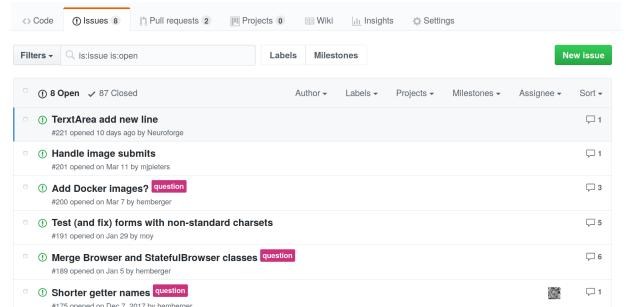
- Small library (400 LOC of Python + 1000 LOC of tests) for browsing websites
- Small, developed on free-time ⇒ no planning, no real methodology
- Tries to follow best practices and uses many fun tools
- Let's go through a few of them... (you can do similar things with Lyon1's GitLab)



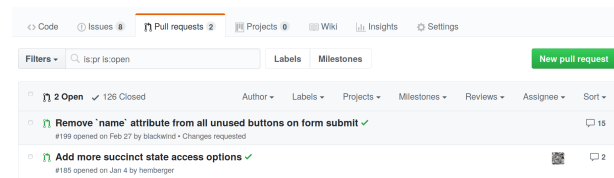
Hosting: Git, GitHub



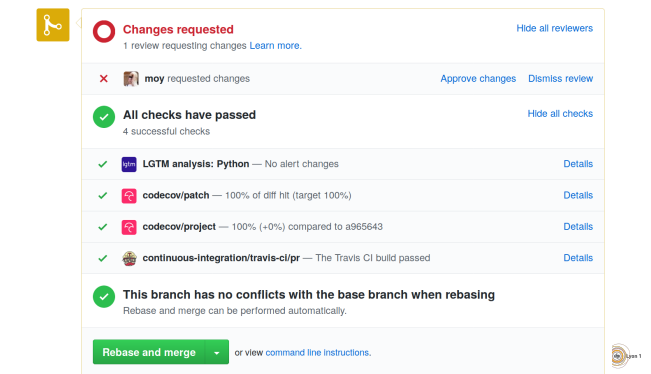
Report bugs, discuss future features: issue tracker



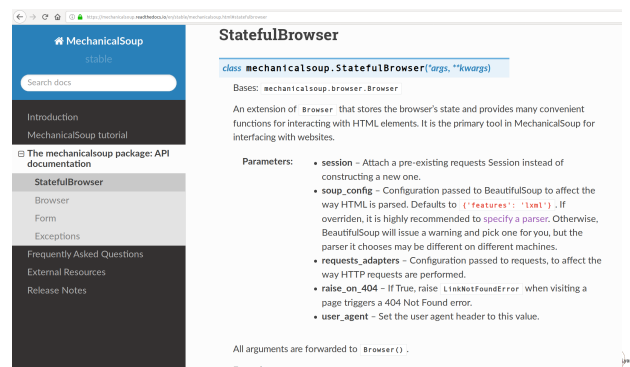
Submit code: pull-requests



Automated checks on pull-requests



Documentation automatically generated at each push



About pull-requests and checks ...

- Anyone can submit a pull-request
- Pull-requests trigger build + testsuite + coverage check + style check + static analysis
 - ▶ Test failing ⇒ fail
 - ▶ Incompatibility with one supported version of Python ⇒ fail
 - ▶ Incorrect style (lines >80 characters, mis-placed space, ...) ⇒ fail
 - ▶ Line of code not covered by a test ⇒ fail
 - ▶ Bad pattern detected by code analysis ⇒ fail
- How we did all that? Mainly "use tools/services" and 30-lines long .travis.yml file.

Automated testing

(Because life is too short to spend time on manual testing)

- Code that tests code:

```
def test_no_404(httpbin):
    browser = mechanicalsoup.StatefulBrowser()
    resp = browser.open(httpbin + "/nosuchpage")
    assert resp.status_code == 404
```
- General form of automated tests:

```
def name_of_test_function():
    # given
    some_object = ...
    # when
    some_object.some_action(...)
    # then
    assert ...
```

Next Course: More on Tooling

- Build a complex Java project with gazillions of dependencies without pain
- Discover the awesomeness of GitLab
- Apply in your lab works!