

# Software Engineering: Basics

Projet GL

Ensimag  
Grenoble INP

December 9, 2022



# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

# Software Engineering

Parallel with:

- Military engineering (oldest one): fortresses, siege & war machines
- Civil engineering: buildings, bridges
- Mechanical engineering
- Electrical engineering
- Chemical engineering...
- Software engineering & bio-engineering: the newest ones

**Engineering:** engineers are experts who

- master scientific and technical bases
- are able to design and guarantee quality
- organize the tasks and processes

# Dual aspects of Software Engineering

## Software engineering of *products*

- Tools and methods to make software *artefacts*
- Artefacts can be: source code, binary code, data structures or repositories, architecture diagrams etc.
- Example of tools: text editor (vim, Atom), IDE (Visual Studio), compiler, debugger, source code generator (ANTLR), model checkers (Avispa), link editor, configuration management (make, Maven), test harnesses, coverage analyzers (Jacoco), etc.

## Software engineering of *processes*

- Methods to organize the production tasks
- In the case of software: mostly work of humans (developers)
- Backbone organization: known as (software development) “(life)cycles”
- Examples: Waterfall lifecycle, V lifecycle, Agile development

# Ensimag Engineers

**Engineers:** are experts who

- master scientific and technical bases: **Computer science**
- are able to design and guarantee quality **Projet GL**
- organize the tasks and processes **SHEME, project Management**

What you will learn and practice (**main skills**)

- Software tools: git, mvn, antlr, log4j, IDE, gitlab, junit, jacoco...
- Languages & artefacts: Java, scripts, grammars, architecture diagrams, tests...
- Management methods: Agile, gantt planners, reporting...
- Working in a team, adapting to peoples' strengths and weaknesses

# Ensimag Future Engineers

What you may start to discover (and learn further at Ensimag)

- Floating point computation
- Ecological impact of digital world
- Testing methodologies (more advanced than project)
- “Bowels” of computing: bytecode, binary (FP), link editing...
- ...

# Quality Criteria for Software

- Criteria for the user
  - Reliable** Gives the expected result,
  - Robust** Doesn't crash, behaves reasonably in unexpected conditions,
  - Effective** Gives the result quickly,
  - Efficient** Uses a minimum of resources
  - User-friendly** Easy to use.
- Main focus for us: **Reliable**
- Secondary focus: **Efficient** (w.r.t. energy)

# 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.

- Third focus for you: **Readable**



# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

# Outline of this section

- 2 Software Lifecycle
  - Stages in the Lifecycle
  - Software Lifecycle Modeling

# Software Lifecycle Stages

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

# Software Stages: Requirements

- Requirement analysis and definition
  - ▶ high level specifications
  - ▶ feasibility study

## Projet GL:

- Decac compiler: specifications are ready (just read them)
- Extension: specs. are negotiated with teachers

In real life, discussing requirements with customers is an important task (time consuming and critical).

# Software Stages: Design

- Analysis and design
  - ▶ Detailed specification  
(for us, this is booklet part II)
  - ▶ Architecture  
(for us, 3 stages, Java packages, ...)
  - ▶ Detailed design (algorithms, data-structures)

## Software Stages: Coding

- Coding: translating algorithms into programming language
- Debugging: compiling and exercising the code to check it

## Software Stages: Coding

- Coding: translating algorithms into programming language
- Debugging: compiling and exercising the code to check it

Beware: testing is NOT debugging

- Debugging done by developer to check whether the lines of code are actually written as he/she meant.
- Testing done by testing team to check whether program behaves as specified.

# Software Stages: Validation

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



# Software Stages: Maintenance

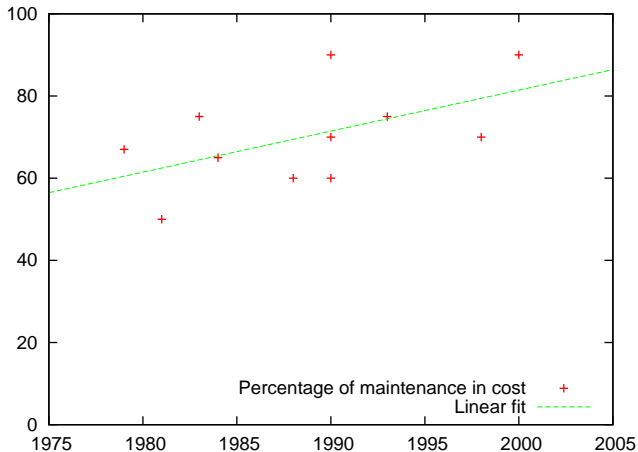
- Evolution and maintenance:
  - ▶ Corrective maintenance (Bug fixing)
  - ▶ Adaptive maintenance (Porting, ...)
  - ▶ Evolutionary maintenance (New features, ...)

“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%
    - ★ `#!/` debugging is not part of validation.
    - ★ Validation starts when the code looks correct.
  - ▶ Validation: 40%

## And for our project

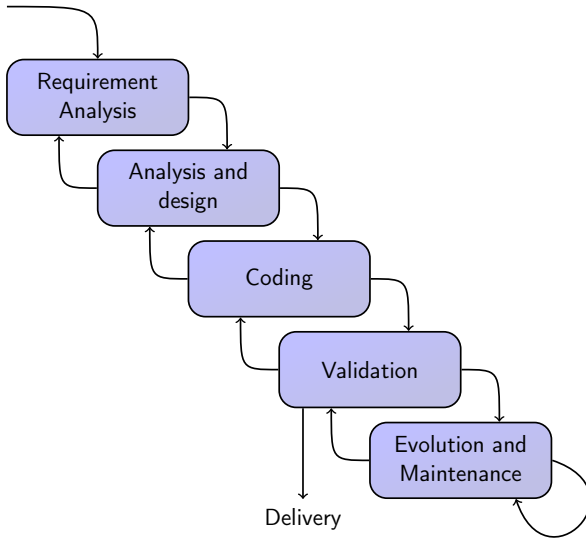
- Analysis, architectural design: 15% (reading, splitting into packages, ...)
- Detailed design, coding, debugging: 20%
- Validation: 40% (including scripting)
- Extension: 25%, of which 40% on analysis

And time will also be used by management, synchronizing with team and with professors.

# Outline of this section

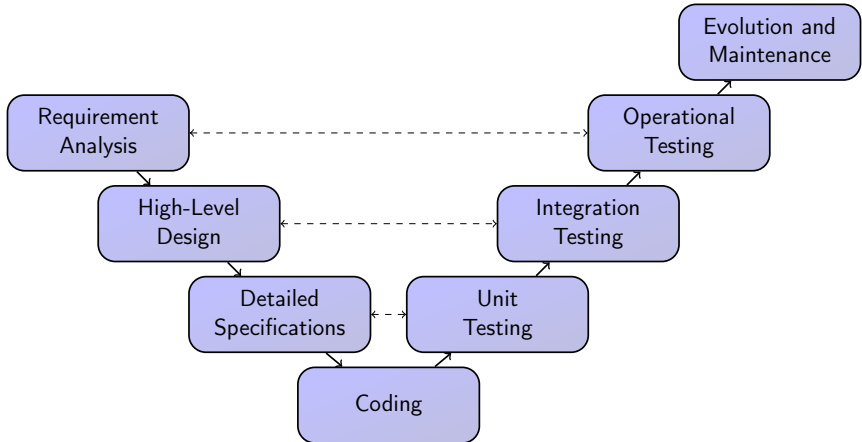
- 2 Software Lifecycle
  - Stages in the Lifecycle
  - Software Lifecycle Modeling

# Waterfall Lifecycle - Historic



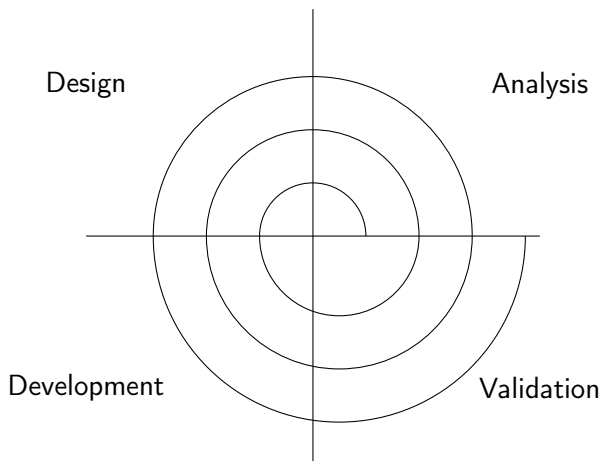
# V Lifecycle - Large critical systems

Stresses Validation



# Incremental (Spiral) Lifecycle

Typically in Agile development



# 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 set of rules in Deca's grammar
- 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


## Examples of increment

- First goals:
  - ▶ Compile the empty program
  - ▶ Compile a hello-world
- Without objects:
  - ▶ Simple expressions ( $2+2$ ,  $2-2$ , ...)
  - ▶ Variables (int, float)
  - ▶ Control-structures (if/while)
- Objects:
  - ▶ Objects without methods
  - ▶ Methods (definitions and calls)

Planning should be driven by **language subsets**,  
not by stage/passes (B1, B2, B3, C1, C2)

## Incremental Lifecycle in our Project

- Hardly applicable to stage A (too short)
- Mandatory for stages B and C (type-checking and code generation).
- Avoids big-bang validation right before the final deadline
- Avoids half-done or untested features at the final deadline
- Necessary to get the **intermediate delivery** on time

 prefer complete and well-tested compiler  
on a subset of the language  
to a buggy feature-complete compiler.

# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 **Validation**
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

# Validation

- Validation means making sure the program is
  - ▶ Correct
  - ▶ Robust
  - ▶ Efficient
  - ▶ Readable
  - ▶ Usable
  - ▶ Documented

Reminder: focus in our project is on **Correctness** (reliability).

# Outline of this section

## 3 Validation

- Validation Techniques
  - Testing
  - Types of Tests
  - Code Coverage
  - Automating the Test Suite
  - Mandatory Conventions in our Project

# Validation Techniques

- Use of static analysis tools  
(typing, coding style, absence of overflows, ...)
- Formal proof: costly, rarely used except for critical systems.  
Example: “meteor” subway in Paris, developed with “Atelier B”,  
CompCERT C compiler proved in coq.
- **Code review**: one person reviews the code written by another, and checks the readability and correctness of the code.
- **Test**: run the program with different test-cases, and check that the results correspond to the expected results.

# Outline of this section

## 3 Validation

- Validation Techniques
- **Testing**
- Types of Tests
- Code Coverage
- Automating the Test Suite
- Mandatory Conventions in our Project



# Testing

- Testing is the main validation technique.
- Objective: “show” that the program is correct, or find defects.
- Cannot “prove” the correctness, can indeed only exhibit defects.

# Phases

- Test objective: select the feature to test
- Write test-cases
- Execute tests
- Observing, assessing and recording the result (oracle)
- Fix defects
- Evaluation: was the test sufficient?

# Test objective and test cases

- Test objective: select the feature to test
- Examples:
  - ▶ Test stage A, test stage B (imprecise),
  - ▶ Test passe 2 of stage B,
  - ▶ Test type-checking of declarations,
  - ▶ Test rule 2.9.
- Select relevant data to accomplish the test objective.
- Exhaustive test usually impossible (infinite)

# Execute tests and observe the result

- We execute the program with inputs and get outputs,
- Oracle: Verify that the output matches the expected output,
- Fix defects if some are found.

## Evaluation: was the test sufficient?

- Is the test-suite sufficient? ...
- ... or shall we continue testing?
- How can we “measure” the effectiveness of a test-suite?
- $\Rightarrow$  one answer is the notion of **coverage** (details follow).

# Outline of this section

## 3 Validation

- Validation Techniques
- Testing
- **Types of Tests**
- Code Coverage
- Automating the Test Suite
- Mandatory Conventions in our Project

## Types of Tests: Overview

**Unit tests** Test small parts of the system,

**Integration tests** Check that the components work well together,

**System tests** Test the system under real conditions,

**Acceptance tests** Tests to run before any release  
(useful when the test-suite is not 100% automated),

**Black-box tests** Tests for an objective based on the *specification* of the program,

**Glass-box tests** Tests for an objective based on the *implementation* of the program,

**(Non-)Regression tests** Check that what *used to work still* works.

## Example of program to test: factorial

```
public class Util {  
    public static int fact(int n) {  
        if (n == 0) return 1;  
        else return n * fact(n - 1);  
    }  
}  
  
import java.io.*;  
public class FactMain {  
    public static void main(String[] args) {  
        BufferedReader stdin = new BufferedReader(  
            new InputStreamReader(System.in));  
        System.out.print("Enter a value: ");  
        try {  
            int v = Integer.parseInt(stdin.readLine());  
            System.out.println("fact(v) = " + Util.fact(v));  
        } catch (Exception e) {  
            System.out.println("Input error");  
        }  
    }  
}
```



# Unit tests

## « Tests unitaires »

- Test a small portion of code (one method, one class, ...)
- Example: test for the class `EnvironmentExp`
- Advantages:
  - ▶ Can be executed before building the whole system,
  - ▶ Finds errors more easily than testing the whole system,
  - ▶ Can test conditions hard to reach in normal executions,
  - ▶ Debugging unit-test is easy.
- Drawbacks:
  - ▶ Requires **drivers** to call the code under test (example: class `TestEnvironmentExp`),
  - ▶ May require **stubs** to replace the portions needed by the code under test.

# Unit test for factorial

## The manual way

```
class FactUnit {
    static void assertTrue(boolean c) {
        if (c) {
            System.out.println("ok");
        } else {
            throw new RuntimeException();
        }
    }
    public static void main(String[] args) {
        assertTrue(Util.fact(0) == 1);
        assertTrue(Util.fact(1) == 1);
        assertTrue(Util.fact(3) == 6);
    }
}
```

# Unit test for factorial

Using JUnit (cf. III-[Tests])

```
import static org.junit.Assert.*;
import org.junit.Test;

public class FactTest {
    @Test
    public void testFact() {
        assertEquals(Util.fact(0), 1);
        assertEquals(Util.fact(1), 1);
        assertEquals(Util.fact(3), 6);
    }
}
```

- JUnit provides:

- ▶ A library of assertions (assertTrue, assertFalse, assertEquals...),
- ▶ A launcher that runs all methods decorated with @Test in classes named Test... or ...Test,
- ▶ Integration with Maven (mvn test), IDE (Right-click → Test file with Netbeans)...

# Integration tests

## « Tests d'intégration »

- Test for a set of methods, classes, or packages
- Examples: `test_synt`, `test_context`

## Example system test for FactMain

```
#!/bin/sh
```

```
echo "Enter a value: fact(v) = 24" > expected
```

```
# we test both input/output and computation of fact(4).
```

```
echo 4 | java FactMain > actual
```

```
if ! diff expected actual; then
```

```
# exit with error if expected and actual differ.
```

```
exit 1
```

```
fi
```

```
# we should try "echo -1 | java FactMain" too ...
```

# Black-box Tests

« Tests boîte noire »

- Black-box test = functional tests,
- Based on **specifications** of the program,
- Can, and **should** be written before coding,
- Preferably not written by the implementer of the code under test, otherwise
  - ▶ Ambiguities in the specifications are interpreted the same way,
  - ▶ Missing functionality will hardly be detected.
- Example: from the attribute grammar of Deca, one can
  - ▶ Identify the possible errors,
  - ▶ Write the list of error messages,
  - ▶ Prepare black-box tests for stage B,
  - ▶ Write part of the user manual.
  - ▶ **before writing a single line of code!**

# Glass-box Tests

« Tests boîte transparente (ou blanche) »

- Glass-box test = structural tests
- Based on the **implementation** of the program.
- Goal: cover as much as possible of the program source code.
- Example:  
Dictionary implemented with a hash-table  $\Rightarrow$  test the colliding cases and the non-colliding ones.

# Regression Tests

« Tests de non-regression »

- Re-execute the tests after each modification of the program,
- Check that the new result matches the old ones,
- Example: use “diff old new”



# Outline of this section

## 3 Validation

- Validation Techniques
- Testing
- Types of Tests
- **Code Coverage**
- Automating the Test Suite
- Mandatory Conventions in our Project

# Code Coverage

« Couverture de code »

- Goal: “everything in the code must have been tested”
- What does it mean?
  - ▶ Each instruction has been executed?
  - ▶ Each variable took all the possible values?
  - ▶ ...?
- No perfect coverage metric in a finite world.

# Statement coverage

« Couverture des instructions »

- Definition: a statement is covered when at least one test-case triggers its execution.
- Coverage ratio: number of statements covered/number of statements.
- Goal: cover 100% of the code
- (except dead code, as a result of defensive programming)

# Branch Coverage

« Couverture des arcs »

- Definition: **Branch** = path from an instruction to the next.
- Example:

```
I1;  
if (C1) {  
    I2;  
}  
I3;
```

⇒ Instruction coverage achieved by one execution if C1 is true. Does not cover I1 → I3.

## Jacoco: Statement Coverage Measure

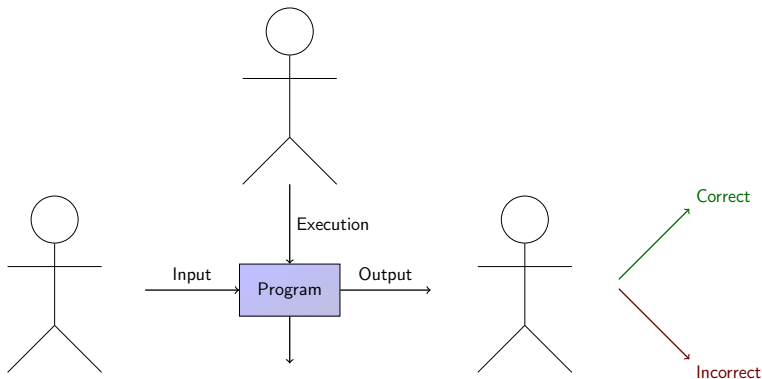
- Compile the program with the right options (see III-[Jacoco]),
- Execute the test-suite,
- Jacoco tells which line of code has been executed, which hasn't.
- $\Rightarrow$  essential to finish the validation  
or some lines of code have not even been **tried**!
- Add extra tests to increase coverage,
- However 100% usually not reachable (dead code, esp. with defensive programming)

# Outline of this section

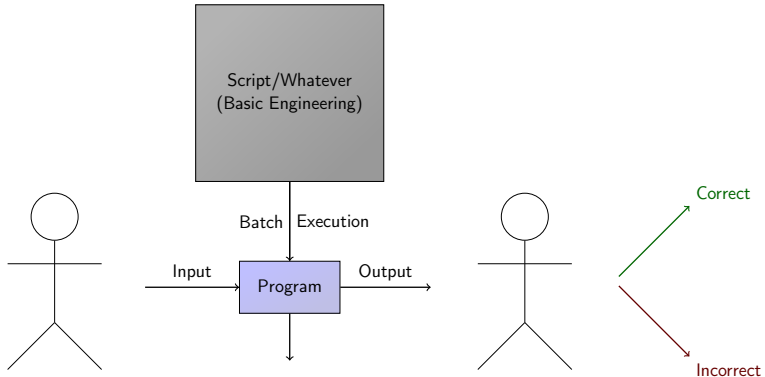
## 3 Validation

- Validation Techniques
- Testing
- Types of Tests
- Code Coverage
- **Automating the Test Suite**
- Mandatory Conventions in our Project

# General Considerations on Testing

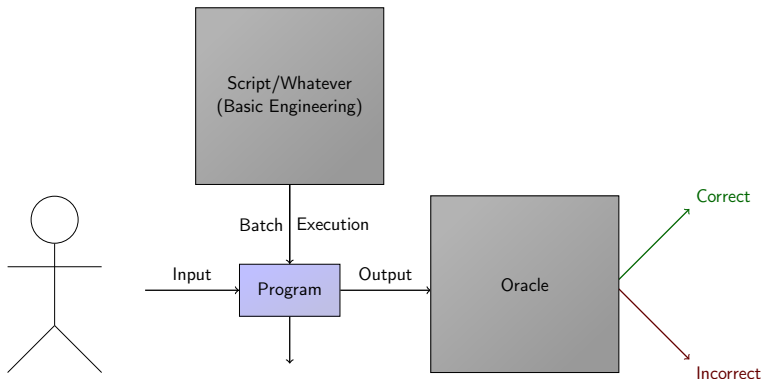


# General Considerations on Testing

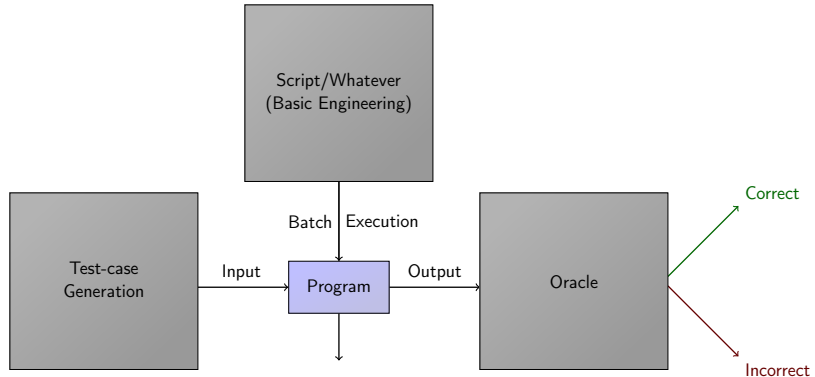




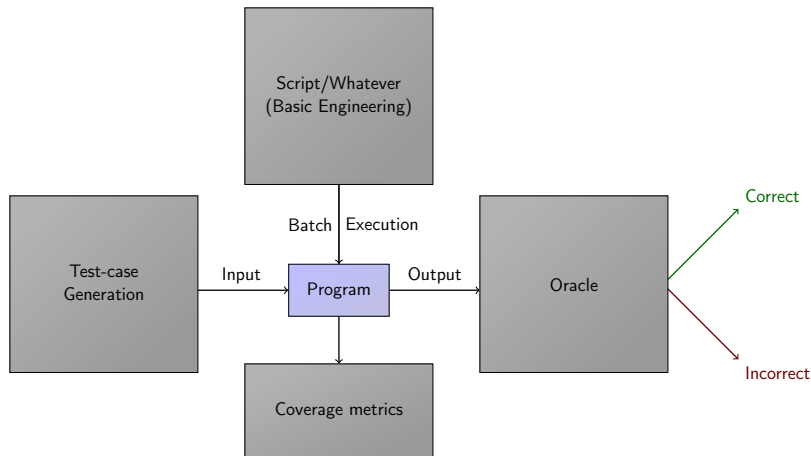
# General Considerations on Testing



# General Considerations on Testing



# General Considerations on Testing



# Execution automation

- Minimal launcher:

```
#!/bin/sh
for i in *.deca
do
    echo "$i"
    # replace <executable> with test_synt or
    # test_lex or test_context or decac
    <executable> "$i"
done
```

# Oracle: Checking the Result

- Automatic oracle essential:
  - ▶ Manual checking of output is boring and error-prone
  - ▶ Regression testing almost impossible without automatic oracle.
- Many ways to manage oracles:
  - ▶ Manual validation the first time, diff the next times,
  - ▶ Comparison of two implementations,
  - ▶ Assertions, defensive programming,
  - ▶ Approximation (example: casting out 9).

## Regression and Efficiency

Regression testing is heavily used in s/w industry, esp. with Continuous Integration (every commit triggers tests to avoid a regression).

BUT automated execution of large test suites takes a lot of CPU and energy.

- Automated regression testing is necessary to ensure **Reliability**
- Requires **smart scripts** to focus on impacted parts.

⇒ You will have to manage conflicting goals:

- Ensure highest **reliability** (primary goal)
- While being conservative about energy (secondary goal)

Your approach to solving this conflict will be reported (and graded) in your report on energy.

# Outline of this section

## 3 Validation

- Validation Techniques
- Testing
- Types of Tests
- Code Coverage
- Automating the Test Suite
- **Mandatory Conventions in our Project**

# Mandatory Conventions for our Project

## ● Directories:

- ▶ Deca tests must be in sub-directories of `src/test/deca/syntax`, `src/test/deca/context` and `src/test/deca/codegen`.
- ▶ Each directory must have
  - ★ `valid/`: Deca program correct with respect to the current stage.
  - ★ `invalid/`: Deca program triggering a compiler error in the current stage.
- ▶ `src/test/deca/codegen` has in addition:
  - ★ `interactive/`: all interactive tests.  
`valid/` and `invalid/` must not contain any read instruction.
  - ★ `perf/`: performance tests, to assess the number of ima cycles used executing them. They should all be valid programs.



# Mandatory Conventions for our Project

cf. III-[Tests]

- File name extensions:
  - ▶ .deca: Deca source files,
  - ▶ .ass: Generated (archived) assembly files.

# Mandatory Conventions for our Project

- Automation: `mvn test`
- Test-suite **must** be automated as much as possible (scripts).
  - ▶ Scripts must exit with 0 if the test succeeds, with another value (exit 1) if the test fails.
  - ▶ Add test scripts in pom.xml file (cf III-[Tests], section 1.6).
  - ▶ Example scripts are provided, but are minimalistic.
- Must be non-interactive by default (both for success and failures)
- (for info: the teacher's test infrastructure is >2000 lines of shell-script).
- `/bin/sh` is the suggested language for test automation (perl, python are other good candidates).
- Resources for shell-scripts available on EnsiWiki.

# Test Suite in the Software Engineering Project

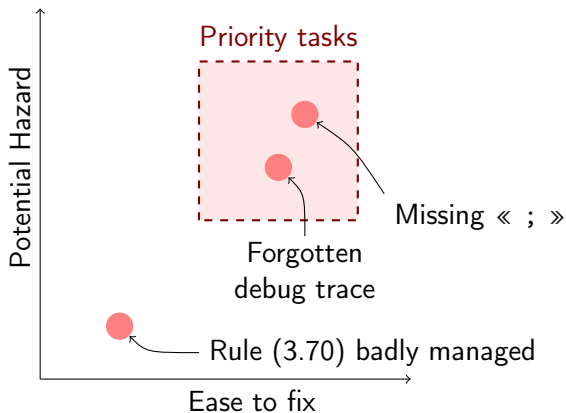
- Test suite is an important part of the grade.
  - ▶ It weighs almost the same as the compiler
  - ▶ If your compiler is perfect, but tests are absent, you get half the points.
  - ▶ And you will get the same if you do not write a single line of code for decac but have excellent tests
  - ▶  $\Rightarrow$  share your efforts accordingly.
  - ▶ You can and should write tests before writing a single line of code.
- Grading takes into account:
  - ▶ Coverage of the test-suite
  - ▶ Test-case layout (conventions above)
  - ▶ Automation

See III- [Tests]

# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

## Risks management: cost-benefit



## Risks management & release process

- Risk assessment and control:

Document	Danger	Action
I- [Introduction]	Miss a deadline	Use an agenda
IV- [Example]	Fail on provided example	Test it!
II- [Decac]	...	...
...		

- Release Process

- ▶ Checklist of actions to perform before a release
- ▶ Should prevent **all** major risks (i.e. compiler unusable with respect to the teachers' testsuite, grossly mis-classified testsuite. . .)

See III- [Tests], section 2.

# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

# Roles

- Designer
- Developer
- Reviewer
- Tester
- Documentation writer
- Scrum master etc.

Tips (see resources on Project Management on Chamilo):

- Ideally, try all the possible roles.
- Practically, in this project, more efficient to choose depending on your individual skills.
- However, a big task should never be assigned to a single person, as this single person can fail (lack of skills, health problem...)
- At any time, the team should be able to re-assign tasks quickly.



# Provisional Schedule

« Planning prévisionnel »

- Make a provisional schedule at the beginning of the project.
- Use “planner Planning.planner”<sup>1</sup> in Projet\_GL/planning to modify your schedule
- Specify your increments and distribute the tasks between the members of the team.
- (You may use an alternative to Gantt charts, e.g. burndown chart, for day-to-day planning)

---

<sup>1</sup>or your favorite alternative, as long as it can create PDF files

# Actual Schedule

« Planning effectif »

- Make the actual schedule of your daily work.
- You must explain the differences between your estimated and effective schedules (evaluation takes into account your explanations, not the differences themselves)
- Use “`planner Realisation.planner`” in `Projet_GL/planning` to modify your schedule.

## Typical Efforts for Projet GL

According to parts of project

- Stage A lex+synt: 10%
- Stage B ctx verif: 20%
- Stage C gencode: 45%
- Extension: 25%

According to type of activity (see Rule of thumb, adapted for Projet GL)

- Analysis & Design: 25% (mostly for stage C and extension)
- Coding & debugging: 20%
- Validation (reviews & tests): 35%
- Documentation & Management (incl. lectures) : 20%

# Activity Report

You are expected to make an activity report for each “progress meeting”. Specify:

- what has been done since the last “progress meeting”,
- the current differences between your effective and estimated schedules.

We advise you to keep a detailed count of how many hours were spent on each task (including tasks such as meetings, preparations etc): this will help you for the final report (“bilan”), and also for your own feedback and planning.

# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques

# Progress Meetings in our Project

## Reminders

- I-[Suivis]
- 3 meetings, 30 minutes each
- 20 minutes “progress report”
  - ▶ **You** convince the teacher that the progress is good,
  - ▶ Must be prepared.
- 10 minutes “technical support”
  - ▶  $\Rightarrow$  The teacher can help you.
- first two meetings with a “SHEME” teacher.

# First Progress Meeting

- See I-[Suivi-SHEME1]
- Prepare a short document presenting your team and your organization,
- Prepare a provisional schedule (See III-[GuidePlanner]).
- Present a proposition of an extension (2 pages)
  - ▶ analysis
  - ▶ draft specification of the extension

# Outline

- 1 Introduction
- 2 Software Lifecycle
- 3 Validation
- 4 Risks management, release management
- 5 Schedule, Roles
- 6 Progress Meetings in our Project
- 7 Programming Techniques



# Outline of this section

## 7 Programming Techniques

- Tracing
- Additional recommendations

## Execution Traces

- Traces can be useful to debug a program
- *must* be easy to remove  
⇒ **never** use `println` for debugging.
- Implementation (the manual way - prefer `log4j`):

```
class TraceDebug {  
    private static final int LEVEL = 5;  
    public static void trace(int level,  
                             String message) {  
        if (level <= LEVEL) {  
            System.out.println("trace: " + message);  
        }  
    }  
}
```

Usage: `TraceDebug.trace(4, "Message");`

# Log4j library

cf. III-[ConventionsCodage]

```
● import org.apache.log4j.Logger;
  public class LogClass {
    // Instantiation of logger, done once for each class.
    private static final Logger LOG =
      Logger.getLogger(LogClass.class);
    // ...
    LOG.trace("Trace Message!");
    LOG.debug("Debug Message!");
    LOG.info("Info Message!");
    LOG.warn("Warn Message!");
    LOG.error("Error Message!");
    LOG.fatal("Fatal Message!");
```

- To choose the level:

- ▶ method `setLevel` of each logger,
- ▶ configuration file `log4j.properties`  
(in `src/test/resources/` and `src/main/resources/`).


- warn level  $\Rightarrow$  messages corresp. to warn, error and fatal displayed

# Outline of this section

## 7 Programming Techniques

- Tracing
- Additional recommendations

## Good Practices and Coding Style

- Keep methods short ( $\approx 1$  screen)
- Do not write long lines (80 characters max)
- Indent consistently with 4 spaces (if using tabs, 1 tab = 8 spaces)  
 Not the default with Eclipse :-)
- Class names start with an uppercase letter, method and variable names start with a lowercase letter
- Comment your code to explain *why* the code is how it is, not *what* it does
- Comment your method headers (javadoc) to explain what methods are doing (pre/post conditions, ...)
- cf. III-[ConventionsCodage], section 3

# Defensive Programming

- See III-[ProgrammationDefensive]
- Method preconditions: conditions that the method arguments must satisfy:
  - ▶ partial functions,
  - ▶ conditions that the arguments must be satisfied, so that the algorithm works correctly.  
Example: dichotomic search in a sorted array.
- Method postconditions: conditions that must be satisfied after a method call.
- Invariant: condition that is always satisfied (loop invariant, class invariant)

# Defensive Programming

- Defensive programming: explicit check of preconditions, postconditions and invariants.
- Allows the programmer to detect and correct bugs at a lower cost.
- When an assertion is violated, the program is stopped by raising an exception.

# Checking Preconditions

- Use of the class `Validate` from apache commons
- Example:

```
import org.apache.commons.lang.Validate;
// ...
/**
 * precondition : x >= 0
 */
float sqrt(float x) {
    Validate.isTrue(x >= 0 ,
                    "x should be positive");
    // ...
}
```

- Methods: `isTrue`, `isFalse`, `notNull`, `notEmpty`.



# Checking postconditions and invariants

- Use of assertions
- Assertions are enabled during development, and disabled during final testing and release.
- In Java: assertions are disabled by default, enable with `java -enableassertions`
- Syntax:  
`assert condition;`
- Violating an assertion raises the `AssertionError` exception (deriving from `Error`).

# Checked and unchecked exceptions

Two types of exceptions in Java:

- unchecked exceptions
  - ▶ derive from `RuntimeException` or `Error`;
  - ▶ are the result of a programming problem (e.g. `NullPointerException`), or other unrecoverable error (e.g. `OutOfMemoryError`);
  - ▶ should not be caught.
  - ▶ `Validate` and `assert` raise unchecked exceptions.
- checked exceptions
  - ▶ are part of the specification of the method (`throws` clause);
  - ▶ must be caught by the caller.