Development Process

Well-known development processes

• RUP (Rational Unified Process)
• CMMI (Capability Maturity Model Integration)
• Agile / XP (eXtreme Programming)

Waterfall Development Process

Civil engineering:
One first draws plans and only then are they realized!
Why wouldn’t it work with software?

Iterative Process (spiral)

Superficial analysis, design, implementation and test of the whole application, then better analysis, design, implementation and test, and improvement at each revolution?
or?
Thorough analysis, design, implementation and test of a part, and then analysis, design, implementation and test of a new part at each revolution?

Agile Development Process

Most important characteristics:

No long term planning. Deliver working software frequently
(from a couple of weeks to a couple of months, with a preference to the shorter timescale).

RUP
RUP: **Rational Unified Process**

- For the project manager
- What is important in a project
- What leads to failures
- Working team composition
- Management tools
- Very little on how to do things!

---

**RUP: Warning**

What is Rational's RUP?

It is a methodology. But to sell it, Rational supplies a bunch of html pages (about 16000 pages). There are some documentation templates also. A warning though, if you are thinking to get into a good methodology in reasonable time, then you need to rethink, RUP will take a looooooooooong time to grasp, implement and finally there is a 50% risk that you will never reap the benefits.

[http://www.theserverside.com](http://www.theserverside.com)

---


Characteristics and best practices:
1. Iterative process
2. Requirement management
3. Architecture and use of components
4. Modeling and the UML
5. Quality of development process and product
6. Change management (tools)

---

**RUP: 1 - Iterations**

---

**RUP: 2 - Requirement management**

- Requirements are stakeholders’ needs, goals and wishes. (stakeholders - *parties prenantes* = contractor, customers, users, employees)
- *(FURPS: functionality, usability, reliability, performance and supportability)*
- *URSP = non functional requirements*

---

**RUP: 3 - Architecture**

- Components:
  - A nontrivial piece of software, a module, a package, a subsystem, a layer, major classes
  - CORBA (remote objects), ActiveX, JavaBeans, a Web page, database table
  - Mechanisms such as persistency and communication tool, patterns (MVC, ORB object request broker . . . )
RUP
4 - UML and modeling

Use case diagram for an ATM
(automate de banque)

1. A client inserts a card into the ATM.
2. The system reads and validates the card information.
3. The system prompts for a PIN.
4. The client enters the PIN.
5. The system validates the PIN.
6. The client selects “Withdraw Money”.
7. The client enters the requested amount.
8. The system requests the amount from the client’s account.
9. The system asks the client to remove the card.
10. The system dispenses the requested amount from the banknote buffer (only done if the requested amount has been granted in 8).

RUP: 4 - Collaboration diagram

RUP: 5 - Quality

- Quality:
  - Reliability (no bug)
  - Functionality (does what it is meant for)
  - Performance
- Tests:
  - Unit tests
  - Integration tests (interactions of units)
  - System test (complete application)
  - Acceptance test (same, but performed by the buyer)
  - Regression tests (repetition of previous tests to assure that the new features have not introduced bugs in the former part)

RUP: 6 - Change management

- ClearQuest
  - Tool for change request management
- ClearCase
  - Tool for configuration management

CMMI
CMM - CMMI

• Capability Maturity Model
  (developed by the Software Engineering Institute of CMU)

• Capability Maturity Model Integration
  (upgrade of CMM)

  •  
  http://ltiwww.epfl.ch/~petitp/GenieLogiciel/CToMFeb02.pdf
  (article by a RUP vendor, excerpts on the next slides)

CMMI

Level 1 (initial)

Represents a process maturity characterized by unpredictable results.
Ad hoc approaches, methods, notations, tools, and reactive management translate into a process dependent predominantly on the skills of the team to succeed.

CMMI

Level 2 (managed)

Represents a process maturity characterized by repeatable project performance.
The organization uses foundation disciplines for requirements management; project planning; project monitoring and control; supplier agreement management; product and process quality assurance; configuration management and measurement/analysis.

CMMI

Level 3 (defined)

Represents a process maturity characterized by improving project performance within an organization.
Consistent, cross-project disciplines for Level 2 kpas (key process areas) are emphasized to establish organization-level activities and practices.

Additional organizational process areas include:
  - Requirements development, technical solution, product integration, verification, validation, risk management, organizational training, organizational process focus, decision analysis and resolution, organizational process definition, integrated project management

CMMI

Level 4 (quantitatively managed)

Represents a process maturity characterized by improving organizational performance.

Historical results for Level 3 projects can be exploited to make trade offs, with predictable results, among competing dimensions of business performance (cost, quality, timeliness).

Additional Level 4 process areas include:
  - Organizational process performance: setting norms and benchmarks for process performance.
  - Quantitative project management: executing projects based on statistical quality-control methods.

What are risks ?

If your customers need a new system by a specific date the risk is high. If that system is a new challenge for your software group the risk is even greater. If that system is a new challenge to the entire software industry the risk is still greater even.

Unsatisfied employees also participate in the risk.
CMMI

Level 5 (optimized)

Represents a process maturity characterized by rapidly reconfigurable organizational performance as well as quantitative, continuous process improvement.

Additional Level 5 process areas include:

– Causal analysis and resolution: proactive fault avoidance and best practice reinforcement.
– Organizational innovation and deployment: establishing a learning organization that organically adapts and improves.

(few companies arrive at that level)

Agile Development Processes

XProgramming

Agile Development Process

http://agilemanifesto.org
http://martinfowler.com
http://www.ambysoft.com/

Most important characteristics:

No long term planning. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

XProgramming

Complaint and excuse of the programmer:

"the problem with this project is that the requirements are always changing"

Actually it is always and will always be the case, we must thus do with it!

→ XProgramming

(http://www.extremeprogramming.org)

XProgramming

• No global up front planning
• Slices (meaningful slices) defined with the customer
• Created quickly (2-6 weeks)
• Accepted by the customer

→ Early feedback and impact on the requirements

• Pair programming !!
XProgramming

Planning
- User stories are written.
- Release planning creates the schedule.
- Make frequent small releases.
- The project velocity is measured.
- The project is divided into iterations.
- Iteration planning starts each iteration.
- Move people around.
- A stand-up meeting starts each day.
- Fix XP when it breaks.

XProgramming

Designing
- Simplicity
- Choose a system metaphor.
- Use CRC\(^1\) cards for design sessions.
- Create spike solutions (for difficult problems) to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

\(^1\) Class-Responsibility-Collaboration

XProgramming

Coding
- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time (CVS).
- Integrate often.
- Use collective code ownership.
- Leave optimization till last.
- No overtime.

XProgramming

Testing
- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

XProgramming

- Documentation is a planned task and the customer pays for it (some documentation must be integrated in the source code)
- Teams are from 2 to 12 persons
- A CVS (concurrent versioning system) is of course mandatory

XProgramming

Collective Code Ownership

http://www.extremeprogramming.org
TDD: Test Driven Development

- Write a test before any real code writing

On Eclipse
- JUnit 1.3.8: Java 1.4
- JUnit 4: Java 5.0 with annotations

TDD: Test Driven Development

Documentation Eclipse:

- Help > Help Contents > Java Development User Guide > Getting Started > Basic Tutorial > Writing and running JUnit tests (1.3.8)
- Help > Help Contents > Java Development User Guide > What’s new > JUnit Toolings (4)
- (http://www.junit.org) → JUnit 4.0 in 10 minutes

TDD

- Create a Java project as usual
- Project > Build Path > Add Libraries… > JUnit 4
- Right-click the class that will be under test (possibly with an empty method) > New > JUnit Test Case (creates a case of the version entered into the build path)
- Define a package that is going to contain the tests

TDD

Edit the test class:
- Create an instantiation of the Class under test in the @BeforeClass method (once for all methods) or in the @Before method (once for every method)
- Put one test (CTRL-space) per test method. If you put several tests, they will not be identified separately when there is an error.
- The message in an assertion is displayed if there is an error
- It is possible to check the kind of Exception returned by the method under test (see doc)

```java
tdd

tdd

package unitTests;
import static org.junit.*;
public class ComputationTest1 {
    static example.Computation computation = null;

    /**
     * This method is executed only once before all tests
     * Here, it creates an environment to test the modules
     */
    @BeforeClass
    public static void setUp() throws Exception {
        computation = new example.Computation();
        computation.setResult(0);
    }
```
TDD
package unitTests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.BeforeClass;
public class ComputationTest1 {
    static example.Computation computation = null;
    /**
     * The following method is repeated before every test
     */
    @Before
    public void setUpRepeated() throws Exception {
        computation = new example.Computation();
        computation.setResult(0);
    }
    @After
    public void tearDown() throws Exception {
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 1 of computation", computation.getResult()==6);
    }
    @Test
    public void testComputation() {
    }
    @Test
    public void testAdd1() {
        computation.add(7);
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 2 of computation", computation.getResult()==-5);
    }
    @Test
    public void testAdd2() {
        computation.add(-12);
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 2 of computation", computation.getResult()==-5);
    }
}

TDD: running the tests
- Select some test classes or the test package
- Right click the selection > Run As > JUnit Test
- The result is displayed in the view JUnit (same window as the Package Explorer)

Exercice
Develop a class with a method that counts the lines that are not empty
nor just contain comments, creating tests for each new code increment.
As soon as you have a idea to start the problem, create a test, implement
this first part and test it. For each new increment, do the same.
Don’t think too much beforehand, just try and test, but keep all tests
and improve/refactor your code
Use pair programming

```
package unitTests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.BeforeClass;

public class ComputationTest1 {
    static example.Computation computation = null;

    /**
     * The following method is repeated before every test
     */
    @Before
    public void setUpRepeated() throws Exception {
        computation = new example.Computation();
        computation.setResult(0);
    }

    @After
    public void tearDown() throws Exception {
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 1 of computation", computation.getResult()==6);
    }

    @Test
    public void testComputation() {
    }

    @Test
    public void testAdd1() {
        computation.add(7);
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 2 of computation", computation.getResult()==-5);
    }

    @Test
    public void testAdd2() {
        computation.add(-12);
        // To show an error in a test, the next check is incorrect !
        assertTrue("Error 2 of computation", computation.getResult()==-5);
    }
```