Ingegneria del software II – 01BIF
A.A 2009-2010

Introduction

Heating Control System is a system able to control the heating system in your house.
The system loads a set of user-defined global settings and starts operation immediately. If the settings file cannot be found, default settings are automatically used.
You can modify your house settings by a graphical interface.
The control is realized by the interaction among several classes; a main window displays the conditions of each room. Once one or more rooms are selected, a dialog box lets you choose the ideal conditions for them.
Heating Control System can interact with the physical environment: every interaction with the reality is simulated calling a method belonging to the *IPhysicalEnvironment*, *IPhysicalBoiler* and *IPhysicalRoom* interfaces. You have to implement these interfaces to provide a simulation of a real physical environment. For further information see *SystemDocumentation.doc* and *Appendix1.doc*.

To install a running version of the Heating Control System only the file *hcs.jar* is needed. The package includes the binary classes that constitute the control system and the source code for the *IPhysicalEnvironment*, *IPhysicalBoiler* and *IPhysicalRoom* interfaces, for the *PhysicalFactory* class and for the *CRoom*, *Env* and *PhysBoiler* example implementations. In addition, it contains the file etc/settings.xml, containing the description of the house.
It goes without saying that the interfaces are provided just for convenience, and should never be modified.

Projects – Instructions

The project is divided in a certain number of phases.
Documents delivery and code is done by means of svn . Further informations could be published on site web course.
Deliverables are orange colored.

Phase 0 : Software setup and project checkout

Read carefully APPENDIX A, and follow instructions you find there.

Fase 1 – Project management

Input:

- example of 2 project plans, 2 templates

Goals:

- estimation, project planning and control

Output:

- D1 Project Plan : Project Plan document, time and cost estimation

Tools used:

- Word, Excel, Msproject, Openproject....

Guidelines.
Project planning is a continuous activity, that supports the other ones.
You have to plan the work creating tasks and giving them to different group components.

D1 Project Plan:
document containing project plan. Document can contain:

- Gantt of the project
- Effort estimation. Produce this estimation starting from requirements specification using Delfi and Cocomo1 techniques .

Document must be presented modifying the provided template. It must contain:

- deliverable list
- roles and responsibilities (project manager, configuration manager, quality manager,..)
- configuration management policies

- measures: which measures you're collecting

Phase 2 – Acceptance tests, black box

Input:

- requirements
- executable, instructions, installation

Goals:

- Verify that executable is coherent with requirements specification
- Verify requirements specification (Ambiguities, omissions)
- Validate executable (defects)

Output:

- D2 test report (using maven reports)
- D3 defect list (you must identify defects using convention described in Appendix B,filling defects.xml and respecting schema defect_schema.xsd; an example is provided: defects_list_example.xml)
- D4 physical system stubs (you must put your stub code in src/main/java/mystub/)

Tools:

Eclipse, JUnit, JFCUnit, Maven, Cobertura

Guidelines.
Heating control system is a system composed by software and device (sensors, actuators). The test required, for practical reasons, don't use physical sensors and actuators. Each group must realize an external environment model with which the system interacts.
The system interact with external world by means of a GUI (commands and parameters given by the user) and by means of interfaces in package Stub (interfaces with sensors and actuators).
The model of external world can have different levels of precision. Each group must decide to which level stop, with respect to available time.
The technique used is black box (equivalence partitioning and boundary conditions).
The starting point is the requirements specification.
Defects have to be intended in a wide meaning. They can relate to the executable, on documentation, on process, on tools.
While defects of documents must be corrected during the group project, defect on tools and processes (process improvement) will be corrected likely in the next process execution (next course).

Please notice that the deliverable you produce (stubs code, test cases, ….) is defect-prone too.
Tests and reports must be done with the usage of maven.

Test covering. We suggest to cover each requirement and scenario (that means write at least a test case for them), respecting the standard notation in Appendix B.

Evaluation. The evaluation is done first of all on number of found defects on the executable (real defects...), then on number of written tests and on the precision of stubs. The correction is based on automatic reports built up by maven.

Phase 3 – Unit and integration test

!(important)**PreCondition**:
Edit file pom.xml by commenting the following rows (no need anymore to use library):

```
<dependency>
      <groupId>softeng</groupId>
      <artifactId>hs_nostub</artifactId>
      <version>1.0</version>
</dependency>
```

Insert source code you receive in src folder.

Input:

- requirements specification
- source code

Goals:

- Verify and validate source code
- Verify requirements specification (ambiguities, omissions)
- Verify source code is coherent with requirements specification

Output:

- D2a test report (test cases list, results) (it extends D3)
- D3a defect list (you must identify defects using convention described in Appendix A, continuing to fill defects.xml and respecting schema defect_schema.xsd; an example is provided: defects_list_example.xml)
- D5 integration strategy followed : integration sequence of the classes
- D4a stubs and drivers (it extends D4)
- D6 – Mutation testing output

Tools:

Eclipse, JUnit, Cobertura, Maven, Muclipse

Guidelines.
This phase is similar to the previous one, but the focus now is on source code. You can use white box and inspection techniques.
The goal is to execute white box for each class (instruction and branch coverage for each class). In theory you should execute unit test for each class, then the integration test. In the practice, considering dependence graph of the classes, unit test is immediate only for leaves classes. For those classes that aren't leaves, unit test means to write stubs, and the this is linked to the problems related to integration test. So we suggest to plan together unit test and integration test, in such a way you could minimize number of stubs.
One of the problems is to know dependence graph, starting from the provided source code (please notice that the document that specifies requirements contains the dependency graph, but this graph could be not coherent with the one effectively implemented in source code). This problem (reverse documentation) can be resolved by hand or by means of a tool (examples: Omondo, or eclipse plugins - http://www.eclipseplugincentral.com/)
Please notice that the deliverable you produce (stubs code, test cases, ….) is defect-prone too.
Tests and reports must be done with the usage of maven.

Coverage. We suggest to cover instructions and code of all methods of all classes. You can check your coverage by using maven-cobertura reports. Achieve this goal could be difficult, so you could also decide alternative strategies: select classes (o packages) with some criteria and obtain the 100% coverage only there

Mutation test. You evaluate the completeness of your tests with a further technique, the mutati testing. Follow instructions at http://muclipse.sourceforge.net/index.php (section Using MuClipse) to read how to use MuClipse. Stats of MuClipse must be copied in *docs/mutations_results* .

Evaluation. Evaluation is done first of all on number of defects found, then on number of written tests and on coverage.

Phase 4 – Defects correction and regression testing

Input:

- reviewed requirements specification
- source code
- previous deliverables

Goals:

- correct defects found in previous phases
- Regression testing

Output:

- D6 Source code modified

- D7 New design

Tools:

Eclipse, eclipse plugins ...

Guide lines.
In this phase you correct defects found in precedence. Since you can find defects not only in source code but also in other documents, this phase receives as input a corrected requirements specification.
Defect correction could implicates changes on source code and on system design.
Defect correction could cause the creation of new defects. In order to avoid or minimize this problem, it's important to execute regression testing, that means successfully pass test of D1 and likely test of D1a (but please notice that not all tests could be applied in case of changes on design and on classes interfaces, so it is suggested to minimize changes to design).

Fase 5 – post mortem

Input:

- all input/output documents of the previous phases

Goals:

- project balance sheet

Output:

- D8 Post mortem: measures and analysis

Tools:

MS Project, Excel, ...

Guidelines.

D8 post mortem.
This document contains product and process measures and their analysis, main positive or negative experiences to repeat or avoid (lessons learnt).

APPENDIX A - Software setup and initial checkout

Java

Install sun jdk java 6 . If you have previous jdks, be sure environment variable JAVA_HOME effectively points to JAVA6.

Maven
Download maven2 from http://maven.apache.org/download.html and install it.

Eclipse

1. Download Eclipse Ganymede IDE for Java Developers from
http://www.eclipse.org/ganymede/ .

2. Start Eclipse, and install "subclipse" plugin following the order of these steps (it's important to follow this order, otherwise problems with plugin dependencies will occur):

Svn kit plugin
help -> Software Updates

then select tab Availbale Software, and click on the right Add Site. In the new window, type

http://subclipse.tigris.org/update_1.4.x

Check box http://eclipse.svnkit.com/1.2.x/ and, then click on the right Install. Accept general conditions and at the end restart Eclipse.

Subclipse
help -> Software Updates
then select tab Availbale Software, and expand the box http://subclipse.tigris.org/update_1.4.x and check the following one:

- JavaHL Adapter
- Subclipse
- SVNKit Adapter

Then click on the right Install. Accept general conditions and at the end restart Eclipse.

Maven
help -> Software Updates

then select tab Availbale Software, and click on the right Add Site. In the new window, type

http://m2eclipse.sonatype.org/update/

expand the box http://m2eclipse.sonatype.org/update/ and check the following one:

- Maven Integration
- from Maven Optional Components
  1. Maven central repository index
  2. Maven SCM handler for Subclipse
  3. Maven SCM handler for Team/CVS
  4. Maven SCM Integration

Then click on the right Install. Accept general conditions and at the end restart Eclipse.

Muclipse
Follow instructions in http://muclipse.sourceforge.net/updatesite.php

Initial checkout of the project

Start Eclipse and select:

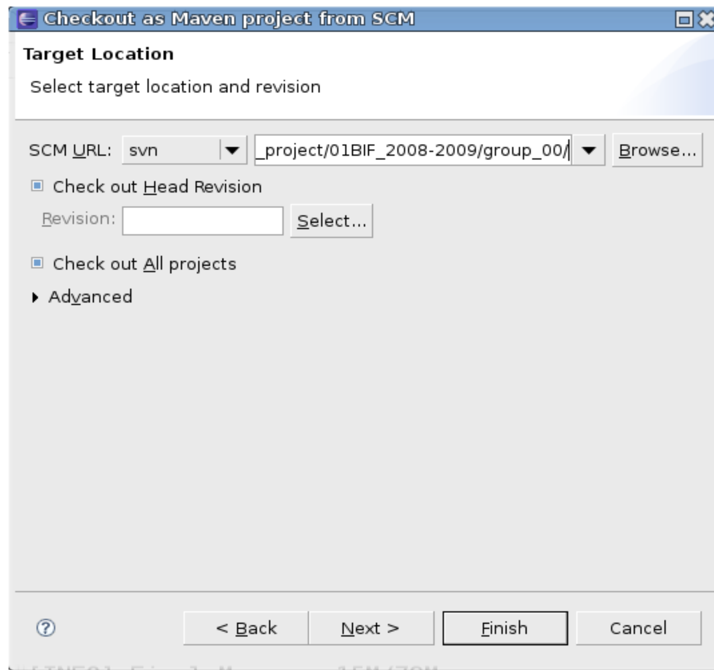
File-> New -> Project...

Then select the tye of project:

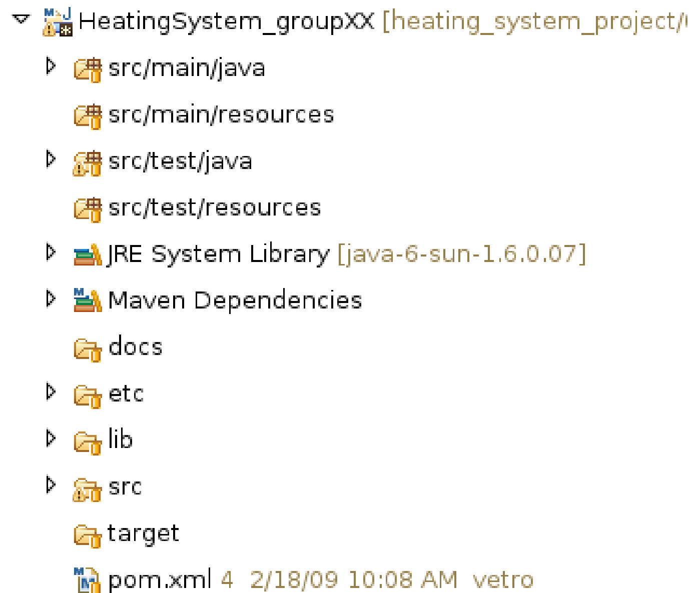
MAVEN -> Checkout Maven projects from SCM

In the new window, select:
SCM URL → svn
http://softeng.polito.it/repoStudents/heating_system_project/01BIF_2008-2009/group_XX/

where XX is your group number. Then click finish. Insert username and password when the window will ask you.



At this point you have checked out your project. You should have something similar to this:



Now you have to add the jar of Heating System Software (the one that you find in /lib folder) into you maven local repository.
Open terminal and type:

mvn install:install-file -Dfile=<path-to-hcs_jar> -DgroupId=softeng -DartifactId=hs_nostub -Dversion=1.0 -Dpackaging=jar

inserting the path to hs.jar where you read <path-to-hcs_jar>

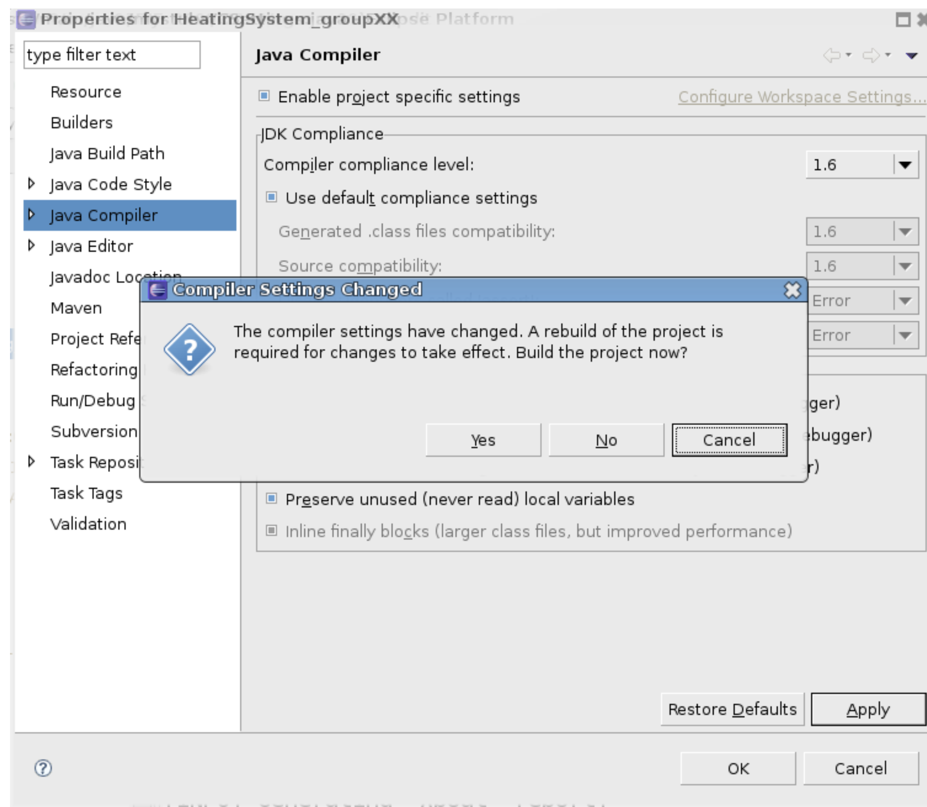
Finally, you have to go in you maven repository directory (usually, it's in YOUR_HOME_FOLDER/.m2) and put file hs_nostub-1.0.pom inside folder <path_to_your_maven_repository>/softeng/hs_nostub/1.0/

PAY ATTENTION: if you would like to use others libraries for your tests, you have to:

- put them in lib folder
- add them to your local maven repository in the same way you installed the jar

It is possible that when you checkout the project some compilation errors appear. This will happen if Eclipse sets automatically compiler compliance level to 1.4 . To delete the error, follow these simple steps:

- go on project main folder, click right on it and select properties
- go to Java Compiler, set compiler compliance to 1.6 then click OK and YES:



The structure of the project is the following one

- src : source code folder
  1. main :
     - java : source code of the Heating System Software

        - mystubs : the package where you create/edit stub classes
    - resources: application resources
2. test : test source code folder
    - java : source code of the Heating System Software: please notice that name of classes should follow a specific convention. The following names are valid:

      **/*Test.java
      **/Test*.java

      **/*TestCase.java

      But the following ones are not valid:

      **/Abstract*Test.java

      **/Abstract*TestCase.java

        - blackboxtest : the package where you create/edit black box tests source code(examples are provided)
        - whiteboxtest : package where you create white box tests source code
        - common: package in which you could put classes that you use both in black box and white box tests
    - resources: test resources

3. config : it can contain configuration files
4. docs : this folder contains System documentation
    - defects:
        - defect_schema.xsd : xml schema to respect
        - defects_list_example.xml : example xml defect file
        - defects.xml : the file you have to fill with defects found
    - mutations_results: if you run mutation testing, the output must go here
    - other_documents_produced: any other document you produce (if any) other than defects, project management and mutations results, must go here.
    - project_management: project managements examples and template to fill
    - tools: documentation and references of tools used (subversion, maven)
5. lib : this folder contains the heating system jar that we provide you and that you have to install to your maven repository, and the jar you should use if you will do mutation testing; in this folder you will put all the libs that you would like to use in order to write you tests
6. etc : this folder contains Heating System configuration example and schema.
7. target: folder that will contain maven outputs (bin, reports)
8. pom.xml : your maven project pom

Edit pom.xml inserting group_xx among following tags (where xx is your group number)

<artifactId><!-- put here group_xx --></artifactId>

<name><!-- put here group_xx --></name>

Edit <developers> section inserting your data

Clicking right on any resource(file, or folder), you find button Team : from it you will access all SVN commands, e you use them in order ti keep your working copy aligned with central repository .
Clicking right on pom.xml , than Run as , you have typical maven commands.

PAY ATTENTION:
don't change your project structure otherwise your tests won't be executed on server side

APPENDIX B- Defects categorization

- **Location** : Src code | Document | Design | Process
  - **SrcCode Location**: package.class, package.class.method, Line (text), Line(number)
  - **Document Location**: document_name, page, Line (text), Line(number)
  - **Design Location** : package | package.class | package.class.method
  - **Process Location** : phase (text), phase(number)
- **Problem Description** : text
- **Requirement** : specify which requirement is not respected. If no requirement,put a "-"
- **Status**: open | assigned | close
- **Type**:
  - Src code:
    - **II:** Interface Implementation.
      Something that I cannot influence does not work as it should.
      This defect should never be used when I am the source of the defect. (In principle, this is a special case of ID.)
    - **IU:** Interface Use.
      An interface was used wrongly, i.e., in such a way as to violate the interface specification.
    - **IV:** Data Invariant.
      A special case of IU. The interface violation is: not maintaining the invariant of some variable or data structure.
      Violating the meaning of a simple variable is a special case of this.
    - **MI:** Missing Implementation (of planned functionality).
      A certain part of a design was not implemented.
      If the part is small, MC, MA or WA may be more appropriate.
    - **ME:** Missed Error handling.
      An error case was not handled in the program (or not handled properly).
    - **MA:** Missing Assignment.
      A single variable was not initialized or updated.
      Only one statement needs to be added.
    - **MC:** Missing Call.
      A single method call is missing.
      Only one statement needs to be added.
    - **WA:** Wrong Algorithm.
      The entire logic in a method is wrong and cannot provide the desired

functionality.
More than one statement needs to be added or changed.

    - **WE:** Wrong Expression.
      An expression (in an assignment or method call) computes the wrong value.
      Only one expression needs to be changed.
    - **WC:** Wrong Condition.
      Special case of WE. A boolean expression was wrong.
      Only one expression needs to be changed.
    - **WN:** Wrong Name.
      Special case of WE. Objects or their names were confused. The wrong method, attribute, or variable was used.
      Only one name needs to be changed.
    - **WT:** Wrong Type.
      Two `similar' types were confused.
  - **Document:**
    - **I :** Incompleteness (Informations about an object are missing. For example a class has not been described in the class diagram but is used in a scenario)
    - **W :** Wrong information (A given information about an object is clearly wrong. For example position of the window is in class Door)
    - **R :** Over specification and useless information (A given information is redundant, so that is not useful. An example in the class diagram in that of temperature. This is not important referring to expected functionalities.)
    - **C :** Inconsistence or contradictions (An information to an object is inconsistent or contradicts another information on the object.)
    - **O :** Forward references and other errors
  - Design :
    - **IC:** Interface Capability.
      The design of an interface is wrong, so that the interface does not provide the functionality that it must provide.
    - **IS:** Interface Specification.
      The specification of an interface is wrong, so that the parameters involved cannot transfer all of the information required for providing the intended functionality.
      This is a less fundamental variant of IC: Only parameters need be added.
    - **ID:** Interface Description.
      The non-formal part of the description of an interface is incomplete, wrong, or misleading. This is typically diagnosed after an IU.
      Note that the description of a variable or class attribute or data structure invariant is also an (internal) interface.

        - **MD:** Missing Design (of required functionality).
          A certain requirement is covered nowhere in the design.
          This is stronger than IC, where the coverage is present, but incomplete.
    - Process: -
- **Gravity** : minor | normal | major
- **Test :** test that discover this defect. If no test, put a “-”

APPENDIX C - maven basic commands

Create an empty project:	mvn archetype:create -DgroupId=<groupid> -DartifactId=<artifactid> -DpackageName=<name>
Install a library on local repo:	mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id> -DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging>
Compile	mvn compile
Test	mvn test
Generate reports	mvn site
Clean up	mvn clean