Report for lab 1, TDDD04

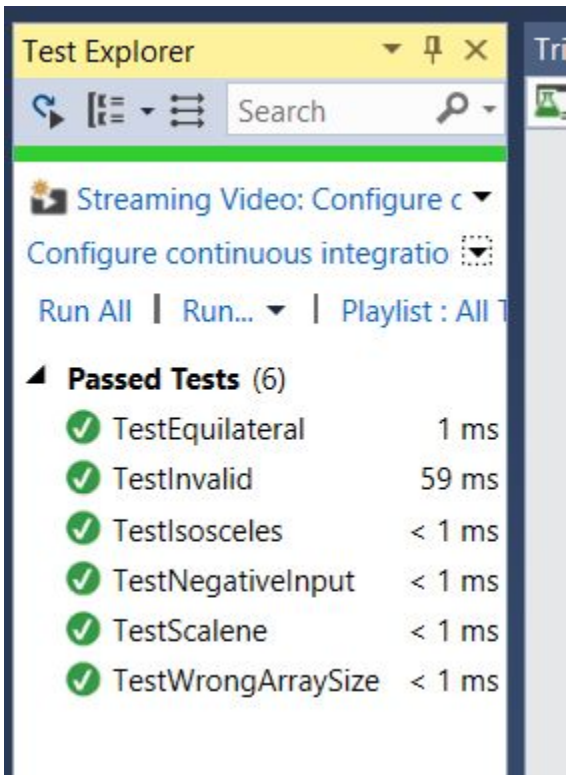
All code needed for this lab is available on gitlab, group tddd04-2016-c3-1.
https://gitlab.ida.liu.se/groups/tddd04-2016-c3-1

Part 1

The test cases that we used to test the program was

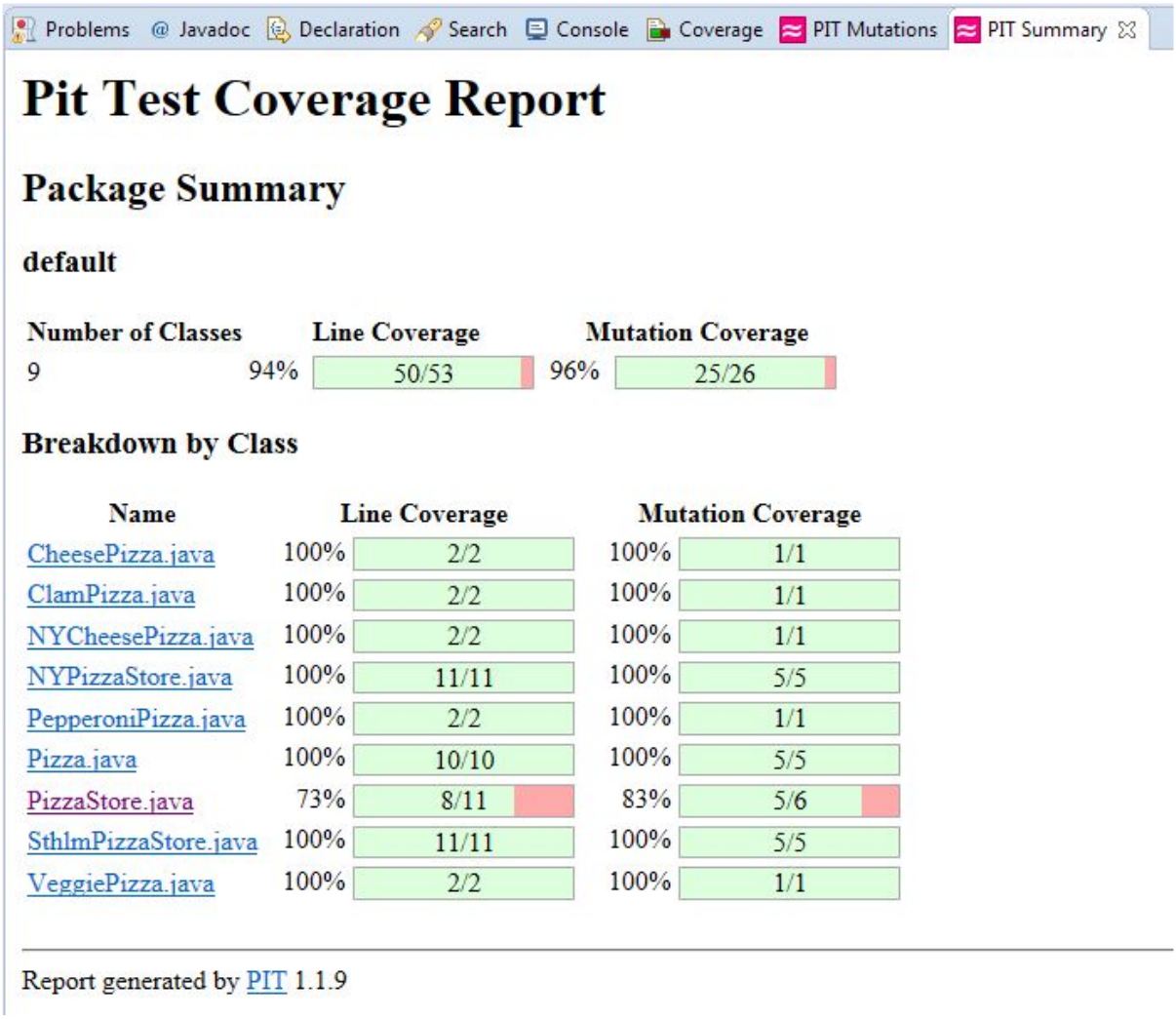
- 2 2 1 (isosceles)
- 4 5 6 (scalene)
- 1 1 1 (equilateral)
- 1 1 2 (non-triangle)
- 1 3 (non-triangle)
- -1 1 1 (negative length of one side)

These inputs were converted to unit tests. In the first three of these cases the program behaved as expected, but in the last three the program crashed.
The code was later fixed to handle these types of input and all the test cases from before now passes, as seen in the figure below.



Part 2

In the factory method code example we managed to get close to 100% line and mutation coverage as shown in the screenshot below. The only missed mutations are in the class PizzaStore.



For the Visitor example we were only able to reach 60 % mutation coverage. When inspecting the missed mutations we could see that they all were in unreachable parts of the program. For example, mutations in the two classes ExpressionTest and VisitorTest were missed since these classes are never used. We also missed mutations in the accept method in the class AbstractExpression. This method will never be executed since all subclasses of this abstract class will override this method.

Pit Test Coverage Report

Package Summary

code_examples

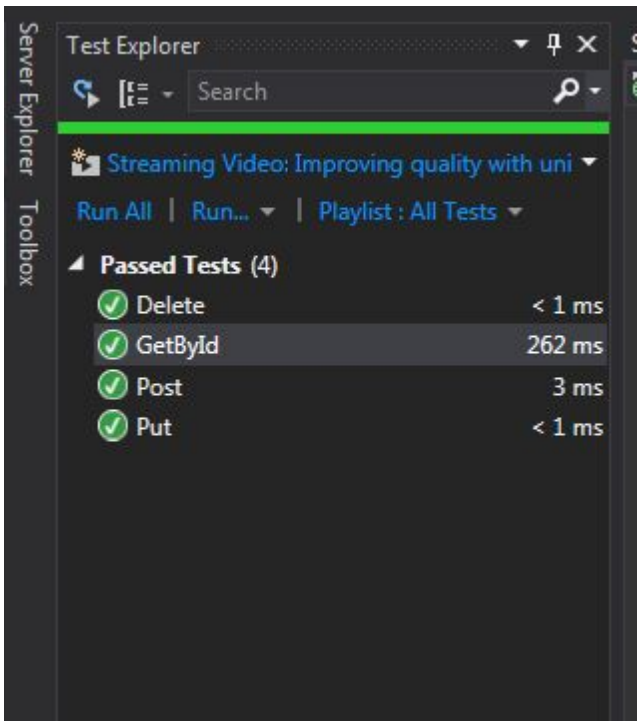
Number of Classes	Line Coverage		Mutation Coverage	
9	76%	56/74	60%	18/30

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CompoundExpression.java	60%	6/10	50%	3/6
CountingVariablesVisitor.java	100%	10/10	75%	3/4
EvaluationVisitor.java	100%	7/7	100%	1/1
ExpressionTest.java	0%	0/4	0%	0/1
Minus.java	100%	8/8	50%	2/4
Number.java	100%	7/7	67%	2/3
Sum.java	100%	9/9	75%	3/4
Variable.java	100%	9/9	100%	4/4
VisitorTest.java	0%	0/10	0%	0/3

Report generated by PIT 1.1.9

Part 3



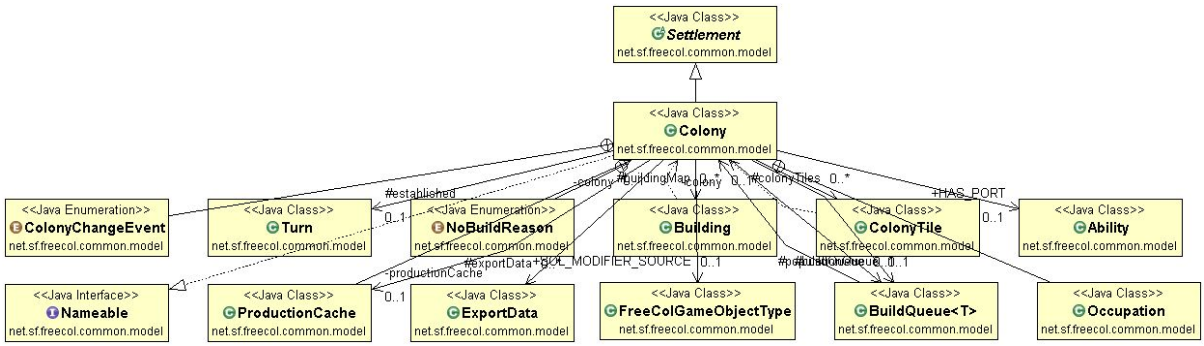
Above we have included the succeeded test cases, all stubs are commented in the code for the lab attached, after implementing the actual storage.

Part 4

All statements in the tests are not covered. The test cases that are not fully covered contains a lot of if statements (branches), some of which are missed when running the tests. This is likely due to the fact that the program i highly coupled which means that it is hard to test small parts of the program (proper unit testing). When the classes are dependent on each other to work, it is hard to test them separately. Therefore we have to use long test cases with many if statements to test as much of the code as possible, which is also why we miss some branches of the test cases.

From the UML class diagram below it is clear that the program is highly coupled due to the many dependencies in both directions from and to the Colony Class to other classes. To enable more thorough testing without creating other concrete objects one could implement mock objects. This allows the tester to decide the behaviour of moc objects to create different scenarios and by so covering all branches of possible executions.

The danger of testing high coupled programmes is the high factor of branching possible; when one class behaviour changes, it creates many different scenarios in other classes as well. This creates the illusion of coverage when in fact lots of branches may have been missed in the tests. Same goes for path coverage.


<<Java Class>>
Settlement
net.sf.freecol.common.model
<<Java Class>>
Colony
net.sf.freecol.common.model
<<Java Enumeration>>
ColonyChangeEvent
net.sf.freecol.common.model
<<Java Class>>
Turn
net.sf.freecol.common.model
<<Java Enumeration>>
NoBuildReason
net.sf.freecol.common.model
<<Java Class>>
Building
net.sf.freecol.common.model
<<Java Class>>
ColonyTile
net.sf.freecol.common.model
<<Java Class>>
Ability
net.sf.freecol.common.model
<<Java Interface>>
Nameable
net.sf.freecol.common.model
<<Java Class>>
ProductionCache
net.sf.freecol.common.model
<<Java Class>>
ExportData
net.sf.freecol.common.model
<<Java Class>>
FreeColGameObjectType
net.sf.freecol.common.model
<<Java Class>>
BuildQueue<T>
net.sf.freecol.common.model
<<Java Class>>
Occupation
net.sf.freecol.common.model
#established
0..1
-colony
#buildingMap
0..*
-colony
0..*
#colonyTiles
0..*
+HAS_PORT
0..1
#exportData
+GOL_MODIFIER_SOURCE
0..1
-productionCache
0..1