SAARLAND UNIVERSITY
COMPUTER SCIENCE

Universität des Saarlandes
Software Engineering Chair
Prof. Andreas Zeller
Gordon Fraser
David Schuler

2. Project: Mutation Testing

The task of this project is to (a) write a mutation testing tool, and (b) to enhance tests to detect mutations.

1 Get the project

You can check out the project from our SVN repository. Your user name is your matriculation number, and the password was sent to you by e-mail. The location of your repository for the second project is:
`https://prog2.cs.uni-saarland.de/debugging/students/project2/<MATR>`
At the end of the project (27th May, at 23:59) the version in the repository will be graded.

2 Overview

The mutation testing tool works in 3 phases:

1. First, the mutations are applied to the source code.
2. Then, the transformed code is compiled and the tests are run for each mutation. At the end of the run the test results for each mutation are written to disk.
3. Finally, the data from the previous step is read and analyzed.

The following sections give an overview over the project. For more details see also the Javadoc comments, and the lecture on *Mutation Testing*.

2.1 Transforming files

MutationTransformer receives a list of Java source files that are mutated using the following mutation operators (defined in enum *MutationType*):

Constant Replacement Every literal in the source code, which is not 0, is replaced with corresponding 0.

Relational Operator Replacement All occurrences of an operator are replaced with another operator. Table 1 shows all operators that are mutated and their replacement operator.

In the transformed files all possible mutations are applied at once using *mutant schemata*. This means that every mutated statement is guarded by a condition, such that at runtime either the original or the mutated statement can be executed. For example, the condition `a > b` could become `mutation1Activated ? a < b : a > b`.

Table 1: Mutations for operators

Operator	Replacement
$>$	$<$
$\geq$	$\leq$
$<$	$>$
$\leq$	$\geq$
$==$	$!=$
$!=$	$==$

After all files are transformed, information about the applied mutations (*MutationInfo*) is stored, such that it can later be read by *MutationInfoReader*. Each mutation is identified by the compilation unit that it is applied to, the type of the mutation, the line it is applied in, and a unique id.

3 Executing the mutants

MutationRunner runs the JUnit tests that are passed as arguments on all mutations applied in the previous step. Therefore, a mutation is first activated. Then all tests are run. After all tests are run it is recorded whether the mutation was detected (killed) or not. At the end of the run the results are stored, and the mutation is deactivated.

Before activating the first mutation the tests are run with no mutation activated. If one of the test cases fails, *MutationRunner* stops with a failure message that includes the names of the failing tests. This also implies that none of the mutations gets a result, which means that the *getMutationResult()* method returns `null`.

The command `ant check-mutation` runs integration tests for the mutation testing tool. Before the first invocation *mvn assembly:assembly* has to be run once. The results of the integration tests are summarized in the files `check-mutation-failures.txt` and `check-mutation-passing.txt` in the `target` directory.

4 Detecting mutations

With the mutation testing tool the quality of tests can be analyzed. To this end, analyze your tests for the *org.apache.commons.math.stat.descriptive.moment.Variance* class from the previous project, and if necessary enhance them in such a way that all detectable mutations in the class are detected. The test suite *AllTests* in the *testing.debugging.project2.mutation.math.tests* package should should include the enhanced tests. Note that we do not expect the mutation in line 69, where the `serialVersionUID` is set, to be detectable.

Enjoy!