Mutation Testing

Stuart Anderson

Overview

Mutation testing is a structural testing method, i.e. we use the structure of the code to guide the test process.

We cover the following aspects of Mutation Testing:

- What is a mutation?
- What is mutation testing?
- When should we use mutation testing?
- Mutations
- Examples
- Mutation testing tools

What is a mutation?

- A mutation is a small change in a program.
- Such small changes are intended to model low level defects that arise in the process of coding systems.
- Ideally mutations should model low-level defect creation.

What is Mutation Testing?

- Mutation testing is a structural testing method aimed at assessing/improving the **adequacy** of test suites, and estimating the number of faults present in systems under test.
- The process, given program P and test suite T, is as follows:
  - We systematically apply mutations to the program P to obtain a sequence P_1, P_2,... P_n of mutants of P. Each mutant is derived by applying a single mutation operation to P.
  - We run the test suite T on each of the mutants, T is said to **kill** mutant P_j if it detects an error.
  - If we kill k out of n mutants the adequacy of T is measured by the quotient k/n. T is **mutation adequate** if $k = n$.
- One of the benefits of the approach is that it can be almost completely automated.

When should we use mutation testing?

- Structural test suites are directed at identifying defects in the code. One goal of mutation testing is to assess or improve the efficacy of test suites in discovering defects.
- When we are carrying out structural testing we are worried about defects remaining in the code. Often we are keen to measure the **Residual Defect Density (RDD)** in the program P under test.
- The Residual Defect Density is usually measured in defects per thousand lines of code.
- Advocates of mutation testing argue that it can provide us with an estimate of the RDD of a program P that has satisfied all the tests in a test suite T.

Using Mutation Testing to Estimate the RDD

We want to estimate the RDD of Program P given that it has satisfied all the tests in test suite T. We follow the procedure:

- Suppose we have an estimate r of the RDD of programs produced by our development process before they are subject to test (this could be gathered using production data and field experience, or it could be based on the number of faults our tests have already detected).
- Generate n mutants of the program P.
- Test each mutant with the test suite T.
- Find the number, k, of mutants that are killed by T. To yield a non-zero RDD we need to test enough mutants to ensure that $0 < k < n$.
- Use $r.\ (n - k)/k$ as the estimate for the RDD of the tested program.
- k/n is a measure of the adequacy of T in finding defects in P.

Assumptions

The validity of this rests on many assumptions:

- That mutations are a good model for defects.
- That defects are usually independent
- That the construction of T is not influenced by knowledge of the mutation process (i.e. we do not use knowledge of the mutation process to build tests that are better at finding defects generated by mutations than normal defects).
- If we are interested in making confident estimates of very low RDDs we will need very large numbers of mutants.
- For example, if our development process left us with 10 defects per kLoc before test and we want to be confident our RDD after test is lower that 0.1 per kLoC then we need to test many mutants to be confident of such an RDD estimate.

An Approach to Mutation

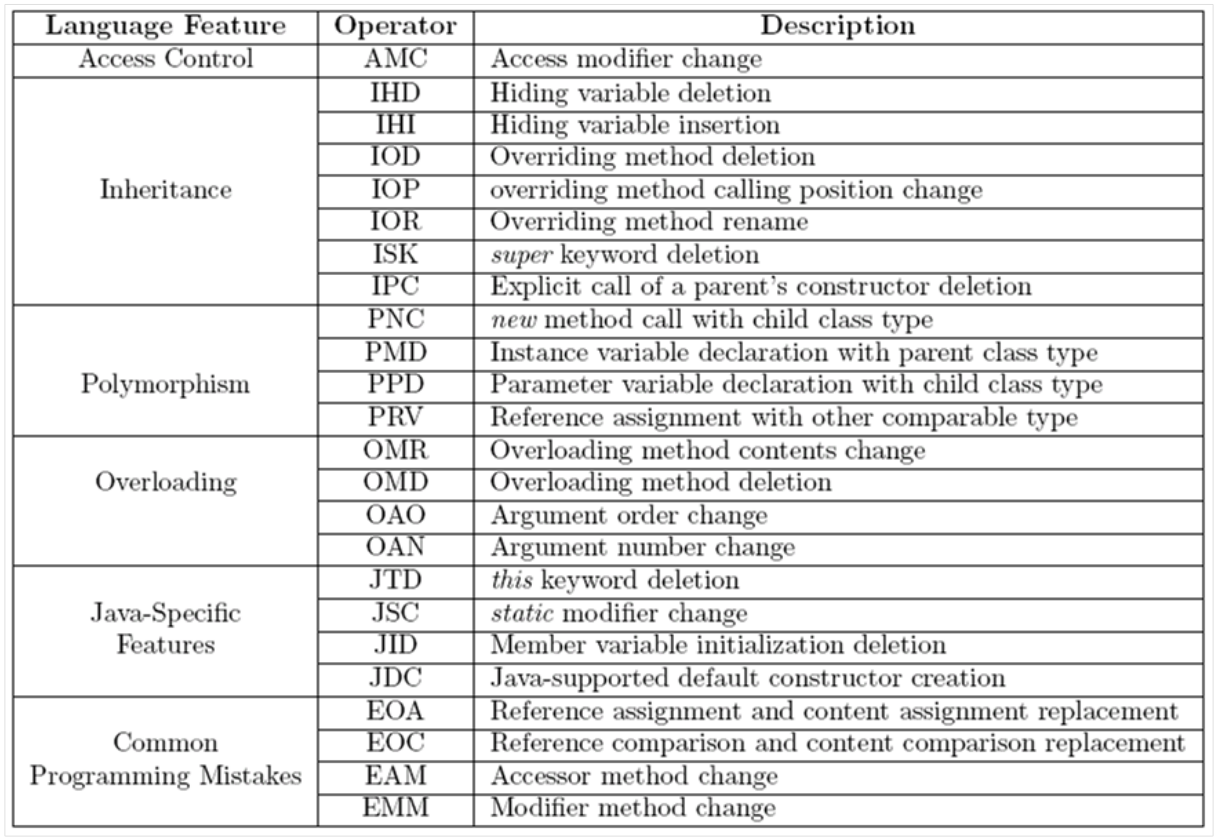
- Ideally we need systematically to apply mutations to the program under test. This would involve some criterion of applicability.
- Usually we consider mutation operators in the form of rules that match a context and create some systematic mutation of the context to create a mutant.
- The simple approach to coverage is to consider all possible mutants but this may create a very large number of mutants (in the case of estimating RDDs the value and confidence required of the estimated RDD would control the number of mutants to be generated).
- Mutation testing is best supported by tools because of the potentially very large numbers of mutations to be generated during testing.

Kinds of Mutation

- **Value Mutations:** these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.
- **Decision Mutations:** this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing a $>$ by a $<$ in a comparison.
- **Statement Mutations:** these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

A wide range of mutation operators is possible...

Offutt's Mutations for Inter-Class Testing



Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Value Mutation

- Here we attempt to change values to reflect errors in reasoning about programs.
- Typical examples are:
  - Changing values to one larger or smaller (or similar for real numbers).
  - Swapping values in initialisations.
- The commonest approach is to change constants by one in an attempt to generate a one-off error (particularly common in accessing arrays).
- Coverage criterion: Here we might want to perturb all constants in the program or unit at least once or twice.

Value Mutation

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]  l){
                 k++;
            }
      }
      return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

Decision Mutation

- Here again we design the mutations to model failures in reasoning about conditions in programs. As before this is a very limited model of programming error really modelling slips in coding rather than a design error.
- Typical examples are:
  – Modelling "one-off" errors by changing $<$ to $<=$ or vice versa (this is common in checking loop bounds).
  – Modelling confusion about larger and smaller, so changing $>$ to $<$ or vice versa.
  – Getting parenthesisation wrong in logical expressions e.g. mistaking precedence between $\&\&$ and $||$
- Coverage Criterion: We might consider one mutation for each condition in the program. Alternatively we might consider mutating all relational operators (and logical operators e.g. replacing $||$ by $\&\&$ and vice versa)

Decision Mutation

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to t[i]>u will cause miscounting

We can model “one-off” errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.

Statement Mutation

- Here the goal is primarily to model editing slips at the line level – these typically arise when the developer is cutting and pasting code. The result is usually omitted or duplicated code. In general we might consider arbitrary deletions and permutations of the code.
- Typical examples include:
  - Deleting a line of code
  - Duplicating a line of code
  - Permuting the order of statements.
- Coverage Criterion: We might consider applying this procedure to each statement in the program (or all blocks of code up to and including a given small number of lines).

Statement Mutation

```
public int Segment(int t[], int l, int u){
        // Assumes t is in ascending order, and l<u,
        // counts the length of the segment
        // of t with each element l<t[i]<u
        int k = 0;

        for(int i=0; i<t.length && t[i]<u; i++){
                if(t[i]>l){
                        k++;
                }
        }
        return(k);
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

Observations

- Mutations model low level errors in the mechanical production process. Modelling design errors is much harder because they involve large numbers of coordinated changes throughout the program.
- Ensuring test sets satisfy coverage criteria are often enough to ensure they kill mutants (because mutants often do not “make sense” and so provoke a failure if they are ever executed).
- Black-box test sets are poorer at killing mutants – we’d expect this because black-box tests are driven more by the operational profile than by the need to cover statements.
- We could see mutation testing as a way of forcing more diversity on the development of test sets if we use a black-box approach as our primary test development approach.

Concepts from the literature

- **Syntactic vs semantic size of a mutant** – the size the source change a mutant involves, versus the size of its effect on program behaviour. It has been hypothesised that mutation operators which produce semantically small faults are better (because semantically large faults will be caught by most tests). Justification for elimination of certain types of mutation.
- **Competent programmer hypothesis** – the program under test is “close to” the correct program. So exploring the space of small mutations will lead us to that program.
- **Coupling effect hypothesis** — tests for detecting simpler faults will be sufficient also for detecting more complex faults. So even though many faults are a product of logical errors with wide consequences in the code, small mutants will lead to recognition of these faults.

Mutation Testing Tools

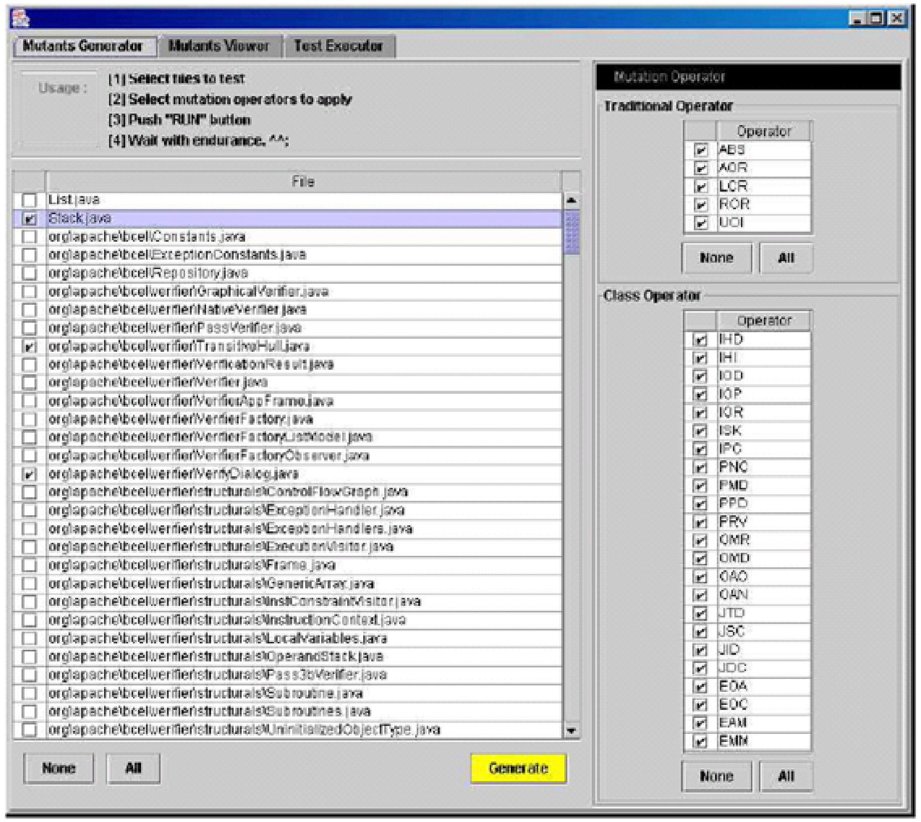
- There is a range of possible mutation tools. Recently Offutt and others have created MuJava, a tool for creating Java mutants.

  MuJava: An Automated Class Mutation System, Yu-Seung Ma, Jeff Offutt and Yong-Rae Kwon. Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005. `http://dx.doi.org/10.1002/stvr.v15:2`

- Their system is designed specifically to include a range of mutation operators that target OO languages in particular.

- They incorporate an efficient version of generating a "metamutant" that is capable of behaving like all mutants of the program (using Java reflection to instantiate operators at execution time).

Mutant Generation Interface



Usage :
[1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

	File
☐	List.java
✓	Stack.java
☐	org\apache\bcel\Constants.java
☐	org\apache\bcel\ExceptionConstants.java
☐	org\apache\bcel\Repository.java
☐	org\apache\bcel\verifier\GraphicalVerifier.java
☐	org\apache\bcel\verifier\NativeVerifier.java
☐	org\apache\bcel\verifier\PassVerifier.java
✓	org\apache\bcel\verifier\TransitiveHull.java
☐	org\apache\bcel\verifier\VerificationResult.java
☐	org\apache\bcel\verifier\Verifier.java
☐	org\apache\bcel\verifier\VerifierAppFrame.java
☐	org\apache\bcel\verifier\VerifierFactory.java
☐	org\apache\bcel\verifier\VerifierFactoryListModel.java
☐	org\apache\bcel\verifier\VerifierFactoryObserver.java
✓	org\apache\bcel\verifier\VerifyDialog.java
☐	org\apache\bcel\verifier\structurals\ControlFlowGraph.java
☐	org\apache\bcel\verifier\structurals\ExceptionHandler.java
☐	org\apache\bcel\verifier\structurals\ExceptionHandlers.java
☐	org\apache\bcel\verifier\structurals\ExecutionVisitor.java
☐	org\apache\bcel\verifier\structurals\Frame.java
☐	org\apache\bcel\verifier\structurals\GenericArray.java
☐	org\apache\bcel\verifier\structurals\InstConstraintVisitor.java
☐	org\apache\bcel\verifier\structurals\InstructionContext.java
☐	org\apache\bcel\verifier\structurals\LocalVariables.java
☐	org\apache\bcel\verifier\structurals\OperandStack.java
☐	org\apache\bcel\verifier\structurals\Pass3bVerifier.java
☐	org\apache\bcel\verifier\structurals\Subroutine.java
☐	org\apache\bcel\verifier\structurals\Subroutines.java
☐	org\apache\bcel\verifier\structurals\UninitializedObjectType.java

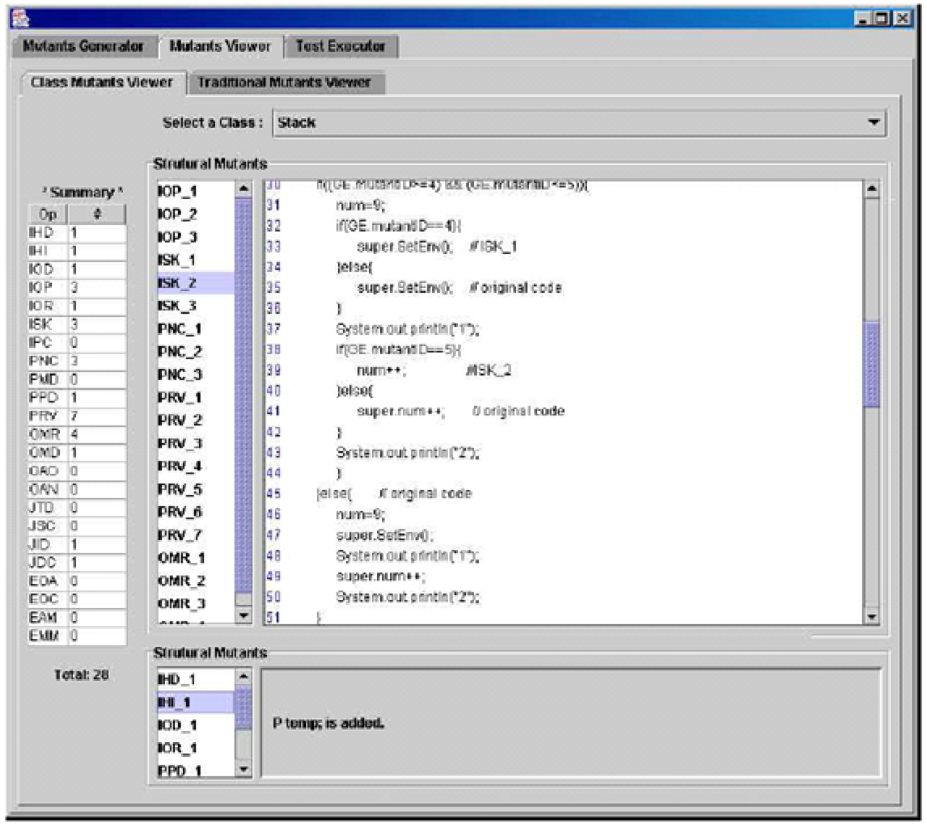
Traditional Operator

	Operator
✓	ABS
✓	AOR
✓	LCR
✓	ROR
✓	UOI

Class Operator

	Operator
✓	IHD
✓	IHI
✓	IOD
✓	IOP
✓	IOR
✓	ISK
✓	IPC
✓	PNC
✓	PMD
✓	PPD
✓	PRV
✓	OMR
✓	OMD
✓	OAO
✓	OAN
✓	JTD
✓	JSC
✓	JID
✓	JDC
✓	EOA
✓	EOC
✓	EAM
✓	EMM

Mutant Analysis Interface



Select a Class : Stack

' Summary '

Op	#
IHD	1
IHI	1
IOD	1
IOP	3
IOR	1
ISK	3
IPC	0
PNC	3
PMD	0
PPD	1
PRV	7
OMR	4
OMD	1
OAO	0
OAN	0
JTD	0
JSC	0
JID	1
JDC	1
EOA	0
EOC	0
EAM	0
EMM	0

Total: 28

Strutural Mutants

IOP_1
IOP_2
IOP_3
ISK_1
ISK_2
ISK_3
PNC_1
PNC_2
PNC_3
PRV_1
PRV_2
PRV_3
PRV_4
PRV_5
PRV_6
PRV_7
OMR_1
OMR_2
OMR_3

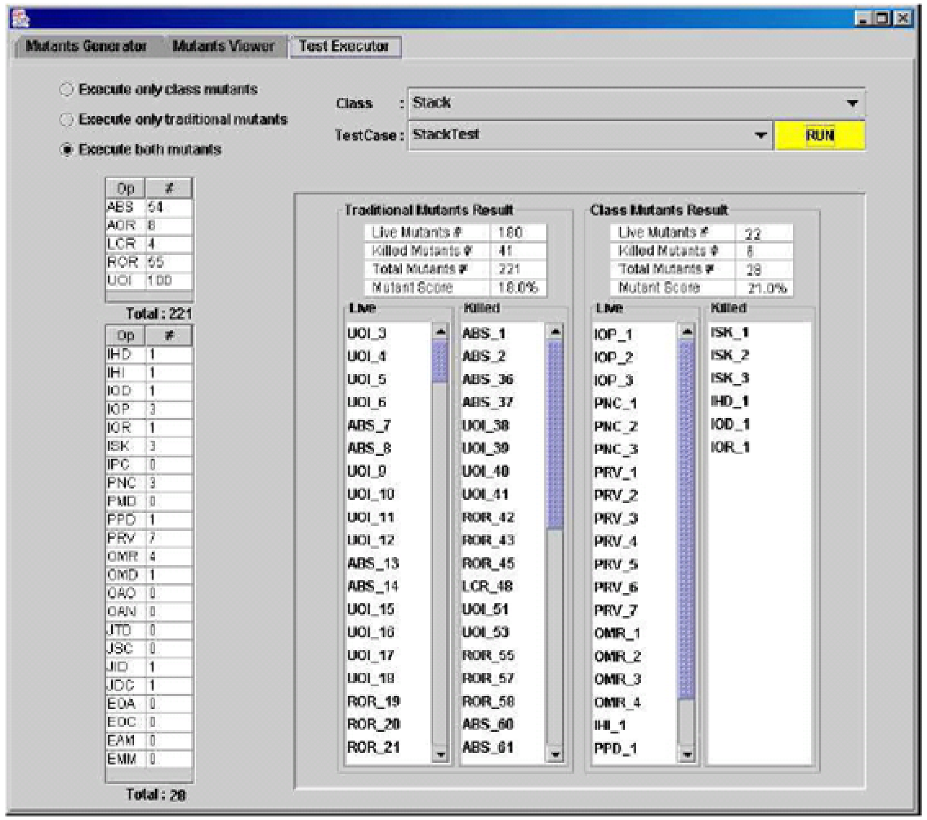
```
if((GE.mutantID>=4) && (GE.mutantID<=5)){
    num=9;
    if(GE.mutantID==4){
        super.SetEnv();   //ISK_1
    }else{
        super.SetEnv();   // original code
    }
    System.out.println("1");
    if(GE.mutantID==5){
        num++;            //ISK_2
    }else{
        super.num++;      // original code
    }
    System.out.println("2");
    }
}else{     // original code
    num=9;
    super.SetEnv();
    System.out.println("1");
    super.num++;
    System.out.println("2");
}
```

Strutural Mutants

IHD_1
IHI_1
IOD_1
IOR_1
PPD_1

P temp; is added.

Test Execution Interface



○ Execute only class mutants
○ Execute only traditional mutants
● Execute both mutants

Class : Stack
TestCase : StackTest

Op	#
ABS	54
AOR	8
LCR	4
ROR	55
UOI	100

Total : 221

Op	#
IHD	1
IHI	1
IOD	1
IOP	3
IOR	1
ISK	3
IPC	0
PNC	3
PMD	0
PPD	1
PRV	7
OMR	4
OMD	1
OAO	0
OAN	0
JTD	0
JSC	0
JID	1
JDC	1
EOA	0
EOC	0
EAM	0
EMM	0

Total : 28

Traditional Mutants Result

Live Mutants #	180
Killed Mutants #	41
Total Mutants #	221
Mutant Score	18.0%

Live	Killed
UOI_3	ABS_1
UOI_4	ABS_2
UOI_5	ABS_36
UOI_6	ABS_37
ABS_7	UOI_38
ABS_8	UOI_39
UOI_9	UOI_40
UOI_10	UOI_41
UOI_11	ROR_42
UOI_12	ROR_43
ABS_13	ROR_45
ABS_14	LCR_48
UOI_15	UOI_51
UOI_16	UOI_53
UOI_17	ROR_55
UOI_18	ROR_57
ROR_19	ROR_58
ROR_20	ABS_60
ROR_21	ABS_61

Class Mutants Result

Live Mutants #	22
Killed Mutants #	6
Total Mutants #	28
Mutant Score	21.0%

Live	Killed
IOP_1	ISK_1
IOP_2	ISK_2
IOP_3	ISK_3
PNC_1	IHD_1
PNC_2	IOD_1
PNC_3	IOR_1
PRV_1	
PRV_2	
PRV_3	
PRV_4	
PRV_5	
PRV_6	
PRV_7	
OMR_1	
OMR_2	
OMR_3	
OMR_4	
IHI_1	
PPD_1	

Summary

- Mutation testing can be a useful addition to the test process.
- It is laborious and really requires tool assistance if it is to be cost-effective.
- Improving Residual Defect Density estimates requires very large numbers of mutants if we are to have confidence in the results.
- Object Orientation has a wide range of structural and operational mutants that are specific to objects.
- Tools like mu-Java use features of Java to enable the efficient generation and test of mutants.
- Even with efficient techniques execution times can be very slow for large numbers of mutants.

Required Readings

- **Textbook (Pezzè and Young):** Chapter 16, Fault-Based Testing

- MuJava: An Automated Class Mutation System, Yu-Seung Ma, Jeff Offutt and Yong-Rae Kwon. Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005. http://dx.doi.org/10.1002/stvr.v15:2

- A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. 5, 2 (April 1996), 99-118. http://dx.doi.org/10.1145/227607.227610