SCHULICH School of Engineering

UNIVERSITY OF CALGARY

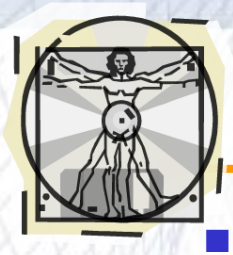
SENG 637
Dependability and Reliability of Software Systems

Chapter 6: Mutation Testing

Department of Electrical & Software Engineering, University of Calgary

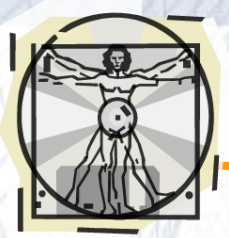
B.H. Far (far@ucalgary.ca)

http://people.ucalgary.ca/~far



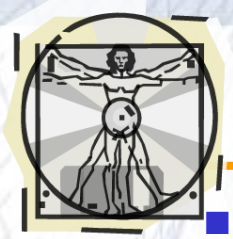
Last Two Weeks …

- Coverage-based testing
  - Black-box
    - Boundary values, equivalent classes
  - White-box
    - Control-flow: Statement, branch, conditions, MC/DC, path
    - Data flow: DU pair, DU path, all uses
- Weakness of coverage criteria as test adequacy
  - We don't know about quality of the tests
  - How good the tests are to catch a bug?
- This week: Another category of test adequacy
  - Fault-based testing: **Mutation testing**



Contents

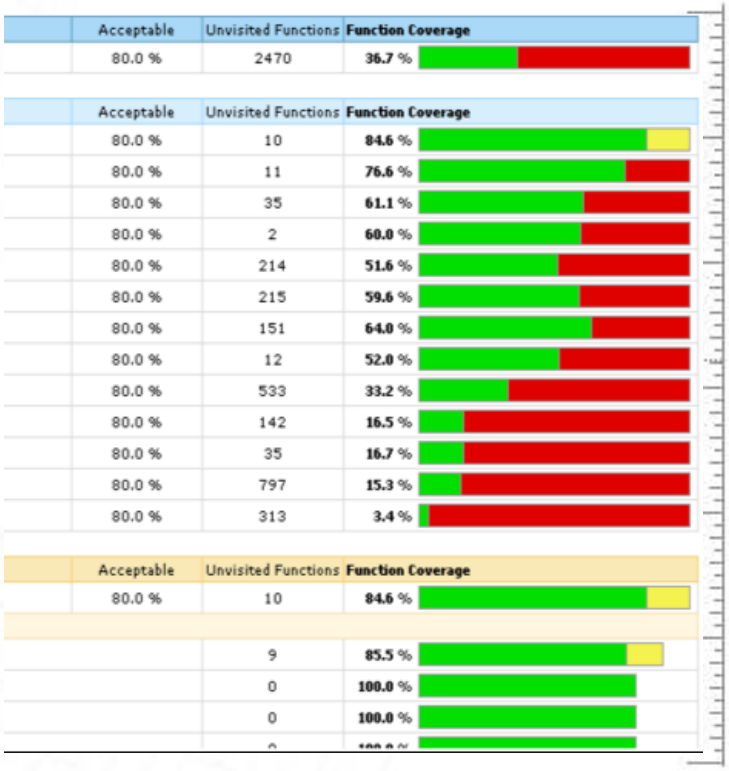
- Mutation testing concept
- Mutant types
- Mutation testing tools



Measuring Coverage

- One advantage of coverage criteria is that they can be measured ***automatically***
- To control testing **progress towards completeness**
- High coverage is **not a guarantee** of fault-free software, just an element of information to increase our confidence

Can we assess testing completeness using measures of captured and/or remaining bugs?

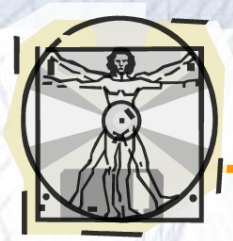


Acceptable	Unvisited Functions	Function Coverage
80.0 %	2470	36.7 %

Acceptable	Unvisited Functions	Function Coverage
80.0 %	10	84.6 %
80.0 %	11	76.6 %
80.0 %	35	61.1 %
80.0 %	2	60.0 %
80.0 %	214	51.6 %
80.0 %	215	59.6 %
80.0 %	151	64.0 %
80.0 %	12	52.0 %
80.0 %	533	33.2 %
80.0 %	142	16.5 %
80.0 %	35	16.7 %
80.0 %	797	15.3 %
80.0 %	313	3.4 %

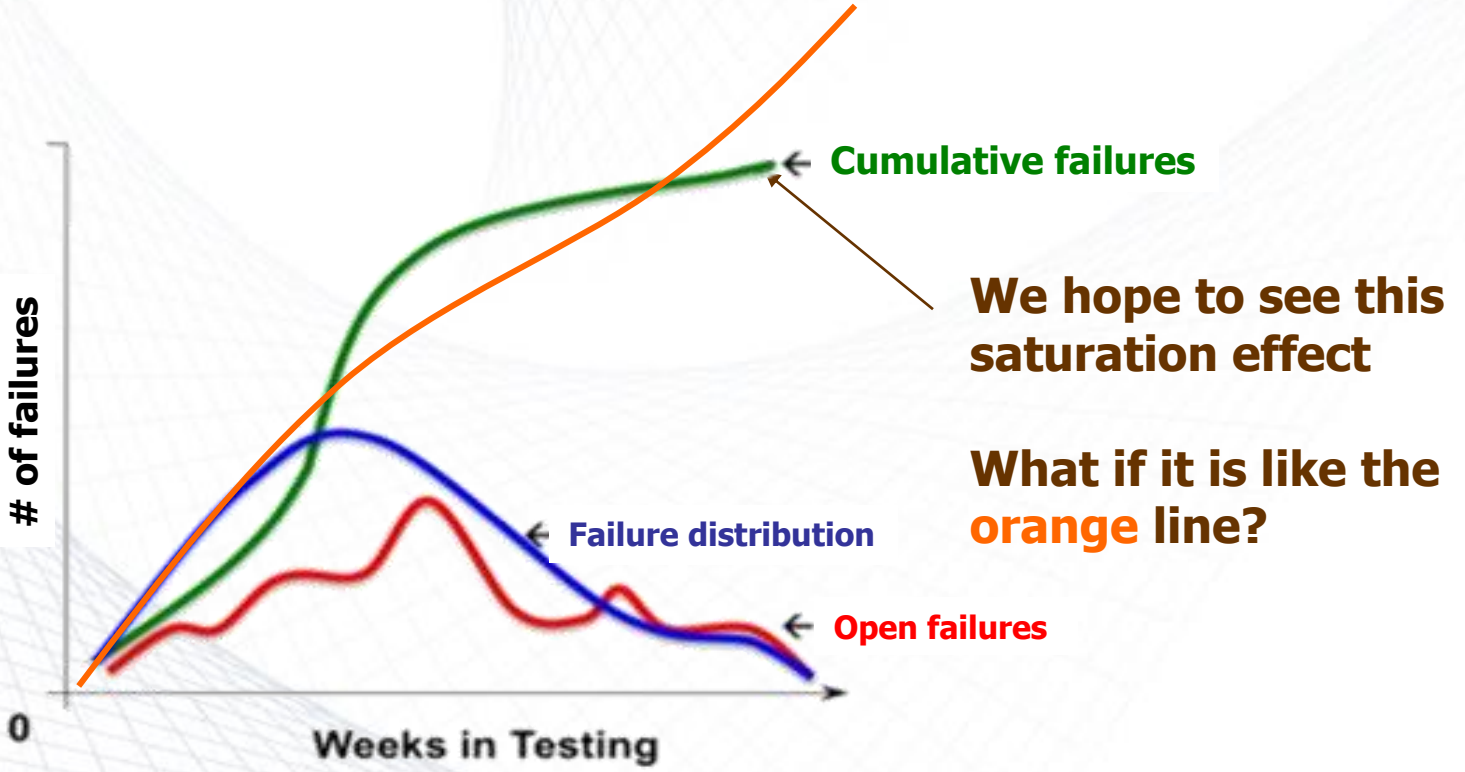
Acceptable	Unvisited Functions	Function Coverage
80.0 %	10	84.6 %
	9	85.5 %
	0	100.0 %
	0	100.0 %

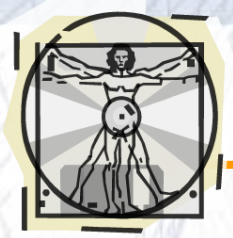
Code coverage is a measure used to describe the degree to which the source code of a program is exercised



When to Stop Testing?

- The most obvious and widely used metric: The cumulative number of failures found so far





Test Adequacy Criteria

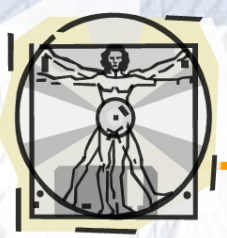
- Coverage-based (BB, WB testing)
  - *Rationale:* the better test suite covers more from specification/code/data
  - Good enough = covers most of the specification/ code/data?
- Fault-based (mutation testing)
  - *Rationale:* the better test suite detects more faults
  - Good enough = finds most of the bugs?

Challenge

- How do we know if a test suite detects most of the faults, if the faults are unknown?
  - The tests are all passing BUT
    - Is it because we have detected all faults?

    OR
    - Because we have not written good test cases?

- How do we know our test suite is good enough?



Is Code Coverage Enough?

- Company X software project manager's concern:
  - I have a test suite with 3,000 JUnit test cases
  - Test suite has almost 100% line/branch coverage
  - All 3,000 test cases are passing (great!)

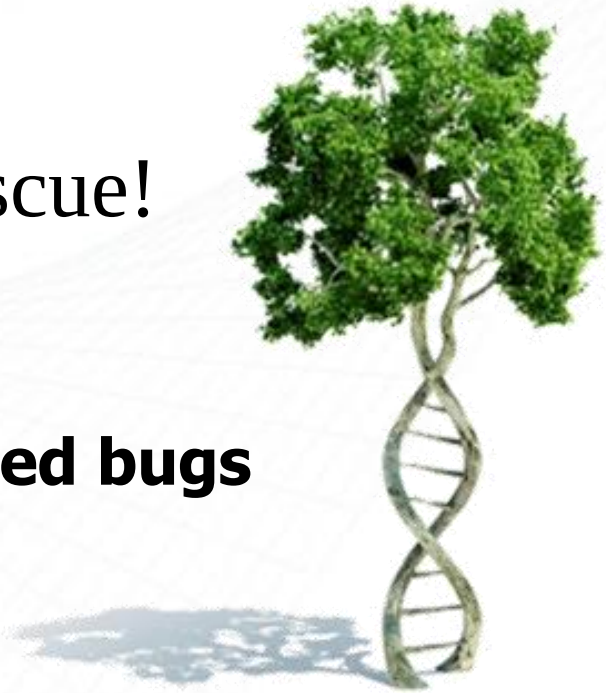
- Questions:
  - Do I have enough *verifications points* in the test suite?
  - How should I know that test suite is really good enough, in terms of fault detection effectiveness? ← **test suite quality**
  - How do I know how many more bugs are left to be found?

Best indicator for effectiveness is usually cumulative number of failures found and fixed

The Problem Still Stands

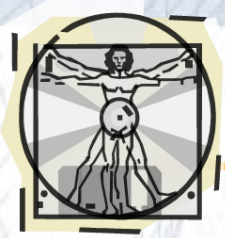
- Coverage metrics cannot ensure quality of testing (i.e. how good is your test suite)
  - Is there another way?
- Mutation testing to the rescue!

Mutation = bug injection
Mutation testing = finding injected bugs

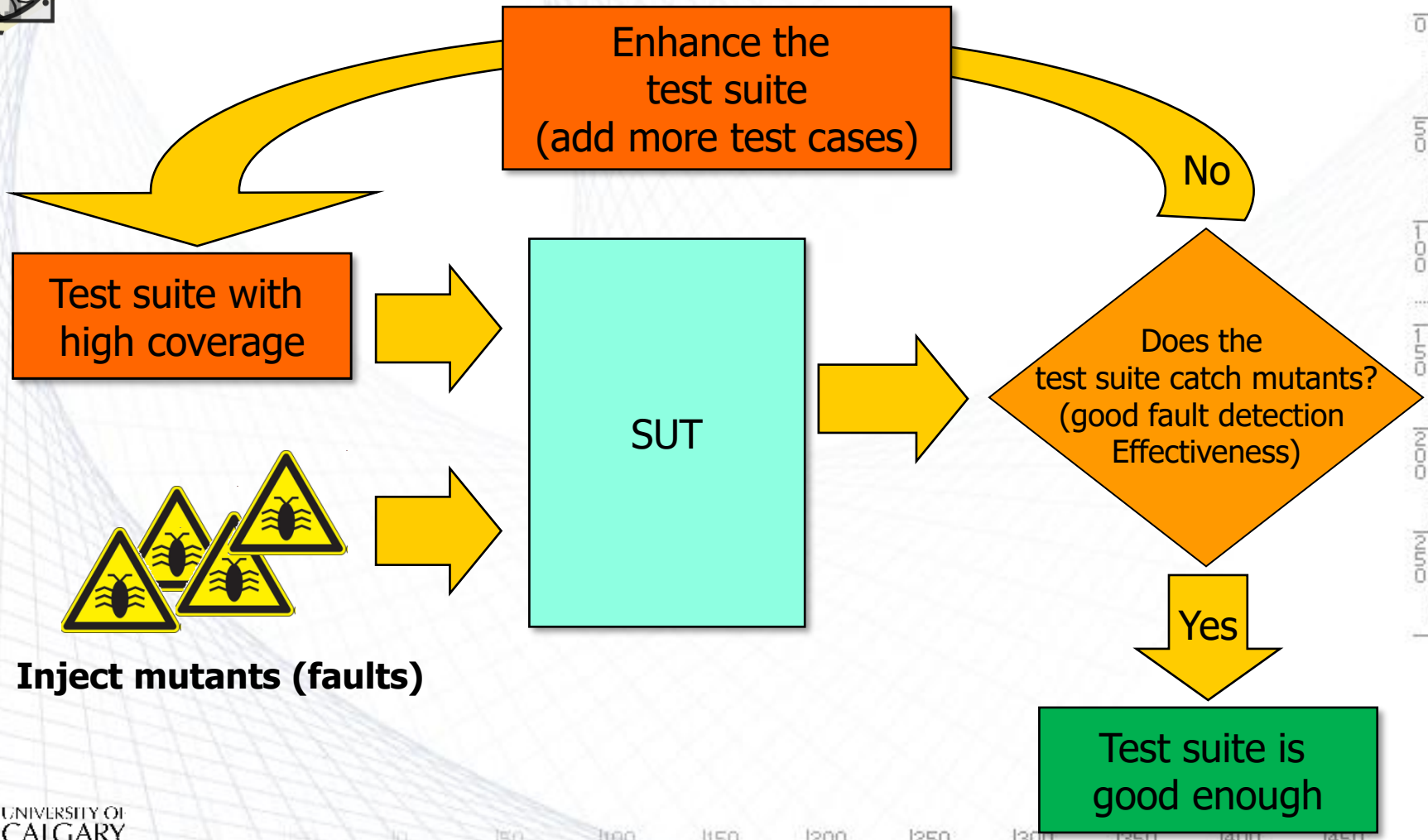


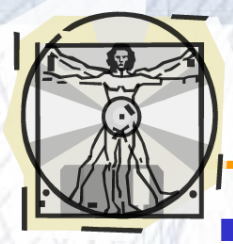
Section 1

Mutation Testing Concept



Mutation (Fault Injection) - Idea





Mutation Testing: Definition

- **Fault-based Testing:** directed towards "typical" faults that could occur in a program
- Syntactic (syntax-based) variations are applied in a systematic way to the program to create faulty versions that exhibit different behavior (each of those faulty versions are called a **Mutant**)
- e.g., changing a statement line from `x=a+b` to `x=a-b`

  or `x<10` to `x=<10` or removing a condition, etc.
- **Mutation testing** helps a user create effective test data in an interactive manner
- The goal is to have a strong (effective) test suite which will be able to catch typical faults, i.e. seeding bugs to find bugs

How Many Bugs Left?

Side advantage: Indicator of remaining bugs in the system

X non-black marbles (representing bugs)

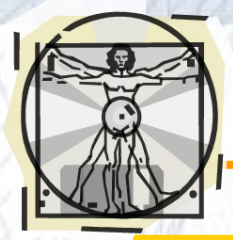
50 black marbles added to the mix (mutated bugs)

Draw 20 at random from the mixed bag: 4 black are found (represents test results)

How many total?

x=(16/4)*50=200

Expect to have total 200 bugs in the SUT (roughly!)



Mutation Testing – Example

```
void testAddPlusZero() {
    assertEquals(3, add(3, 0));
}
```

Test

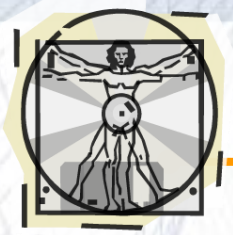
[a=3, b=0] → pass

Code (SUT)

```
int add(int a, int b) {
    return a-b;
}
```

- Is my test suite effective enough, in terms of fault detection?
- We intentionally inject a fault (e.g. change + to -) and run our test suite to see if it can detect the injected faults

If our test case is <a=3, b=0> then both a+b and a-b return 3, i.e. the test result is "Pass" but one of the programs is incorrect!



Mutation Testing: Example

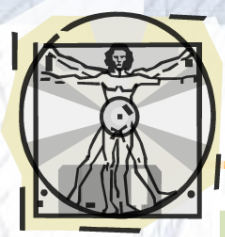
Original program

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index==10)
           break;
}
```

A mutant

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index>=10)
           break;
}
```

Test case: index=11



Mutation Testing – Example

Program P	P'=A mutant of P
int foo(int x, int y) {return(x+y)}	int foo(int x, int y) { return(x-y)}

Test T1

```
int a=foo(1,0);
assertEquals(a,1);
```

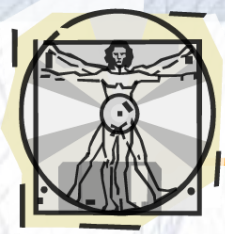
T1 on P ✓ T1 on P' ✓ **alive**

Test T2

```
int a=foo(1,1);
assertEquals(a,2);
```

T2 on P ✓ T2 on P' ✗ **killed**

A "good" test suite is supposed to **kill (all)** the mutants



Mutation Testing

Mutated Code

Execute SAME Test

Good

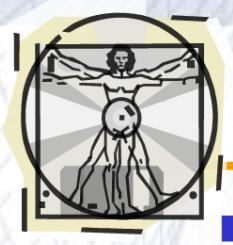
Mutant killed

Mutated Code

Execute SAME Test

NG

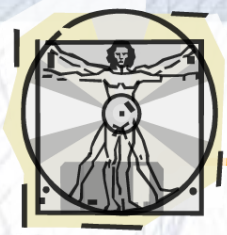
Mutant survived



Killed or Surviving?

- Surviving means changing the source code did not change the test result
  - It is bad!
  - Our test cases are ***NOT*** effective in detecting bad code (i.e. catching faults)

- Killed means changing the source code changed the test results
  - It is good!
  - Our test cases are effective in catching faults



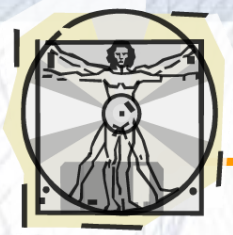
Test the Code

```
public class Math {
        public int add (int i1, int i2){
        return i1+i2;
        }
}
```

← Original code

Execute SAME Test

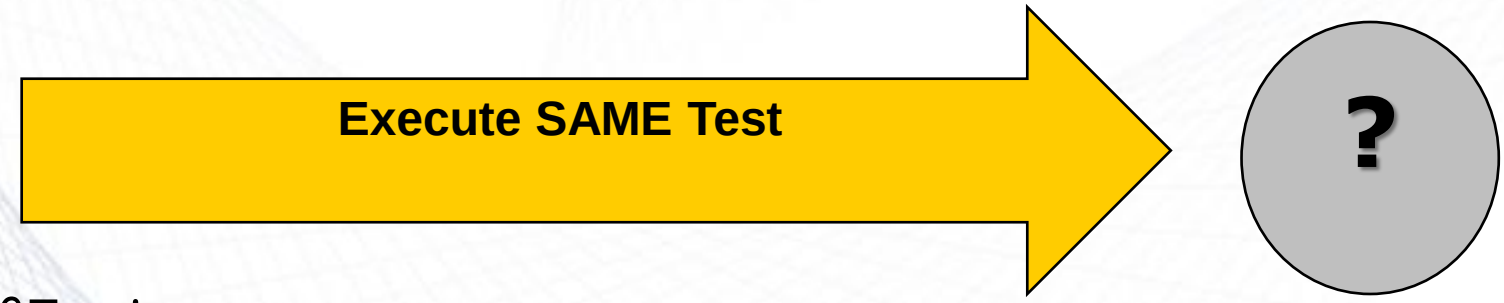
```
@Test
public void add_should_add(){
        int sum = new Math().add(1,1);
        Assert.assertEquals(sum, 2);
    }
```



Test the Code

```
public class Math {
        public int add (int i1, int i2){
        return i1-i2;
        }
}
```

← Mutated




```
@Test
public void add_should_add(){
        int sum = new Math().add(1,1);
        Assert.assertEquals(sum, 2);
    }
```

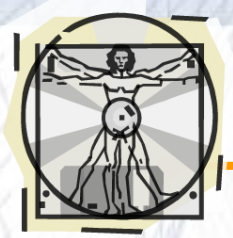
Killed/Survived Mutant

```
public class Math {
        public int add (int i1, int i2){
        return i1-i2;
        }
}
```

← Mutated

Execute SAME Test

```
@Test
public void add_should_add(){
        int sum = new Math().add(1,1);
        Assert.assertEquals(sum, 2);
    }
```



How to Seed Known Bugs?

- Manual and subjective
  - Based on historical faults/experience
    - Look at "Catalogue of Common Software Errors" (D2L)
- Systematic and automated (mutation)
  - Small changes to the source code by a tool
  - e.g. muJava, PIT, Jumble, Jester, etc.
- Use mutants:
  - Mutants are patterns applied to source code to produce mutations

Mutation Testing: Mutants

Different types of Mutants

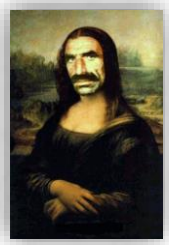
- **Stillborn mutants:** Syntactically incorrect, killed by compiler, e.g., `x=a++b`
- **Trivial mutants:** Killed by almost any test case
- **Equivalent mutant:** Always acts in the same behavior as the original program, e.g., `x=a+b` and `x=a-(-b)`

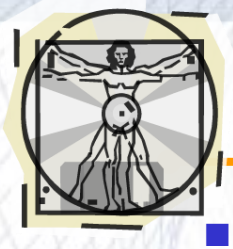
- The above cases are not interesting from a mutation testing perspective
- Only those mutants are interesting which behave differently than the original program, and we do not have test cases to identify them (to detect those specific defects)

Section 2

Mutation Testing: Mutants

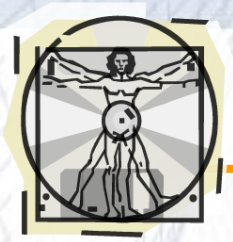


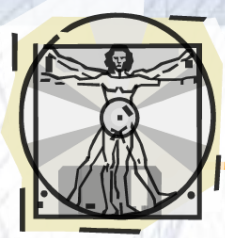
Mutation Testing /1

- We take a program (SUT) and a test suite generated for that program (e.g. Junit suite)
- We create a number of similar SUT programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
  - e.g., replacing a “less than” operator by a “less or equal” operator
- The original test suite is then run on each of the mutants
- If test cases detect differences in a mutant, then the mutant is said to be distinguished (killed)
- If not, it is considered live. We need additional test case(s) to kill it



Mutation Testing /2

- A mutant remains live either
  - because it is equivalent to the original program (functionally identical although syntactically different – called an **equivalent mutant**) or,
  - the test set is inadequate to kill the mutant
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the live mutant
- For the automated generation of mutants, we use **mutation operators**, that is predefined program modification rules (i.e., corresponding to a fault model)
- Example mutation operators next…

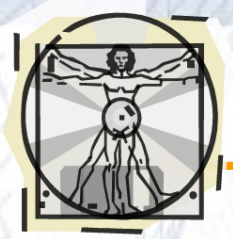


Equivalent Mutants

- P' is an Equivalent Mutant of P
- Equivalent mutants are **syntactically** different but **semantically** equivalent to the original program i.e., they are **NOT** simulating bugs
- We can not kill equivalent mutants

Program P	Mutant P'

foo()
{
for (int i=0; ***i<10***; i++)
{
(do sth // the value of i is unchanged)
}
}
	foo()
	{
	for (int i=0; ***i!=10***; i++)
	{
	(do sth // the value of i is unchanged)
	}
	}



Mutation Testing: Example

Calculate the minimum value

Original Function

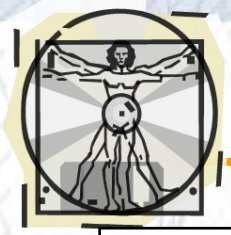
```
int Min (int A, int B)
   int minVal;
{
   minVal = A;
   if (B < A)
   {
       minVal = B;
   }
return (minVal);
} // end Min
```

With Embedded Mutants

```
     int Min (int A, int B)
        int minVal;
     {
        minVal = A;
Δ1      minVal = B;
        if (B < A)
Δ2      if (B > A)
Δ3      if (B < minVal)
        {
            minVal = B;
Δ4          Bomb();
Δ5          minVal = A;
Δ6          minVal = failOnZero (B);
        }
     return (minVal);
     } // end Min
```

Delta represents syntactic modifications

From Ammann and Offut Book



Mutation Testing: Example

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

Each of the Δi will be embedded in a different mutant (i.e. 6 mutants)

With Embedded Mutants

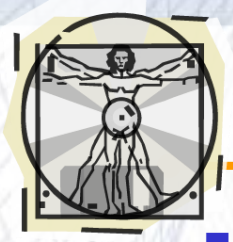
```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1     minVal = B;
        if (B < A)
Δ 2     if (B > A)
Δ 3     if (B < minVal)
        {
                minVal = B;
Δ 4             Bomb ();
Δ 5             minVal = A;
Δ 6             minVal = failOnZero (B);
        }
        return (minVal);
}
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing



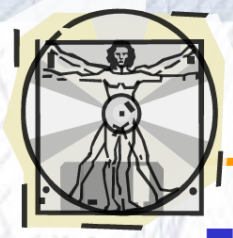
Mutation Testing: Example

- Mutant #3 is **equivalent** as, at this point, `minVal` and `A` have the same value

Original Function		With Embedded Mutants

`int Min (int A, int B)`
`  int minVal;`
`{`
`  minVal = A;`
`  if (B < A)`
`  {`
`  minVal = B;`
`  }`
`return (minVal);`
`} // end Min`
		`int Min (int A, int B)`
		`  int minVal;`
		`{`
		`  minVal = A;`
		
		`  if (B < minVal)`
		`  {`
		`  minVal = B:`
		
		`  }`
		`return (minVal);`
		`} // end Min`

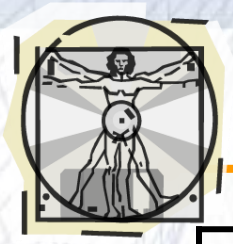
	$\Delta 3$	



Mutation Testing: Example

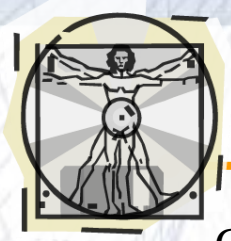
- Mutant #1: In order to find an appropriate test case to kill it, we must
  - Reach the fault seeded during execution (Reachability)
    - Always true (i.e., we can always reach the seeded fault)
  - Cause the program state to be incorrect (Infection)
    - A <> B
  - Cause the program output and/or behavior to be incorrect (Propagation)
    - (B<A) = false

Original Function		With Embedded Mutants
`int Min (int A, int B)`		`int Min (int A, int B)`
`  int minVal;`		`  int minVal;`
`{`		`{`
`  minVal = A;`		
`  if (B < A)`	Δ1	`  minVal = B;`
`  {`		`  if (B < A)`
`    minVal = B;`		
`  }`		
`return (minVal);`		`  {`
`} // end Min`		`    minVal = B;`
		`  }`
		`return (minVal);`
		`} // end Min`



Traditional (Method-Level) Mutations: Examples

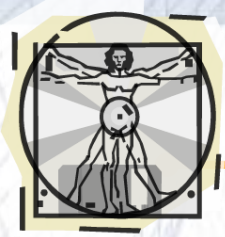
Mutant operator	In SUT	In mutant
Variable replacement	z=x*y+1;	x=x*y+1; z=x*x+1;
Relational operator replacement	if (x<y)	if(x>y) if(x<=y)
Off-by-1	z=x*y+1;	z=x*(y+1)+1; z=(x+1)*y+1;
Replacement by 0	z=x*y+1;	z=0*y+1; z=0;
Arithmetic operator replacement	z=x*y+1;	z=x*y-1; z=x+y-1;



Traditional (Method-Level) Mutations

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

More info
http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf



Object-Oriented (Class-Level) Mutations

Language Feature	Operator	Description
Encapsulation	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
	ISI	super keyword insertion
Polymorphism	PNC	new method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable

More info http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf

Mutant Operators in Java: Encapsulation

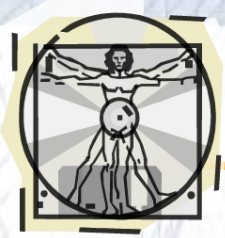
- **AMC – Access modifier change**: The AMC operator changes the access level for instance variables and methods to other access levels. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct.

Original Code

```
AMC Mutants public Stack s;
```

```
AMC Mutant

Δ private Stack s;
Δ protected Stack s;
Δ Stack s;
```



Mutant Operators in Java: Inheritance

- IHD – Hiding variable deletion: The IHD operator deletes a hiding variable, a variable in a subclass that has the same name and type as a variable in the parent class. This causes references to that variable to access the variable defined in the parent (or ancestor). This mutant can only be killed by a test case that is able to show that the reference to the parent variable is incorrect.

Original Code

```
class List {
        int size; ...
}
class Stack extends List {
        int size;
        ...
}
```

IHD Mutant

```
class List {
        int size;
        ...
}
class Stack extends List {
        Δ // int size;
        ... ...
}
```

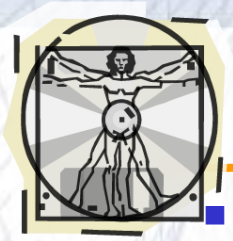
Mutant Operators in Java: Inheritance

- IHI – Hiding variable insertion: The IHI operator inserts a hiding variable into a subclass. It is a reverse case of IHD. By inserting a hiding variable, two variables (a hiding variable and a hidden variable) of the same name become to be exist. Newly defined and overriding methods in a subclass reference the hiding variable although inherited methods reference the hidden variable as before.

Original Code

```
class List {
        int size; ...
}
class Stack extends List {
                ...
}
```

```
IHD Mutant
class List {
        int size;
        ...
}
class Stack extends List {
        Δ int size;
        ... ...
}
```



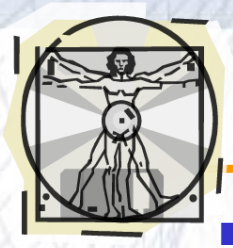
Mutant Operators in Java - Polymorphism

Parameter variable declaration with child class type: The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables. It changes the declared type of a parameter object reference to be that of the parent of its original declared type. In the example below, class Parent is the parent of class Child.

Original Code

```
boolean equals (Child o) {
...
}
```

```
PPD Mutant
Δ boolean equals (Parent o) {
  ...
  }
```

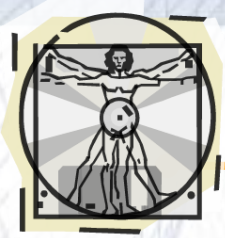


Mutation Assessment

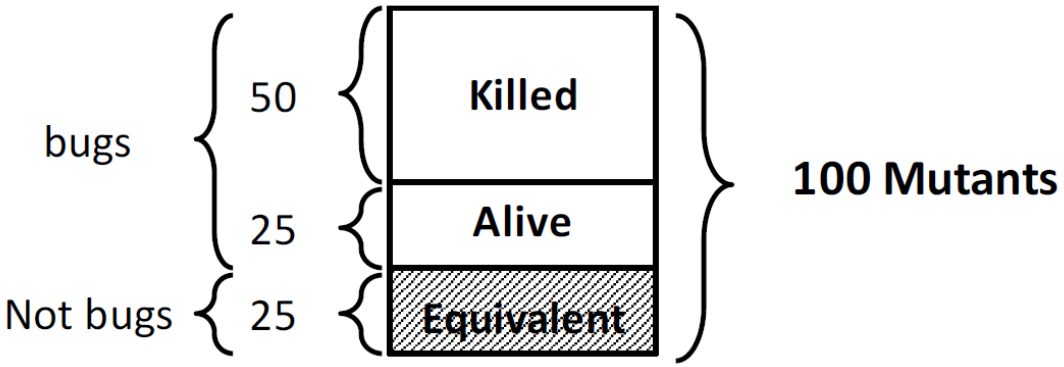
- Complete **mutation coverage** equals to killing all non-equivalent mutants
- The amount of coverage is also called **mutation score**

$$\text{Mutation score} = \frac{\text{\# of mutants killed by the test suite}}{\text{\# of all non-equivalent mutants}}$$

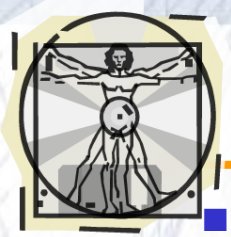
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (random sampling?)



Mutation Coverage - Example



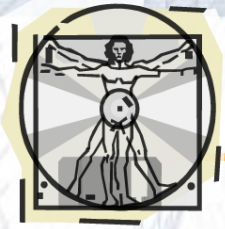
Mutation Score = 50/75 = 67%



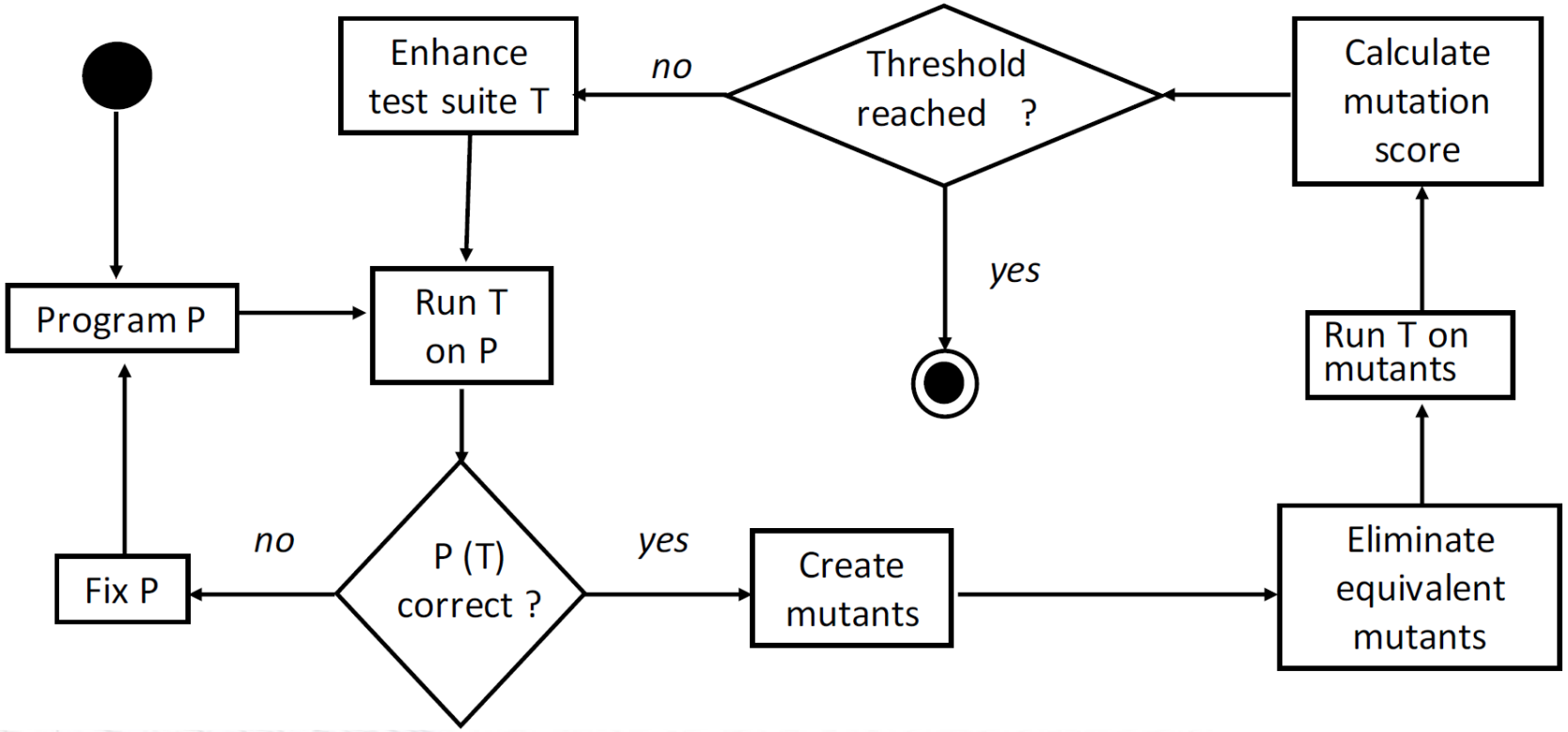
Weak vs. Strong Mutation

- Killing mutants conditions
  - A test must reach the mutated statement
  - Test input data should infect the program state by causing different program states for the mutant and the original program
  - The incorrect program state must propagate to the program's output and be checked by the test
- Strong Mutation (All three)
  - Tests actually can detect the bug
- Weak Mutation (The first two)
  - Does not guarantee that the test detect the bug but still stronger than coverage

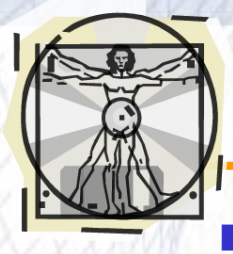
If a mutant is not killed by a test suite the test suite is inadequate and needs to be enhanced



Mutation Testing Process



[Adapted from Offutt et. al.]

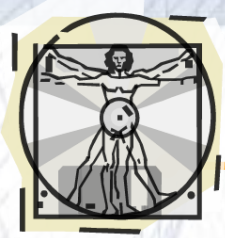


Mutation Testing: Assumptions

- What about more complex defects, involving several statements?
- Let's discuss two assumptions:
  - **Competent programmer hypothesis (assumption):** They write programs that are nearly correct
    - "The essence of this hypothesis is that the mutants simulate the likely effect of real faults. Therefore, if the test set is good at catching the artificial mutants, it will [might] also be good at catching the real mutants – the faults in our program."
  - **Coupling effect assumption:** Test cases that distinguish programs differing from a correct one by only simple errors are usually so sensitive that they would distinguish more complex errors

Mutation Testing: Discussion

- Mutation testing helps with both:
  - Test-case design and test evaluation
- It measures the quality of test cases
- It provides the tester with a clear target (mutants to kill)
- It does force the programmer to inspect the code and think of the test data that will expose certain kinds of faults
- It is computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Most useful at unit testing level

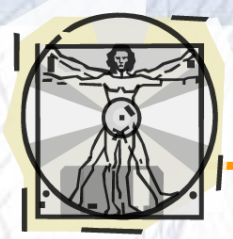


Mutation Testing: Pros

- Automated
- Systematic
- Calculate mutation score
  - When to stop testing
    - Assessing adequacy of a test suite
  - Which test suite (for a given technique) is better
    - Comparing fault detection power of different testing strategies using relative mutation score

Mutation Testing: Challenges

- Equivalent mutants
  - 7 Java programs 5,000 to 100,000 LOC
  - Between **4%** to **45%** equivalent mutants
  - Equivalent mutant detection is feasible in practice, though it is generally an **undecidable** problem
- Too many mutants: Resources required
  - Each manual detection takes on average 15 min
  - In one program: 24,000 mutants ~ **one person-year**
- Are mutants representative of real faults?



Example

- Requirements:
  - Say, there is a hospital site that lets new users register. It reads the Date of birth or age of the patient. If the age is greater than 14, assigns a general physician as their main doctor. To do so, it invokes the 'General_Physician' function that finds the available doctor.
- Program (pseudo) code:

```
1) Read Age
2) if age>14
3) Doctor= General_Physician()
4) end if
```

- Test suite: Age = 14, 15, 0, 13

  (based on boundary conditions)

Example: Mutants

```
Mutant #1:
1) Read Age
2) if age<14    //'Changing the > with <'
3) Doctor= General_Physician()
4) end if
```

```
Mutant #2:
1) Read Age
2) if age=14    //'Changing the > with ='
3) Doctor= General_Physician()
4) end if
```

```
Mutant #3:
1) Read Age
2) if age>=14    //'Changing the > with >='
3) Doctor= General_Physician()
4) end if
```

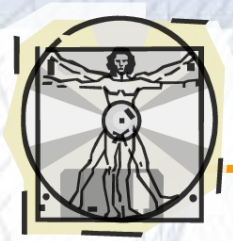
```
Mutant #4:
1) Read Age
2) if age<=14    //'Changing the > with <='
3) Doctor= General_Physician()
4) end if
```

Example: Mutants

```
Mutant #5: Statement Removal
1) Read Age
2) if age>14
   //'remove the doctor assignment statement'
3) end if
```

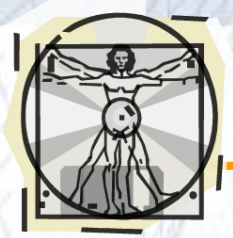
```
Mutant #6: Absolute Value Insertion
1) Read Age
2) if age>14
3) Doctor= Mr_X  //(Absolute value insertion- X is a pediatrician)
4) end if
```

```
Mutant #7: Incorrect syntax
1) Read Age
2) if age%%14 //(incorrect syntax)
3) Doctor=General_Physician()
4) end if
```



Example: Mutants

```
Mutant #8: Does the same thing as the original test
1) Read Age
2) if (age>14 && age>14) //'means the same thing as age>14'
3) Doctor= General_Physician()
4) end if
```



Example: Results

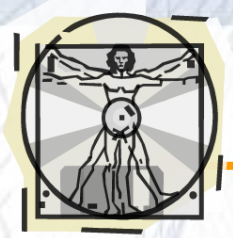
- Running tests: Age = 14, 15, 0, 13

Test set-data	Expected result	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 5	Mutant 6	Mutant 7	Mutant 8
14	GP not assigned	Success-Not assigned	Fails- GP assigned	Fails-GP assigned	Fails- GP assigned	Nothing happens	Success-Not assigned	Syntax error	Success-GP Not assigned
15	GP is assigned	Fail- GP not assigned	Fail- GP not assigned	Success-GP assigned	Fail- GP not assigned	Nothing happens	Fail- GP not assigned	Syntax error	Success-GP assigned
0	GP not assigned	Fails- GP assigned	Success-Not assigned	Success-Not assigned	Fails- GP assigned	Nothing happens	Success-Not assigned	Syntax error	Success-Not assigned
13	GP not assigned	Fails- GP assigned	Success-Not assigned	Success-Not assigned	Fails- GP assigned	Nothing happens	Success-Not assigned	Syntax error	Success-Not assigned

Trivial

Still-born

Equivalent

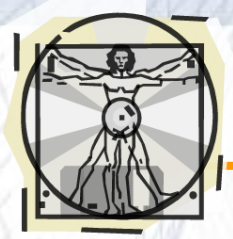


Example: Results

- Running tests: Age = 14, 15, 0, 13

Test set-data	Expected result	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 6
14	GP not assigned	Success- Not assigned	Fails- GP assigned	Fails- GP assigned	Fails- GP assigned	Success- Not assigned
15	GP is assigned	Fail- GP not assigned	Fail- GP not assigned	Success- GP assigned	Fail- GP not assigned	Fail- GP not assigned
0	GP not assigned	Fails- GP assigned	Success- Not assigned	Success- Not assigned	Fails- GP assigned	Success- Not assigned
13	GP not assigned	Fails- GP assigned	Success- Not assigned	Success- Not assigned	Fails- GP assigned	Success- Not assigned

If a mutant is "killed" by at least one test case, you are good, i.e., for each mutant (column) if you see at least one red box, your test suite will be good.



Example

- **Mutants to avoid are:**
  - **Syntactically incorrect/'Still-Born' mutants:** You need syntactically correct mutants ONLY
    - Example: Mutant 7
  - **Equivalent Mutants:** The ones that do the exact same thing as the original program
    - Example: Mutant 8
  - **Trivial Mutant:** Can be killed by any data-set
    - Example: Mutant 5
- If there are some mutants alive at the end of the test. It means either it is an invalid mutant (like #5, 7 and 8) or the test set was inadequate. In the latter case, revise test set

SCHULICH School of Engineering

UNIVERSITY OF CALGARY

Section 3

Mutation Testing Tools

http://pitest.org/

Mutation Testing Tools - PIT

pitest.org

≈ home · quickstart · faq · downloads · blog · email · rss ≈

PIT Mutation Testing

PIT is a fast bytecode based **mutation testing** system for Java that makes it possible to test the effectiveness of your unit tests.

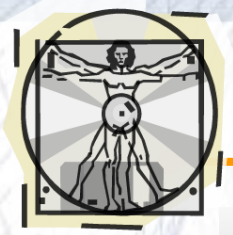
You can think of **mutation testing** as either as an automated test of your tests, or as a much more in depth form of code coverage.

Unlike traditional line and branch coverage tools PIT does not just confirm that your tests execute your code, it confirms that your tests are actually able to detect faults in it.

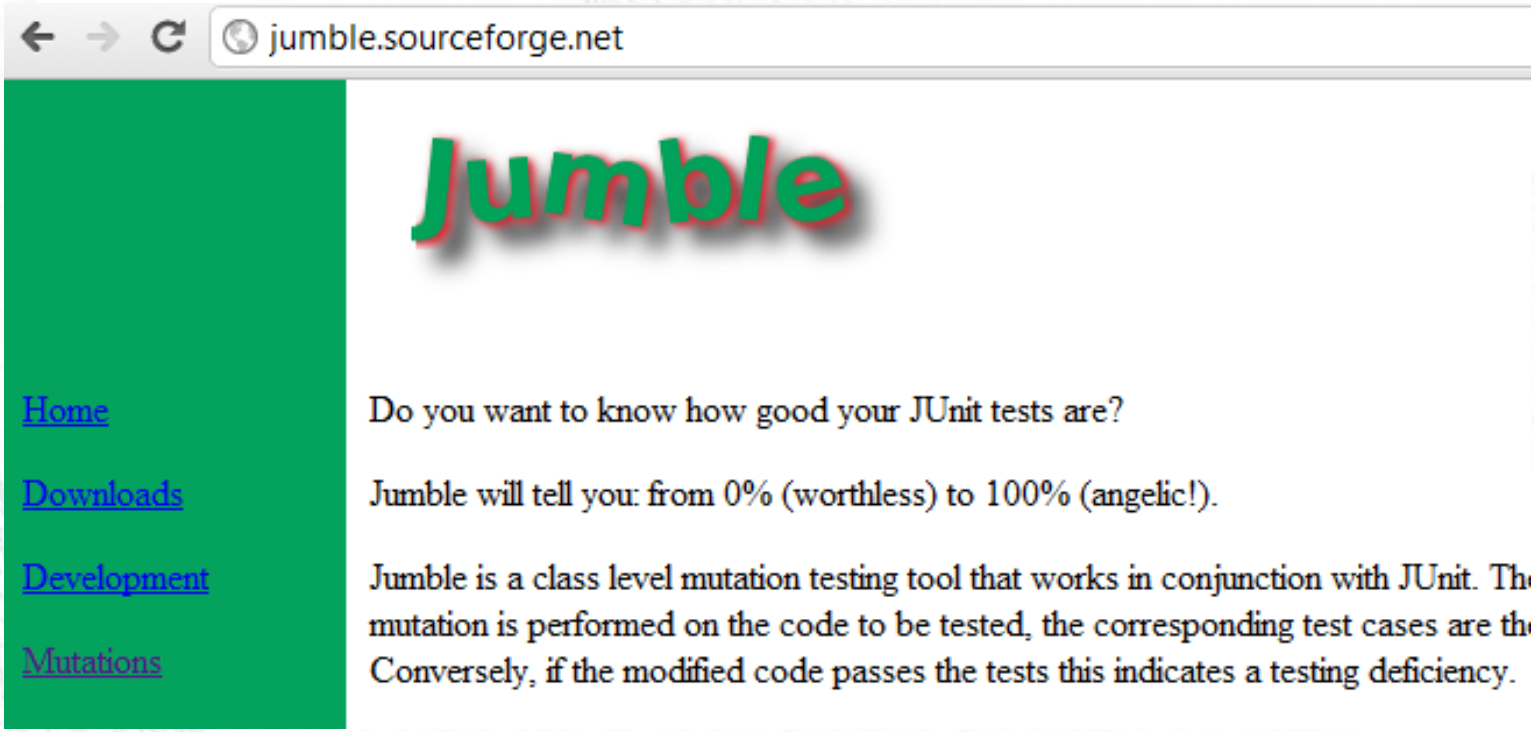
```
1   package pl.kaczanowscy.tomek;
2
3   public class MyClass {
4
5       public boolean myMethod(int a, boolean flag) {
6   3       if (a > 0 && flag) {
7   1           return true;
8           }
9   1       return false;
10      }
11  }
```

Mutations

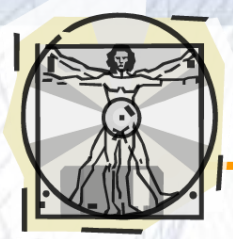
```
    negated conditional : KILLED -> pl.kaczanowscy.tomek.MyClassTest._
6   changed conditional boundary : SURVIVED
    negated conditional : KILLED -> pl.kaczanowscy.tomek.MyClassTest._
7   replaced return of integer sized value with (x == 0 ? 1 : 0) : KILLED -> pl.kaczanowscy.tomek.MyClassTest._
9   replaced return of integer sized value with (x == 0 ? 1 : 0) : KILLED -> pl.kaczanowscy.tomek.MyClassTest._
```



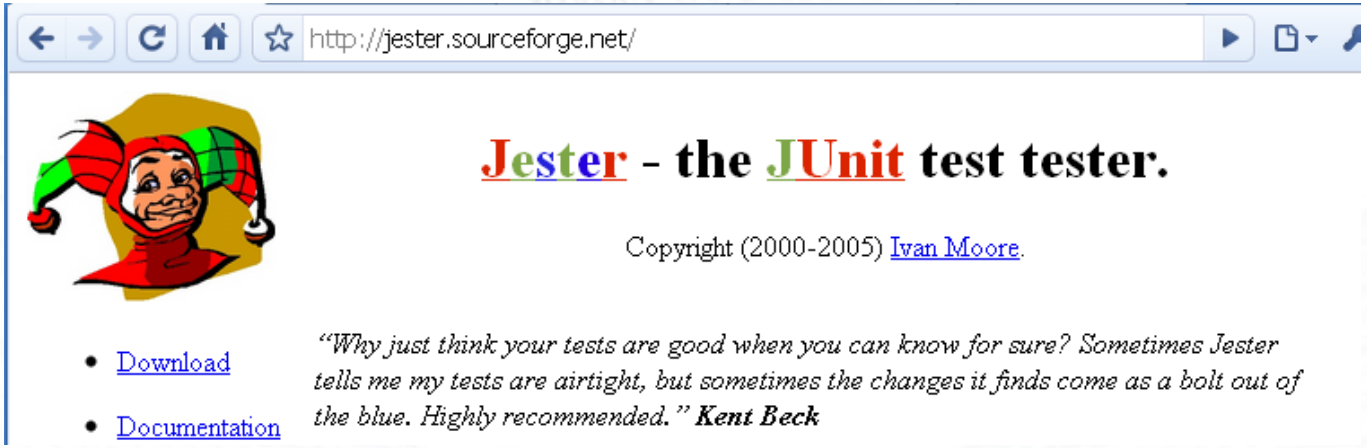
Mutation Testing Tools - Jumble



No longer under active development



Mutation Testing Tools

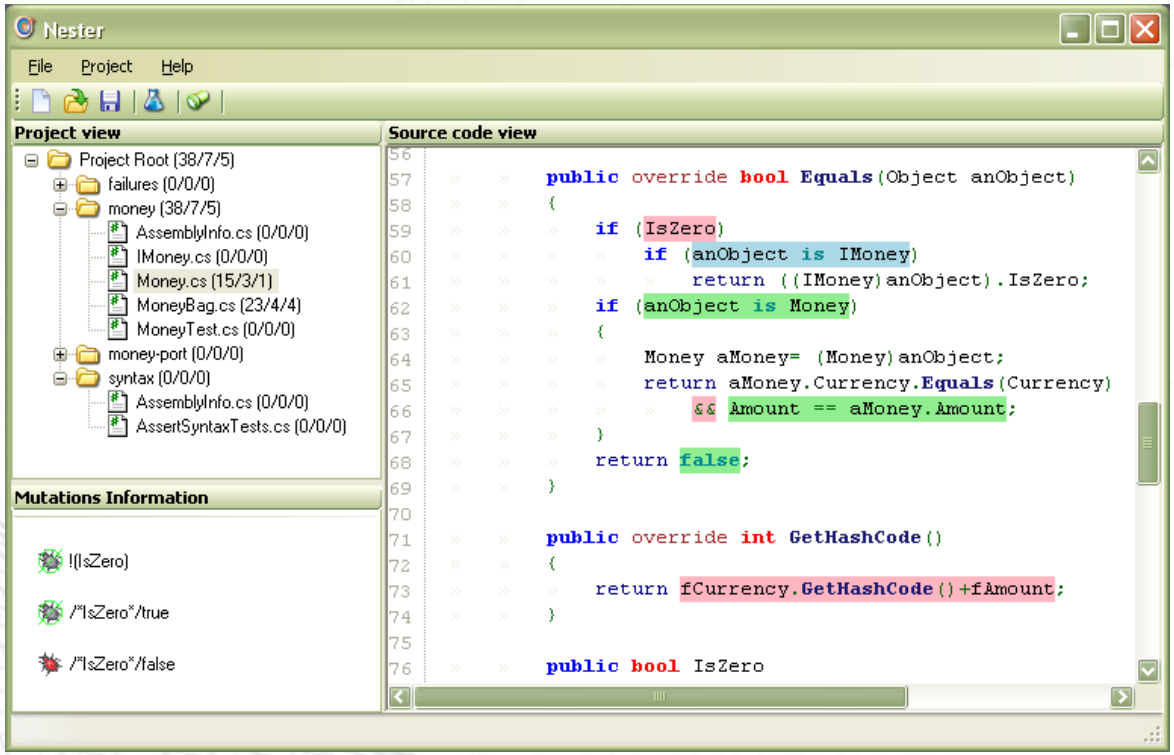


http://jester.sourceforge.net/

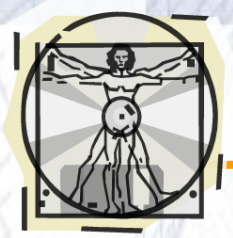
- **Jester: Mutation Testing tool for Java (Open Source)**
- **Pester: Mutation Testing tool for Python (Open Source)**
- **Nester: Mutation Testing tool for C# (Open Source)**

Mutation Testing Tools - Nester

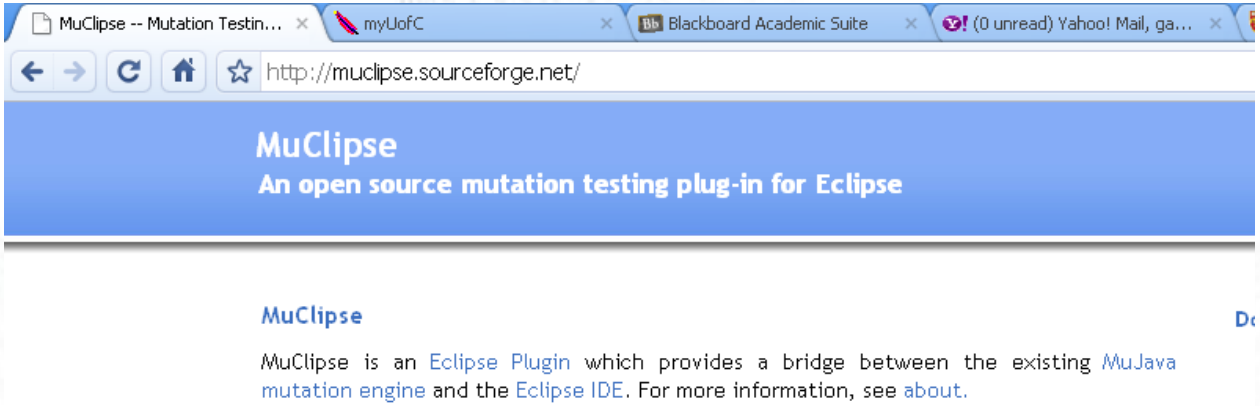
- Green code spots stand for killed mutations
- Red code spots represent survived mutations
- Blue code spots represent pieces not covered by unit tests



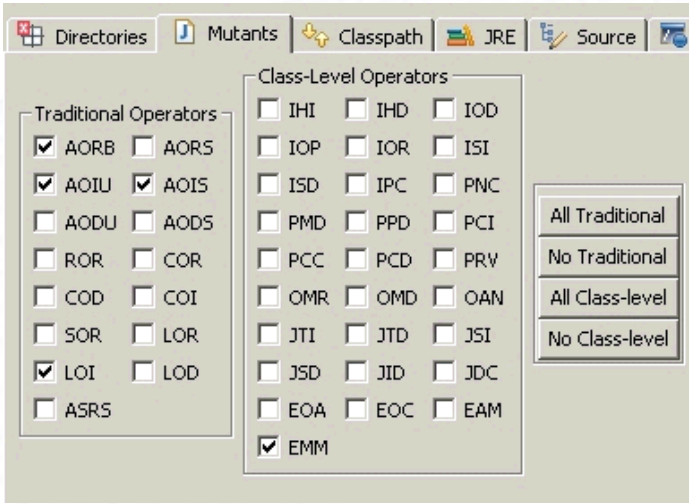
No longer under active development



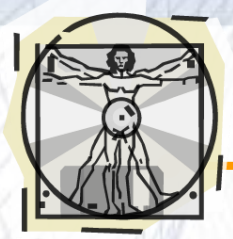
Mutation Testing Tools - MuClipse



Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

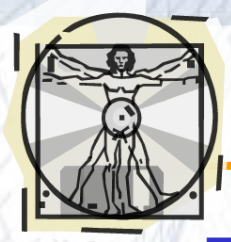


http://muclipse.sourceforge.net/



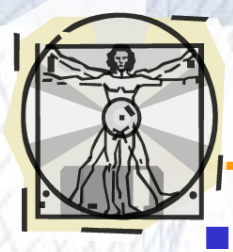
Other Mutation Testing Tools

- Major mutation framework
  - http://mutation-testing.org/
- Javalanche
  - https://github.com/david-schuler/javalanche/
- MuJava
  - http://cs.gmu.edu/~offutt/mujava/



Conclusions

- Code coverage only guarantees the execution of the code segment (reachability), however, mutation score makes sure a statement is reached and checks the state of the execution
- Are we done with Unit testing? Almost there …
  - But we won't go any further
  - There are other approaches
    - Model-based approach
    - …



References

- Mauro Pezzè, "**Software Testing and Analysis: Process, Principles, and Techniques**", John Wiley & Sons, 2008. Ch. 9 and 16.
- Aditya P. Mathur, "**Foundations of Software Testing**", Addison-Wesley Professional, 2008. Ch. 6 and 7.
- Y. Jia and M. Harman, "**An analysis and survey of the development of mutation testing**", IEEE Transactions on Software Engineering, 37(5): 649-678 (2011).