



# CISC/CMPE 327 Software Quality Assurance

Queen's University, 2019–fall

## Lecture #17 White Box Testing – Mutation

# White Box Testing

- Today we continue our look at **white box** testing methods with **mutation testing**
- Definition and role of mutation testing
  - What is a **mutation**?
  - What is a **mutant**?
  - How is mutation testing used?
- Mutation testing methods
  - **Value** mutations
  - **Decision** mutations
  - Other mutations

# Mutation Testing

- Mutation testing is a white box method for checking the **adequacy** of a test suite
- As you have already discovered, creating a test suite can be an expensive and time consuming effort
- No matter what test method is used, discovering if test suites are adequate to uncover faults is **itself** an even more difficult task
- Mutation testing offers an almost completely **automated** way to check the adequacy of a set of tests in uncovering faults in the software

# Mutation Testing

- How does it work?
  - To test the adequacy of a **test suite**, we run the software on the suite and fix problems until the software passes every test
  - We then **save the results** in a set of files to serve as the correct output to compare to
  - We then use **mutation** on the source code to create a set of **mutants** (this step is automated)
  - A **mutant** is a program that is slightly different from the original

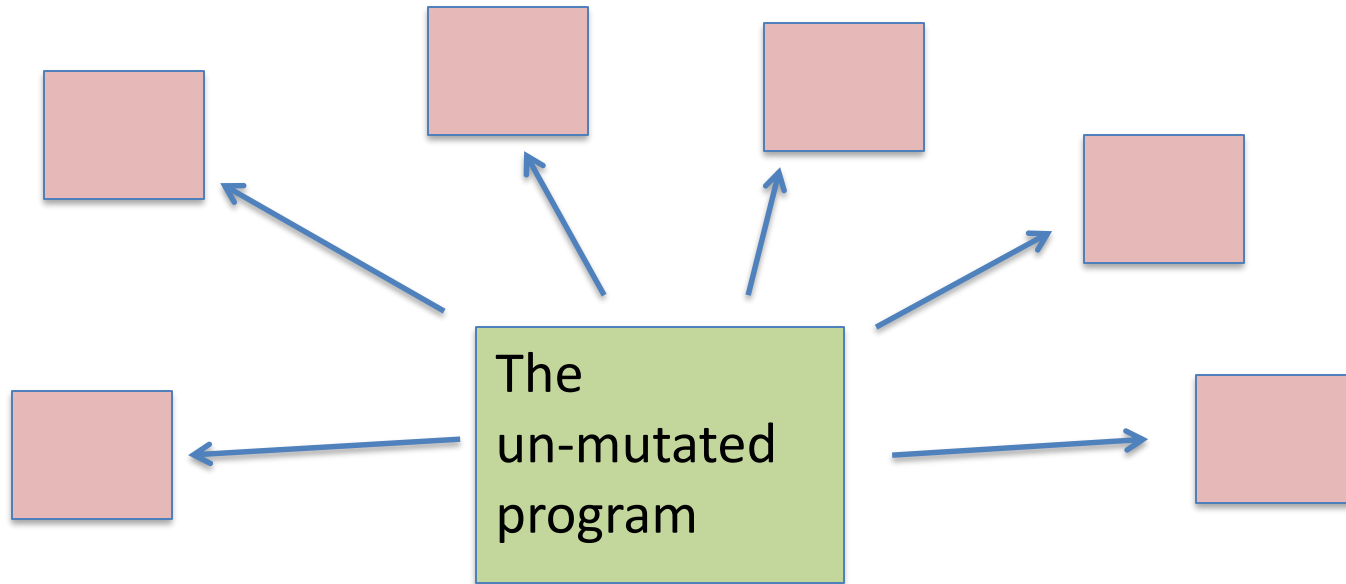
# Mutation Testing

- For each mutant, we run the test suite on the mutant and **compare the results** to the saved results from the original
- If the results differ, then the mutant has been **“killed”** (detected) by the test suite
- If the results do not differ, then the test suite is **inadequate** to detect the mutant, and a new test must be added to the suite to kill that mutant

# Systematic Mutation

- For mutation testing to be systematic, there must be (1) a **system** for creating mutants, (2) a **completion criterion** (knowing when you have enough mutants)
- The **system** is a set of kinds of **syntactic changes** to the program source, generally expected to cause errors
- Each mutant has exactly one change in it
- We are done when every possible single change **mutant** of the system has been generated and “killed”

# Visualizing Mutation



Each arrow represents **one small change**, such as replacing a constant 1 with 0, or exchanging addition for subtraction

*Radioactivity*

*It's in the air for you and me*

*Chain reaction and mutation*

*Contaminated population*

(Kraftwerk)

# Kinds of Systematic Mutation

- **Value** mutations (changing constants, subscripts, or parameters by adding or subtracting one, etc.)
- **Decision** mutations (inverting or otherwise modifying the sense of each decision condition in the program)
- **Statement** mutation (deleting or exchanging individual statements in the program)



# Why does this make sense?

- As we discussed weeks ago:  
*quality is not accidental*
- **Almost any change** to a high-quality program, or even a middling-quality program, will create a bug
- (Analogies: math, literature...)

# Example 1: Value Mutation

- **System:** Mutate the value of each **constant** (or more generally, each integer expression) in the program by adding or subtracting 1
- **Completion criterion:**  
One mutant for each constant in the program
- Note that there are many other possible **value mutations**:
  - **Constants** modified in some other way, e.g. off by -1
  - All **integer expressions** modified (not just constants) e.g.  $x$  changed to  $x+1$ , etc.
  - Floating-point: off by 0.00001, ...

# Example 1: Value Mutation

```
// calculate numbers less than x
// which are divisible by y
```

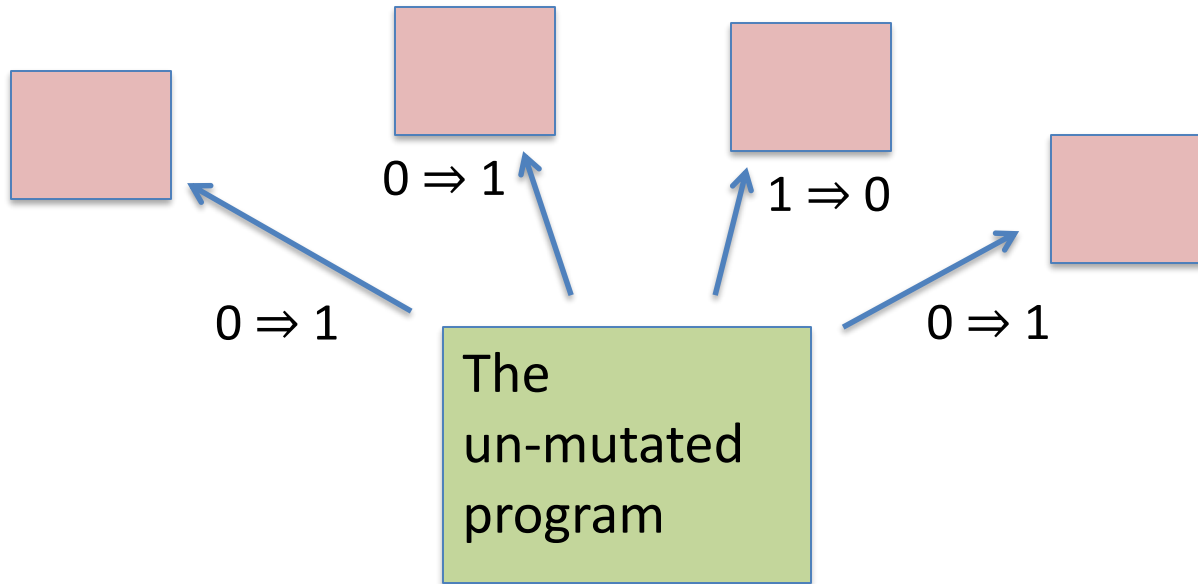
```
int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

Example test suite (statement coverage)

Test	x	y	output
T1	0	0	"y is 0"
T2	0	1	"x is 0"
T3	1	1	1

# Visualizing Mutation



Each arrow represents **one small change**, such as replacing a constant 1 with 0, or exchanging addition for subtraction

# Example 1: Value Mutation

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 1)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 1)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"x is 0"
T2	0	1	"x is 0"	"y is 0"
T3	1	1	1	"y is 0"

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	
T3	1	1	1	"x is 0"

# Example 1: Value Mutation

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 2; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

```
// calculate numbers less than x
//   which are divisible by y
```

```
int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 1)
            c.println (i);
    }
}
```

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

# Example 2: Decision Mutation

- **System:** Invert the sense of each **decision condition** in the program
  - e.g., change  $>$  to  $<$  (or  $<=$ ),  $==$  to  $!=$ , and so on
- **Completion criterion:**  
One mutant for each decision condition in the program

# Example 2: Decision Mutation

```
// calculate numbers less than x
// which are divisible by y
```

```
int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

Example test suite (statement coverage)

Test	x	y	output
T1	0	0	"y is 0"
T2	0	1	"x is 0"
T3	1	1	1



# Example 2: Decision Mutation

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y != 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}

// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x != 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"x is 0"
T2	0	1	"x is 0"	"y is 0"
T3	1	1	1	"y is 0"

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	
T3	1	1	1	"x is 0"

# Example 2: Decision Mutation

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i > x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y != 0)
            c.println (i);
    }
}
```

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

Test	x	y	output or original	output of mutant
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

# Example 3: Statement Mutation

- **System:** Delete each statement in the program
- **Completion criterion:**  
One mutant for each statement
- Many other possible **statement mutations**:
  - **Interchanging** adjacent statements
  - **Reordering** sequences of statements
  - **Doubling** statements
  - ...

# Example 3: Statement Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;
```

```
x = c.readInt ();
```

```
y = c.readInt ();
```

```
if (y == 0)
```

```
    c.println ("y is 0");
```

```
else if (x == 0)
```

```
    c.println ("x is 0");
```

```
else {
```

```
    for (int i = 1; i <= x; i++) {
```

```
        if (i % y == 0)
```

```
            c.println (i);
```

```
    }
```

```
}
```

Example test suite (statement coverage)

Test	x	y	output
T1	0	0	"y is 0"
T2	0	1	"x is 0"
T3	1	1	1

# Example 3: Statement Mutation

```
// calculate numbers less than x
//   which are divisible by y

int x, y;
x = c.readInt ();
y = c.readInt ();

if (y == 0)
    
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

Test	x	y	output of original	output of mutant
T1	0	0	"y is 0"	
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	1

- This time, we show only **one example** - you can make the rest!
- All statement mutants of this program turn out to be **"killed"** by our simple test set

# Determining Test Suite Adequacy

- If **D** is the number of **dead mutants** (program variations that were caught by our existing test suite), and **M** is the **total number of mutants**

$$\text{mutation adequacy score} = D / M$$

# Some Observations

- In practice, simple **statement coverage** will suffice to “**kill**” most kinds of mutants
- Thus they can detect most kinds of **accidental** faults that might be introduced into a working program
  - Quality is not accidental; this is why **coverage** tests are worth doing
- However, mutation **can** catch **missing test cases** even in coverage tests
- Since most projects use **primarily** black box techniques, automated mutation testing can be valuable in making test suites more effective

# Advantages and Disadvantages

- **Advantages**

- Provides a good check for **quality** of a test suite, however created
- Once “**baseline**” of correct results of a test suite has been checked, testing adequacy of the suite using mutation can be **automated**

- **Disadvantages**

- **Expensive** - generates a huge number of mutants, many really checking the same cases
- Detecting mutant **equivalence** is a big problem



# Summary

- **Mutation Testing**

- Mutation testing is a white box method for automatically checking test suites for completeness
- Mutations are simple, **syntactic** variants of programs that can be generated automatically
- Typical mutations are **value** mutations, **decision** mutations, **statement** mutations
- Mutation can find **missing test cases** in a test suite
- **Statement coverage** is a strong testing system, usually “kills” most kinds of mutants

# Summary

- References
  - Van Vliet ch. 13.6
- Next Time
  - Continuous testing methods