Mutation

▶ A small modification to a piece of code

▶ The use of word *mutation* is inspired by "biological mutation"

▶ Mutating is done by applying mutation operators

▶ **Mutation Operator**: a rule that specifies a <u>valid</u> syntactic modification

▶ **Mutant**: The modified (i.e., mutated) code after applying a mutation operator

Example

```
// # original code
int index = 0;
while(true) {
    index++;
    // control flow branching
    if (index == 10)
        break;
}
```

```
// # mutated code (a mutant)
int index = 0;
while (true) {
    index++;
    // mutating
    if (index <= 10)
        break;
}
```
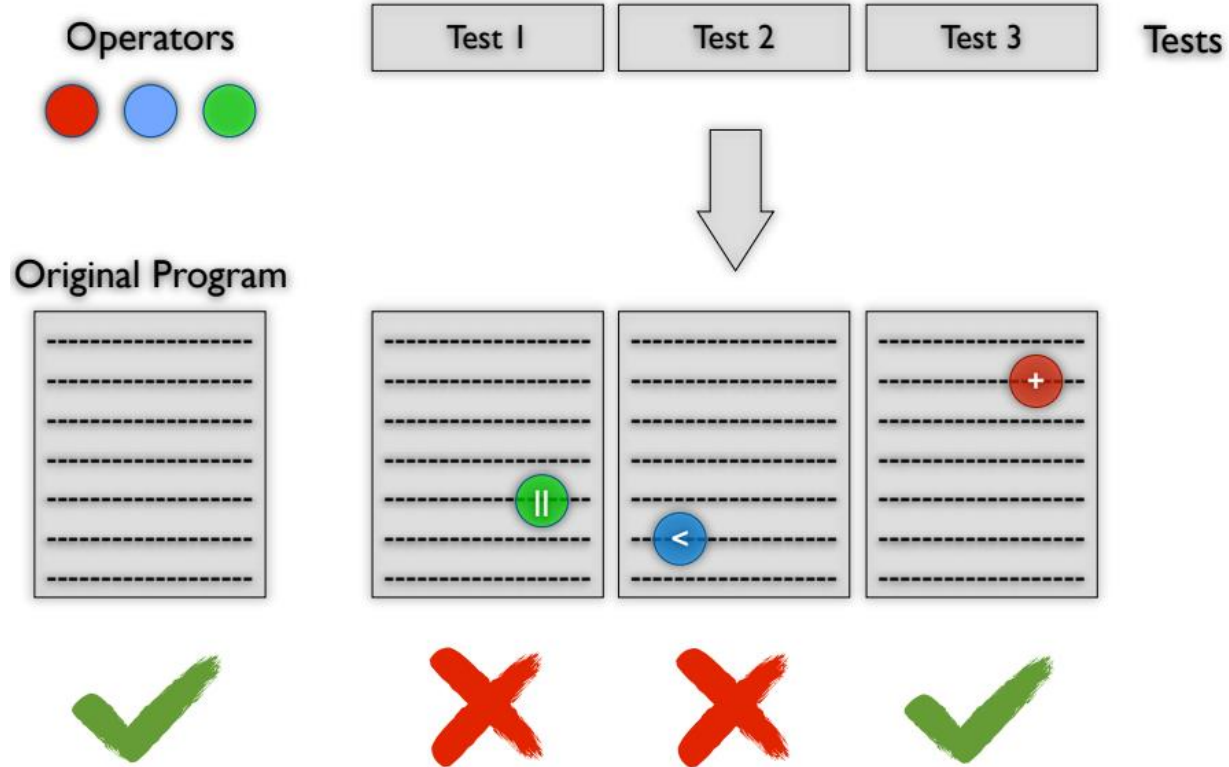
Mutation Testing
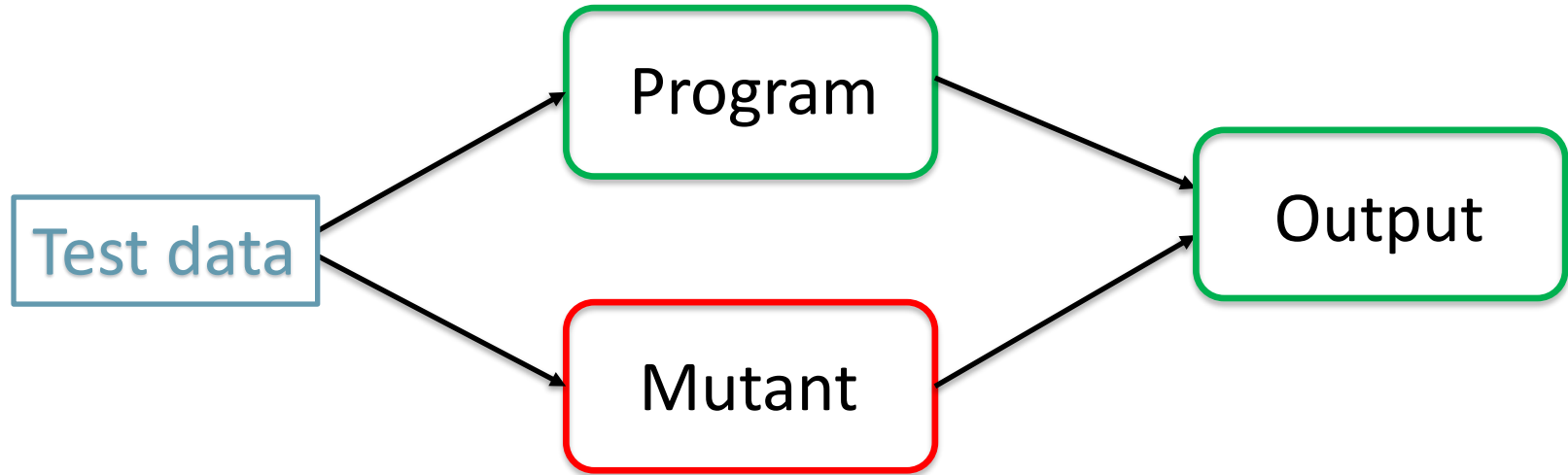
Mutation Testing

▶ 1. Mutating a piece of code to generate a set of mutants

▶ 2. "Kill" all the (or as many as possible) mutants

Assume **M** is the set of all mutants for a given piece of code **P**. Test **t** is said to "kill" **m** ∈ **M** if and only if the output of **t** on **P** is different from the output of **t** on **m**.

Mutation Testing

Example

```
// Program P
int min (int a, int b) {
    if (a < b) {
        return a;
    }
    return b;
}
```

```
// Mutant m
int min (int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}
```

Test t1: a = 10, b = 12 kills m

Test t2: a = 10, b = 10 does not kill m

Mutation Coverage

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m.

▶ Coverage in mutation equates to number of mutants killed
▶ MC level is also called mutation score, which is:

$$\frac{number\ of\ mutants\ killed}{Total\ number\ of\ mutants}$$

Mutation Operator Coverage (MOC)

Mutation Operator Coverage (MOC) : For each mutation operator, TR contains exactly one requirement, to create a mutant that is derived using the mutation operator.

Mutation Testing

► The number of test requirements for mutation depends on two things

  ❖ The syntax of the artifact being mutated
  ❖ The mutation operators

► Mutation testing is very difficult to apply by hand

► Mutation testing is very effective – considered the **"gold standard"** of testing

► Mutation testing is often used to evaluate other criteria

Mutation Testing

▶ Help the test engineer to strengthen the quality of the tests
  ❖ At least one test case in the test suite to kill mutant m

▶ Designing *effective* mutation operators is the key

▶ Mutation Operators are designed to:
  ❖ mimic typical programmer mistakes
  ❖ encourage the test engineer to write fault-revealing tests

Coupling Effect

▶ Mutation Testing Premise:

"Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors."

Mutation Testing

▶ If mutation operators are designed well, the resulting tests will be very powerful

▶ Different operators must be defined for different programming languages and different goals

▶ Testers can keep adding tests until all mutants have been killed
- ❖ *Dead mutant* : A test case has killed it
- ❖ *Stillborn mutant* : Syntactically illegal
- ❖ *Trivial mutant* : Almost every test can kill it
- ❖ *Equivalent mutant* : No test can kill it (same behavior as original)

Equivalent Mutant

▶ A Mutant that is functionally equivalent to the original code

▶ Thus, no test case can kill an equivalent mutant → infeasible test requirement

▶ In general, an *undecidable* problem to identify an equivalent mutant

```
// # original code
int index = 0;
while(true) {
    index++;
    // control flow branching
    if (index == 10)
        break;
}
```

```
// # an equivalent mutant
int index = 0;
while (true) {
    index++;
    // mutating
    if (index >= 10)
        break;
}
```

13

RIPR Model and Mutation Testing

- The RIPR model discussed earlier in class:

  - *Reachability* : The test causes the faulty statement to be reached (in mutation – the mutated statement)

  - *Infection* : The test causes the faulty statement to result in an incorrect state

  - *Propagation* : The incorrect state propagates to incorrect output

  - *Revealability* : The tester must observe part of the incorrect output

- The RIPR model leads to two variants of mutation coverage …

Strong Mutation Coverage

Strong Mutation Coverage (SMC) : For each $m \in M$, TR contains exactly one requirement, to strongly kill m.

Weak vs. Strong Kill

▶ 1) Strongly Killing Mutants:

❖ Given a mutant $m \in M$ for a program P and a test t, t is said to *strongly kill* m if and only if the output of t on P is different from the output of t on m

▶ 2) Weakly Killing Mutants:

❖ Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to *weakly kill m* if and only if the state of the execution of P on t is different from the state of the execution of m on t immediately after l

● *W*eakly killing satisfies reachability and infection, but not propagation

Weak Mutation

> Weak Mutation Coverage (WMC) : For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

- "Weak mutation" is so named because it is **easier to kill** mutants under this assumption
  - Weak mutation also requires **less analysis**
- A few mutants can be killed under weak mutation but not under strong mutation **(no propagation)**
- *"Studies have found that test sets that weakly kill all mutants also strongly kill most mutants"*

Strong vs. Weak Mutation

```
int gcd(int x, int y) {        int gcd(int x, int y) {
  int tmp;                       int tmp;
  while(y != 0) {                while(y != 0) {
    tmp = x % y;       weak mutation    tmp = x * y;
    x = y;                         x = y;
    y = tmp;                       y = tmp;
  }                             }
  return x;         strong mutation    return x;
}                             }
```

Mutation Testing Process

Prog → Input test method

Create mutants → Run equivalence detector → Generate test cases → Run T on P

Run T on mutants

Automated steps

Define threshold

Eliminate ineffective TCs

Threshold reached? — *no* → Generate test cases

Threshold reached? — *yes* → P (T) correct?

P (T) correct? — *no* → Fix P

Fix P → Input test method
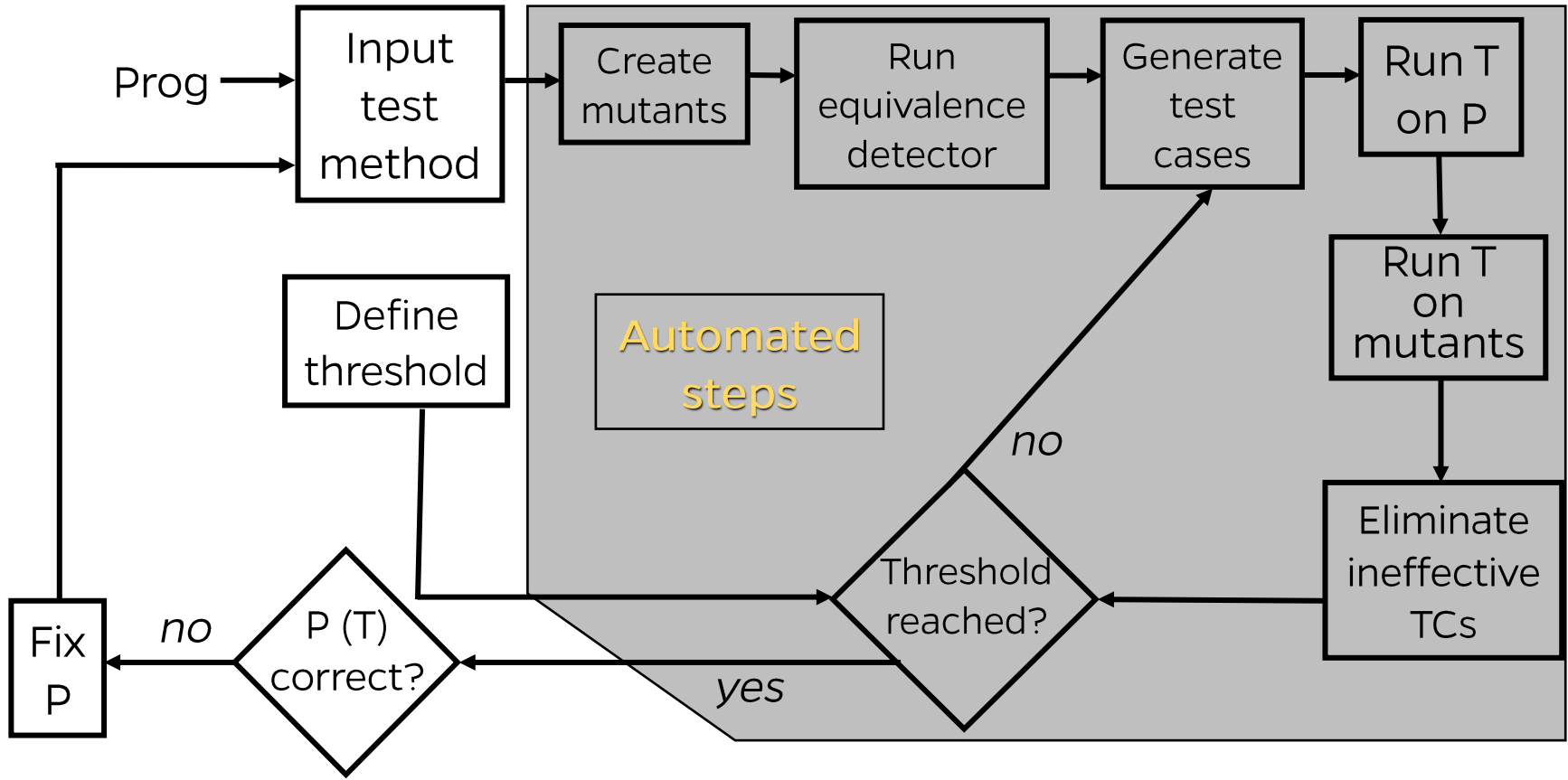
Unit-Level Mutation Operators for Java

1. ABS –– Absolute Value Insertion
2. AOR –– Arithmetic Operator Replacement
3. ROR –– Relational Operator Replacement
4. COR –– Conditional Operator Replacement
5. SOR –– Shift Operator Replacement
6. BOR –– Bitwise Operator Replacement
7. ASR –– Assignment Operator Replacement
8. UOI –– Unary Operator Insertion
9. UOD –– Unary Operator Deletion
10. SVR –– Scalar Variable Replacement
11. BSR –– Bomb Statement Replacement

Absolute Value Insertion

ABS –– Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

Examples:
         a = m * (o + p);
△1   a = abs (m * (o + p));
△2   a = m * abs ((o + p));
△3   a = failOnZero (m * (o + p));

Arithmetic Operator Replacement

AOR –– Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, - , * , ∕ , and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

Examples:
    a = m * (o + p);
△1 a = m + (o + p);
△2 a = m * (o * p);
△3 a = m *leftOp* (o + p);

Relational Operator Replacement

ROR –– Relational Operator Replacement:

Each occurrence of one of the relational operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) is replaced by each of the other operators and by *falseOp* and *trueOp*.

Examples:
        if (X <= Y)
△1   if (X > Y)
△2   if (X < Y)
△3   if (X *falseOp* Y) // always returns false

Conditional Operator Replacement

OR –– Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by *falseOp, trueOp, leftOp,* and *rightOp.*

Examples:
    if (X <= Y && a > 0)
△1 if (X <= Y || a > 0)
△2 if (X <= Y *leftOp* a > 0) // returns result of left clause

Shift Operator Replacement

SOR –– Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.

Examples:
    byte b = (byte) 16;
    b = b >> 2;
△1   b = b << 2;
△2   b = b *leftOp* 2; // result is b

Logical Operator Replacement

6. BOR –– Bitwise Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Examples:
    int a = 60;   int b = 13;
    int c = a & b;
△1 int c = a | b;
△2 int c = a *rightOp* b; // result is b

Assignment Operator Replacement

ASR –– Assignment Operator Replacement:

Each occurrence of one of the assignment operators (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) is replaced by each of the other operators.

Examples:
a = m * (o + p);
△1   a += m * (o + p);
△2   a *= m * (o + p);

Unary Operator Insertion

UOI –– Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

Examples:

    a = m * (o + p);
△1   a = m * -(o + p);
△2   a = -(m * (o + p));

Unary Operator Deletion

UOD –– Unary Operator Deletion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Examples:
        if !(X <= Y && !Z)
△1   if (X > Y && !Z)
△2   if !(X < Y && Z)

Scalar Variable Replacement

SVR –– Scalar Variable Replacement:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

Examples:

a = m * (o + p);

△ 1 a = o * (o + p);

△ 2 a = m * (m + p);

△ 3 a = m * (o + o);

△ 4 p = m * (o + p);

Bomb Statement Replacement

BSR –– Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Example:
    a = m * (o + p);
△1   *Bomb*() // Raises exception when reached

Example: Forcing Branch Coverage using BOMB

Program	Mutant 1	Mutant 2

```
int min(int a, int b) {
    int minVal;
    minVal = a;
    if (b < a) {
        minVal = b;
    }
    return minVal;
}
```

```
int min(int a, int b) {
    int minVal;
    minVal = a;
    if (b < a) {
        minVal = b;
        BOMB(); // raises an exception
                    to kill this mutant
    }
    return minVal;
}
```

```
int min(int a, int b) {
    int minVal;
    minVal = a;
    if (b < a) {
        minVal = b;
    }
    BOMB(); // raises an exception
                to kill this mutant
    return minVal;
}
```

Questions

▶ Should more than one mutation operator be applied at the same time to produce a mutant?

▶ Should we try each and every possible mutation operator on any program?

Ways to Improve Mutation Testing Efficiency

▶ Parallelize

▶ Mutate Bytecode instead of Source Code
  ❖ no need to recompile

▶ Use Coverage
  ❖ measure coverage before doing mutation analysis

▶ Selective Mutation:
  ❖ only use a subset of mutation operators

▶ Strong vs. Weak Killing
  ❖ use weak killing

Use Coverage

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
    return 4; // invalid
  }
  if (!(a + b > c && a + c > b && b + c > a)) {
    return 4; // invalid
  }
  if (a == b && b == c) {
    return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
    return 2; // isosceles
  }
  return 3; // scalene
}
```

(0, 0, 0)
(2, 2, 2)
(1, 1, 3)
(2, 2, 3)
(2, 3, 4)
(4, 3, 2)
(2, 3, 2)
(1, 1, 1)

only these test cases execute mutants in this line

If we mutate the last if expression, then there is no point in executing all of the test cases against the mutants derived from the expression. Only some of the test cases will actually execute the mutation. If a test case does not execute the mutation, then there is no way it could kill it. Therefore, before mutation analysis we determine statement coverage for each of the test cases, and during mutation analysis only execute those test cases for a mutant that actually reach the mutation.

Mutation Testing

▶ Mutation Testing is expensive (lots of mutants):

  ❖ sample from the set of all mutants

  ❖ selectively use mutation operators that are <u>effective</u>

*** If tests that are created to kill mutants produced by mutation operator o_1 also kill mutants produced by mutation operator o_2, we say that o_1 is more effective than o_2 ***

Mutation Testing

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o1, o2, …\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an *effective* set of mutation operators

▶ *** *"Researchers have concluded that the collection of operators that <u>insert unary operators</u> and that <u>modify unary and binary</u> operators are effective."* ***

Integration Mutation Testing

► Integration Testing: Testing multiple units together
  ❖ Specifically, testing connections among units
  ❖ In Java, testing the way classes, classes, packages, and modules are connected
    ● example: which methods of class **A** calls which methods of class **B** and vice versa.

► Integration testing is often based on the existing **couplings** – the explicit and implicit relationships among software components

Integration Mutation Testing

► Faults related to integrations often depend on a _mismatch of assumptions_

- ❖ Callee thought a list was sorted, caller did not
- ❖ Callee thought all fields were initialized, caller only initialized some of the fields
- ❖ Caller sent values in kilometers, callee thought they were miles
- ❖ etc.

Integration Mutation Operators

► In general, there are four kinds:
- ❖ change a calling method by modifying values that are sent to a called method
- ❖ change a calling method by modifying the call
- ❖ change a called method by modifying values that enter and leave a method
  - ● includes parameters as well as variables from higher scopes (class level, package, public, etc.)
- ❖ change a called method by modifying return statements from the method

Integration Parameter Variable Replacement

IPVR: Each parameter in a method call is replaced by each other variable in the scope of the method call that is of compatible type

MyClass a, b;

. . .

callMethod (a);

Δ callMethod (b);

IPEX – Integration Parameter Exchange

IPEX: Each parameter in a method call is exchanged with each parameter of compatible types in that method call

max (a, b);
Δ max (b, a);

IUOI –– Integration Unary Operator Insertion

IUOI: Each expression in a method call is modified by inserting all possible unary operators in front and behind it

callMethod (a);

Δ callMethod (a++);
Δ callMethod (++a);
Δ callMethod (a--);
Δ callMethod (--a);

Integration Method Call Deletion

IMCD: Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value

X = Max (a, b);
Δ X = new Integer (0);

Integration Return Expression Modification

IREM: Each expression in each return statement in a method is modified by applying the UOI and AOR operators

```
int myMethod () {
  return a + b;
∆  return ++a + b;
∆  return a – b;
}
```

Object-Oriented Mutation

▶ So far, we only mutated method bodies

▶ Class level mutation operators can be utilized too

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

```
public class test {
  // ..
  public void do() {
    // ...
  }
}
```

Object-Oriented Mutation Operators

AMC - Access Modifier Change

HVD - Hiding Variable Deletion

HVI - Hiding Variable Insertion

OMD - Overriding Method Deletion

OMM - Overridden Method Moving

OMR - Overridden Method Rename

SKR - Super Keyword Deletion

PCD - Parent Constructor Deletion

ATC - Actual Type Change

DTC - Declared Type Change

PTC - Parameter Type Change

RTC - Reference Type Change

OMC - Overloading Method Change

OMD - Overloading Method Deletion

AOC - Argument Order Change

ANC - Argument Number Change

TKD - this Keyword Deletion

SMV - Static Modifier Change

VID - Variable Initialization Deletion

DCD - Default Constructor 2

Summary

► Mutation testing is:
  - ❖ widely considered the **strongest test criterion**
  - ❖ a **high-end type of testing** i.e., more effective, also more expensive
  - ❖ expensive, i.e., by far the most test requirements
    - ● generate mutants + running all test cases of the test suite on each generated mutant
  - ❖ *Mutation Analysis* is assessing the quality of a test suite
  - ❖ *Mutation Testing* is used to improve the test suite quality by leveraging mutation analysis
  - ❖ Can be applied to any syntax, hence aka syntax-based testing:
    - ● have been applied to formal specification languages such as SVM
    - ● have been applied to markup languages such as XML, HTML etc

Tool Support

▶ **Java** : muJava, PIT, Javalanche, etc.

▶ **C++**: MuCPP, Mutate++, etc.

▶ **Python**: Cosmic Ray, mutmut, etc.

▶ **Ruby**: Heckle

▶ **PHP**: Infection PHP, Humbug

▶ **C#**: Nester, VisualMutator

▶ etc.

Relevant Reads and Resources

► Recommended Text:

❖ Introduction to Software Testing, 2nd Edition: ch9