

# Introduction to Software Testing

## Chapter 5.3

### Integration and Object-Oriented Testing

Paul Ammann & Jeff Offutt

[www.introsoftwaretesting.com](http://www.introsoftwaretesting.com)

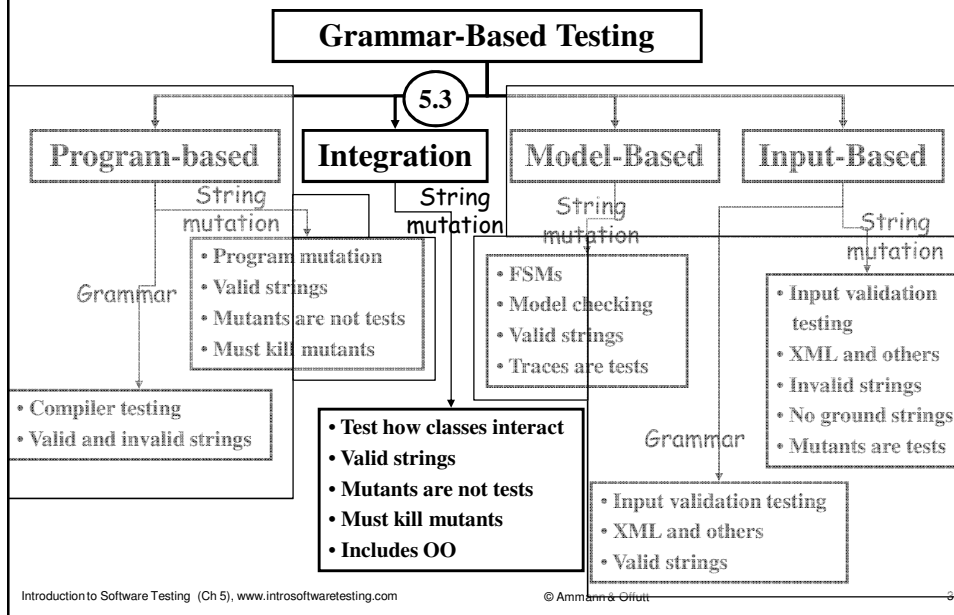
## Integration and Object-Oriented Testing

### Integration Testing

#### Testing connections among separate program units

- In Java, testing the way classes, packages and components are connected
  - “*Component*” is used as a generic term
- This tests features that are unique to object-oriented programming languages
  - inheritance, polymorphism and dynamic binding
- Integration testing is often based on couplings – the explicit and implicit relationships among software components

# Instantiating Grammar-Based Testing



## Grammar Integration Testing (5.3.1)

**There is no known use of grammar testing at the integration level**

## Integration Mutation (5.3.2)

- **Faults related to component integration 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
- **Integration mutation focuses on mutating the connections between components**
  - Sometimes called “*interface mutation*”
  - Both caller and callee methods are considered

## Four Types of Mutation Operators

- 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

## Five Integration Mutation Operators

### 1. *IPVR* — *Integration Parameter Variable Replacement*

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

- **This operator replaces primitive type variables as well as objects.**

### 2. *IUOI* — *Integration Unary Operator Insertion*

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

- **The unary operators vary by language and type**

### 3. *IPEX* — *Integration Parameter Exchange*

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

- **max (a, b) is mutated to max (b, a)**

## Five Integration Mutation Operators (2)

### 4. *IMCD* — *Integration Method Call Deletion*

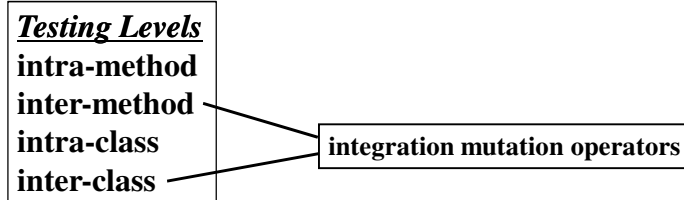
Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.

- **Method calls that return objects are replaced with calls to “new ()”**

### 5. *IREM* — *Integration Return Expression Modification*

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

## Object-Oriented Mutation



- These five operators can be applied to non-OO languages
  - C, Pascal, Ada, Fortran, ...
- They do **not support object oriented features**
  - Inheritance, polymorphism, dynamic binding
- Two other language features that are often lumped with OO features are information hiding (encapsulation) and overloading
- Even experienced programmers often get encapsulation and access control wrong

## Encapsulation, Information Hiding and Access Control

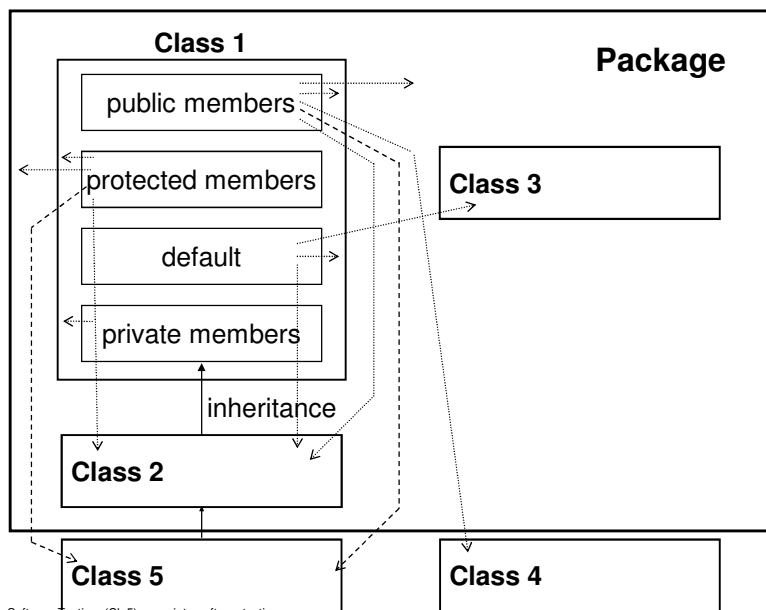
- **Encapsulation** : An abstraction mechanism to implement **information hiding**, which is a design technique that attempts to protect parts of the design from parts of the implementation
  - Objects can restrict access to their member variables and methods
- **Java provides four access levels (C++ & C# are similar)**
  - private
  - protected
  - public
  - default (also called package)
- **Often not used correctly or understood, especially for programmers who are not well educated in design**

## Access Control in Java

Specifier	Same class	Same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

- Most class variables should be private
- Public variables should seldom be used
- Protected variables are particularly dangerous – future programmers can accidentally override (by using the same name) or accidentally use (by mis-typing a similar name)
  - They should be called “unprotected”

## Access Control in Java (2)



## Object-Oriented Language Features (Java)

- **Method overriding**  
Allows a method in a subclass to have the same name, arguments and result type as a method in its parent
- **Variable hiding**  
Achieved by defining a variable in a child class that has the same name and type of an inherited variable
- **Class constructors**  
Not inherited in the same way other methods are – must be explicitly called
- **Each object has ...**
  - a declared type : *Parent P*;
  - an actual type : *P = new Child ()*; or assignment : *P = Pold*;
  - Declared and actual types allow uses of the same name to reference different variables with different types

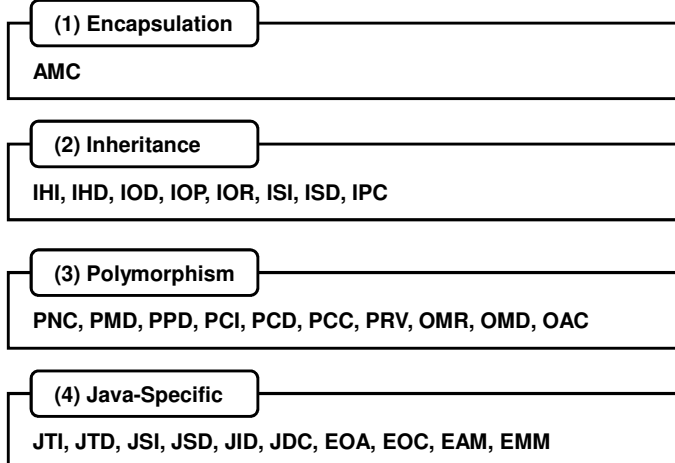
## OO Language Feature Terms

- **Polymorphic attribute**
  - An object reference that can take on various types
  - Type the object reference takes on during execution can change
- **Polymorphic method**
  - Can accept parameters of different types because it has a parameter that is declared of type Object
- **Overloading**
  - Using the same name for different constructors or methods in the same class
- **Overriding**
  - A child class declares an object or method with a name that is already declared in an ancestor class
  - Easily confused with overloading because the two mechanisms have similar names and semantics
  - Overloading is in the same class, overriding is between a class and a descendant

## More OO Language Feature Terms

- **Members associated with a class are called class or instance variables and methods**
  - **Static methods can operate only on static variables; not instance variables**
  - **Instance variables are declared at the class level and are available to objects**
- **29 object-oriented mutation operators defined for Java – muJava**
- **Broken into 4 general categories**

## Class Mutation Operators for Java



## OO Mutation Operators—*Encapsulation*

### 1. AMC — *Access Modifier Change*

The access level for each instance variable and method is changed to other access levels.

## OO Mutation Operators—*Example*

### 1. AMC – *Access Modifier Change*

	<b>point</b>
	private int x;
Δ1	public int x;
Δ2	protected int x;
Δ3	int x;

## OO Mutation Operators—*Inheritance*

### 2. IHI — *Hiding Variable Insertion*

A declaration is added to hide the declaration of each variable declared in an ancestor.

### 3. IHD — *Hiding Variable Deletion*

Each declaration of an overriding or hiding variable is deleted.

### 4. IOD — *Overriding Method Deletion*

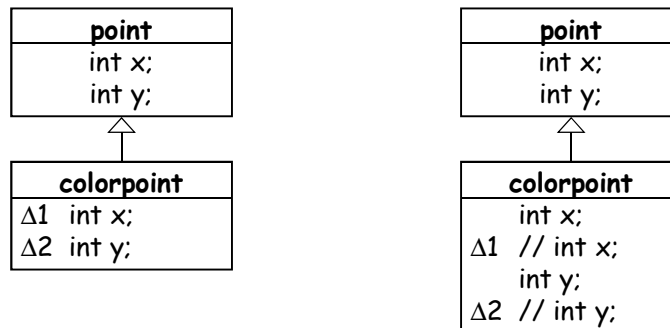
Each entire declaration of an overriding method is deleted.

### 5. IOP — *Overriding Method Calling Position Change*

Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.

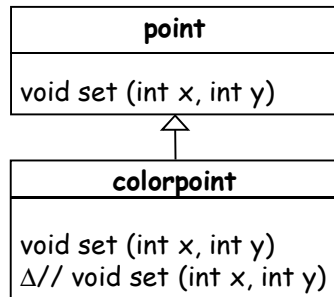
## OO Mutation Operators—*Example*

### 2. IHI – *Hiding Variable Insertion*    3. IHD – *Hiding Variable Deletion*

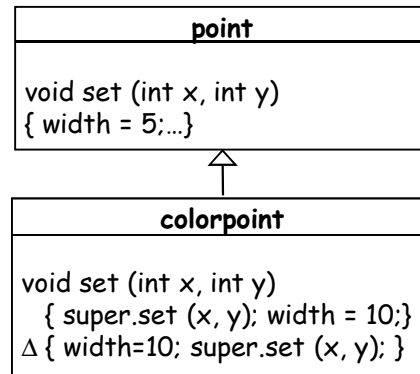


## OO Mutation Operators—*Example*

### 4. IOD – Overriding Method Deletion



### 5. IOP – Overriding Method Calling Position Change



## OO Mutation Operators—*Inheritance*

### 6. IOR — *Overriding Method Rename*

Renames the parent's versions of methods that are overridden in a subclass so that the overriding does not affect the parent's method.

### 7. ISI — *Super Keyword Insertion*

Insert the keyword **super** in front of each overriding method or variable.

### 8. ISD — *Super Keyword Deletion*

Delete each occurrence of the **super** keyword.

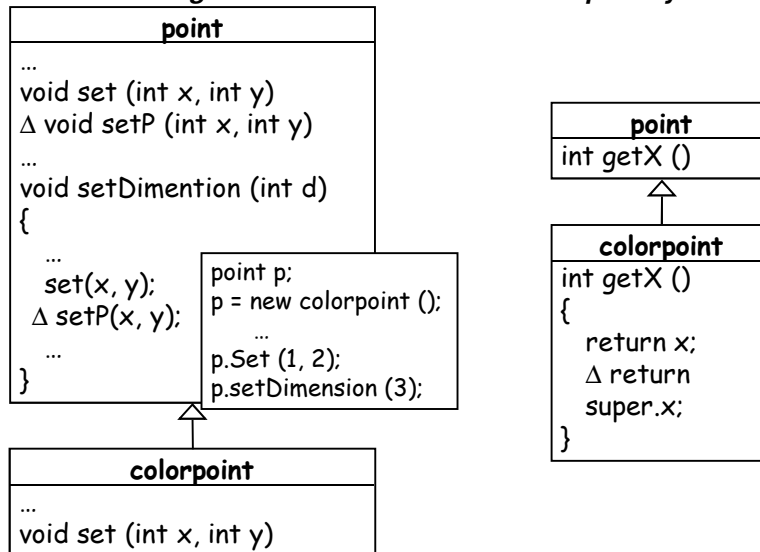
### 9. IPC — *Explicit Call of a Parent's Constructor Deletion*

Each call to a **super** constructor is deleted.

## OO Mutation Operators—Example

6. IOR – Overriding Method Rename

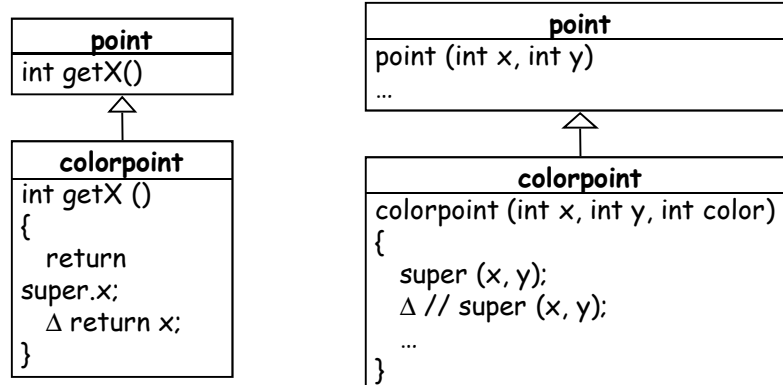
7. ISI – Super keyword Insertion



## OO Mutation Operators—Example

8. ISD – Super keyword Deletion

9. IPC – Explicit Call of a Parent's Constructor Deletion



## OO Mutation Operators—*Polymorphism*

### 10. PNC — *New Method Call With Child Class Type*

An object's actual type is changed to a child of the original actual type in the `new()` statement.

### 11. PMD — *Member Variable Declaration With Parent Class Type*

An object's declared type is changed to the parent of the original actual type in the declaration statement.

### 12. PPD — *Parameter Variable Declaration With Child Class Type*

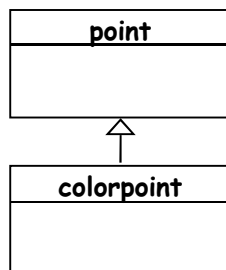
Changes the declared type of a parameter object reference to be the parent of its original declared type.

### 13. PCI — *Type Cast Operator Insertion*

Change the actual type of an object reference to the parent or to the child of the declared type.

## OO Mutation Operators—*Example*

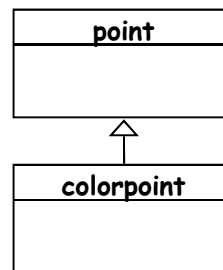
### 10. PNC – *New Method Call with Child Class Type*



```

point p;
p = new point ();
Δ p = new colorpoint ();
  
```

### 11. PMD – *Member Variable Declaration with Parent Class Type*

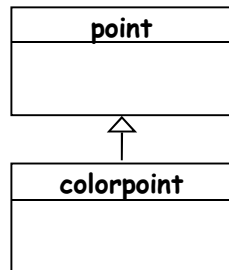


```

point p;
Δ p = new point ();
p = new colorpoint ();
  
```

## OO Mutation Operators—*Example*

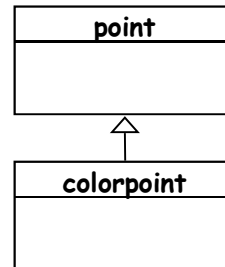
### 12. PPD – Parameter Variable Declaration with Child Class Type



```

boolean equals (point p)
{ ... }
Δ boolean equals (colorpoint p)
{ ... }
  
```

### 13. PCI – Type Case Operator Insertion



```

colorpoint pc;
point p = cp;
p.getX();
Δ ((colorpoint) p).getX();
  
```

## OO Mutation Operators—*Polymorphism*

### 14. PCD — Type Cast Operator Deletion

Delete a type casting operator

### 15. PCC — Cast Type Change

Change the type to which an object reference is being cast

### 16. PRV — Reference Assignment With Other Comparable Variable

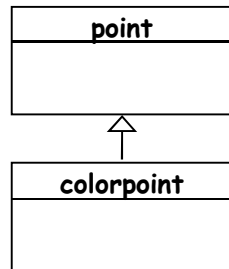
Changes operands of reference assignments to be assigned to objects of subclasses.

### 17. OMR — Overloading Method Contents Replace

For each pair of methods that have the same name, the bodies are interchanged.

## OO Mutation Operators—*Example*

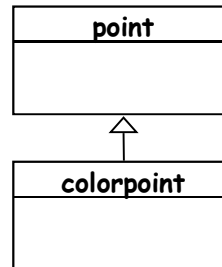
### 14. PCD – Type Case Operator Deletion



```

colorpoint cp;
point p = cp;
((colorpoint) p).getX();
Δ p.getX();
    
```

### 15. PCC – Cast Type Change

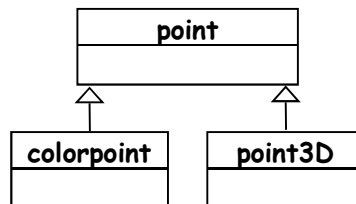


```

((point) p).getX();
Δ ((colorpoint) p).getX();
    
```

## OO Mutation Operators—*Example*

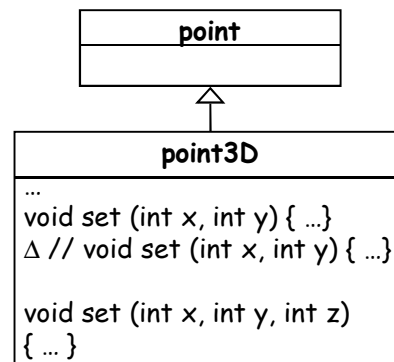
### 16. PRV – Reference Assignment with Other Comparable Variable



```

point p;
colorpoint cp = new colorpoint(0, 0);
point3D p3d = new point3D(0, 0, 0);
p = cp;
Δ p = p3d;
    
```

### 17. OMR – Overloading Method Contents Replace



## OO Mutation Operators—*Polymorphism*

### 18. OMD — *Overloading Method Deletion*

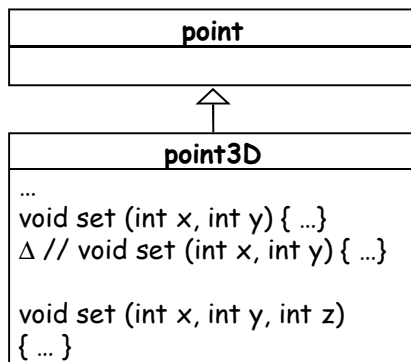
Each overloaded method declaration is deleted, one at a time.

### 19. OAC — *Arguments Of Overloading Method Call Change*

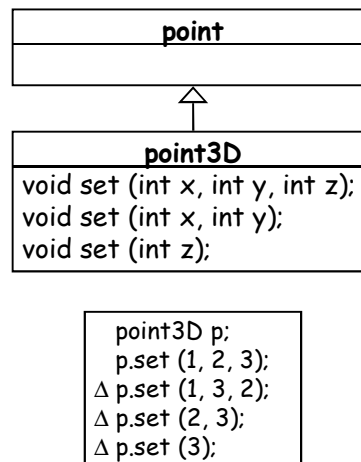
Changes the number and order of arguments in method invocations, but only if there is an overloading method that can accept the new argument list.

## OO Mutation Operators—*Example*

### 18. OMD – *Overloading Method Deletion*



### 19. OAC – *Arguments of Overloading Method Call Change*



## OO Mutation Operators—*Language Specific*

### 20. JTI — *This Keyword Insertion*

The keyword **this** is added when appropriate.

### 21. JTD — *This Keyword Deletion*

Each occurrence of the keyword **this** is deleted.

### 22. JSI — *Static Modifier Insertion*

The **static** modifier is added to instance variables.

### 23. JSD — *Static Modifier Deletion*

Each instance of the **static** modifier is removed.

## OO Mutation Operators—*Example*

### 20. JTI – *This Keyword Insertion*

### 21. JTD – *This Keyword Deletion*

```
point
...
void set (int x, int y)
{
  this.x = x;
  Δ this.x = this.x;

  this.y = y;
  Δ this.y = this.y;
}
...
```

```
point
...
void set (int x, int y)
{
  this.x = x;
  Δ x = x;
  this.y = y;
  Δ y = y;
}
...
```

## OO Mutation Operators—*Example*

22. JSI – *Static Modifier Insertion*

23. JSD – *Static Modifier Deletion*

```
point
public int x = 0;
Δ public static int x = 0;
...
```

```
point
public static int x = 0;
Δ public int x = 0;
...
```

## OO Mutation Operators—*Language Specific*

24. JID — *Member Variable Initialization Deletion*

Remove initialization of each member variable.

25. JDC — *Java-supported Default Constructor Creation*

Remove the implemented default constructor, forcing Java to create a default constructor.

26. EOA — *Reference Assignment and Content Assignment Replacement*

Replaces the right hand side of an object assignment with an assignment of a clone of the object.

27. EOC — *Reference Comparison and Content Comparison Replacement*

Replaces a comparison of two objects using the "==" operator with the "equals()" method.

## OO Mutation Operators—*Example*

### 24. JID – Member Variable Initialization Deletion

```
point
-----
int x = 0;
Δ int x;
...
```

### 25. JDC – Java-supported Default Constructor Creation

```
point
-----
point() { ... }
Δ // point() { ... }
...
```

## OO Mutation Operators—*Example*

### 26. EOA – Reference Assignment and Content Assignment Replacement

```
point p1, p2;
p1 = new point (1, 2);
p2 = p1;
Δ p2 = p1.clone();
```

### 27. EOC – Reference Comparison and Content Comparison Replacement

```
point p1 = new point (1, 2);
point p2 = new point (1, 2);
...
if (p1 == p2)
Δ if (p1.equals (p2))
{
...
}
```

## OO Mutation Operators—*Language Specific*

### 28. EAM — *Accessor Method Change*

Replaces calls to “**getter**” methods ( `getX()` ) with calls to other compatible getter methods ( `getY()` ).

### 29. EMM — *Modifier Method Change*

Replaces calls to “**setter**” methods ( `setX()` ) with calls to other compatible getter methods ( `setY()` ).

## OO Mutation Operators—*Example*

### 28. EAM – *Accessor Method Change*

```
point
...
public int getX () { ... }
public int getY () { ... }
public void setX (int x)
{ ... }
public void setY (int y)
{ ... }
...
```

```
point p1 = new point (1, 2);
...
p1.getX();
Δ p1.getY();
```

### 29. EMM – *Modifier Method Change*

```
point
...
public int getX () { ... }
public int getY () { ... }
public void setX (int x)
{ ... }
public void setY (int y)
{ ... }
...
```

```
point p1 = new point (1, 2);
...
p1.setX(4);
Δ p1.setY(4);
```

## Integration Mutation Summary

- **Integration testing often looks at couplings**
- **We have not used grammar testing at the integration level**
- **Mutation testing modifies callers and callees**
- **OO mutation focuses on inheritance, polymorphism, dynamic binding, information hiding and overloading**
  - The access levels make it easy to make mistakes in OO software
- **muJava is an educational / research tool for mutation testing of Java programs**
  - <http://ise.gmu.edu/~offutt/mujava/>