# Software testing

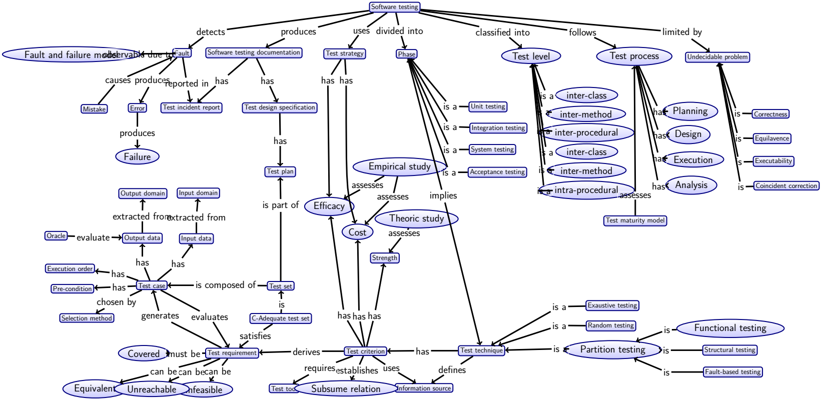
## MuJava

Marco Aurélio Graciotto Silva[1],
Ellen Francine Barbosa[1],
José Carlos Maldonado[1]

[1]University of São Paulo (USP)
São Carlos, SP, Brazil

February 2011

# Software testing



# muJava

## muJava

muJava (*μJava*) is a mutation tool for Java programs that automatically generates mutants using mutation analysis and interface mutation criteria.

## General information

- It was developed by Yu Seung Ma, Yong Rae Kwon and Jeff Offutt.
- Main page for muJava:
  - `http://cs.gmu.edu/~offutt/mujava/`
- A Eclipse plugin is also available (developed by Laurie Willians and Ben Smith):
  - `http://muclipse.sourceforge.net/`
- Compatible with Java 1.5 and 1.6 (but it still does not support generics).

## Method level operators

-

## Class level operators

- Set of 24 operators that are specialized to object-oriented faults.

## Limitations

### Limitations

- It does not implement any automatic equivalent mutant detection technique.
    - Equivalents mutants must be identified by hand.

# Method operators

## Method operators

Method-level mutation operators apply simple syntactical modifications to the methods of the classes of the application under testing.

## Design rationale

- Method operators were designed using the selective approach.
  - The selection results found that operand and statement modifications have poor effectiveness.
  - Thus, only operators that replace, delete, or insert elements in expressions were selected.

# Method operators

## Types of expression operators

- Arithmetic operator.
- Relational operator.
- Conditional operator.
- Shift operator.
- Logical operator.
- Assignments.

# Method operators

## Operators

- For arithmetic, conditional and logical operators, it is defined replacement, insertion, and deletion mutation operators.
    - For arithmetic operators, more operators are defined according to the type and number of operands.
- For relational, shift, and assignment operators, it is defined replacement operators.

# Class operators

## Class operators

Class-level mutation operators apply simple syntactical modifications to the classes of the application under testing.

## Affected elements

- Based on language features related to object orientation, four groups of operatores are defined:
    - Encapsulation.
    - Inheritance.
    - Polymorphism.
    - Java-specific features.

# Class operators
## Encapsulation

### Rationale

- It exploits the following facts:
  - the semantics of the various access levels are often poorly understood,
  - the access for variables and methods is often not considered during design.
- Although poor access definition do not always cause faults, it can lead to faulty behaviour when the class is integrated with other classes.

### Operators

- **AMC** (Access modifier change): it changes the access level for instance variables and methods.

# Class operators

## Inheritance

### Rationale

- Incorrect use of inheritance can lead to faults (variable shadowing, method overriding, parent access and constructors).

### Operators

- Variable shadowing:
  - Hiding variable deletion (**IHD**) and insertion (**IHI**).
- Method overriding:
  - Overriding method deletion (**IOD**), change in position (**IOP**), and renaming (**IOR**).
- Parent access:
  - `super` keyword insertion (**ISI**) and deletion (**ISD**).
- Constructor:
  - Explict call to a parent's constructor deletion (**IPC**).

# Class operators
## Polymorphism

### Rationale

- Polymorphism allows the late binding of types to the object that will be actually accessed.

### Operators

- Create a new object using a child type (**PNC**).
- Declare a variable using the parent class type (**PMD**).
- Declare parameter using the parent class type (**PPD**).
- Type cast operator insertion (**PCI**), delection (**PCD**), and change (**PCC**).
- Reference assignment with other compatible type (**PRV**).
- Overloading method contents modification (**OMR**), and deletion (**OMD**).
- Overloading argument change (**OAC**).

# Class operators
## Java-specific features

### Types of expression operators

- Some mistakes are not due to intrinsic object-orientation features, but of Java-specific ones.

### Operators

- `this` keyword insertion (**JTI**) and deletion **JTD**).
- `static` keyword insertion (**JSI**) and deletion (**JSD**).
- Member variable initialization deletion (**JID**).
- Java-supported default constructor creation (**JDC**).
- Reference assignment and content assignment replacement (**EOA**).
- Reference comparison and content comparison replacement (**EOC**).
- Accessor method change (**EAM**).
- Modifierr method change (**EMM**).

# muJava
## Installation

### Installation

1. Save muJava to a directory (for example, `/opt/mujava-3.0`).
2. Add muJava libraries (`mujava.jar` and `openjava2005.jar`) to the `CLASSPATH`.
3. Add the library `tools.jar`, distributed along Java SDK (usually at `$JAVA_HOME/lib/tools.jar`), to the CLASSPATH.

## Test project configuration

1. Configure the file `mujava.config` with the directory (absolute path) that MuJava will use for the test project data (for example, `MuJava_HOME=/opt/mujava-3.0`).
   - This file must be accessible from the `CLASSPATH`.
   - The value of `MuJava_HOME` must not have a trailing slash (otherwise muJava will not work correctly).
2. Create the following directories at `MuJava_HOME`: `classes`, `result`, `src`, and `testset`.
3. Copy the source code of the application under testing to the directory `$MuJava_HOME/src`.
4. Copy the compiled classes of the application under testing to the directory `$MuJava_HOME/classes`.
5. Copy the test cases to the directory `$MuJava_HOME/classes`.

# Mutation generation

## Mutant generation

1. Add the directory `$MuJava_HOME/classes` to the `CLASSPATH`.
2. Run the command `javamujava.gui.GenMutantsMain`.
3. Select the files to be tested (or click the button labeled `All` in the bottom left of the window).
4. Select the mutation operators to be used to generate the mutants.
5. Generate the mutants (using the yellow button in the bottom center of the window).

# muJava
## Test case definition

### Test case format

- Test cases must comprise of public classes with public methods.
- The public methods represents each test case.
  - The method name must start with `test`.
- The methods must return some value (such as `String`).
  - They cannot return `void`.

### Test case definition

1. Implement the test cases.
   - Save them to `$MuJava_HOME/testset`.
2. Compile the test cases.

## Test case execution

1. Remove the directory `$MuJava_HOME/classes` of the `CLASSPATH`.
2. Run the command `javamujava.gui.RunTestMain`.
3. Select the class to be tested.
   - Even if the correct class is already selected, click on the combo box and select it again, otherwise muJava will not recognize the mutants previously generated.
4. Select the the type of mutants to be executed.
   - If no class mutants has been generated, select just the option `Executeonlytraditionalmutants`, otherwise muJava will not run the test cases.
5. Run the test cases by activating the yellow button (which is labeled `RUN`).

# MuClipse

## MuClipse

MuClipse is an Eclipse Plugin which provides a bridge between the existing MuJava API and the Eclipse Workbench.

## Benefits

- Most of the configuration (source files, test files) are read from Eclipse's project definition.
- JUnit support.

## Generation information

- Developed by B. Smith and L. Williams.
- Main site:
    - `http://muclipse.sourceforge.net/`

### Installation

1. Add a new update site to Eclipse:
   - `http://muclipse.sf.net/site`
2. Select the update site just created.
3. Select the MuClipse feature and install it.
4. Restart the workbench after the installation.

# MuClipse
## Project configuration

### Project configuration

1. Change all test case's methods annotated with `@Before` and `@After` to public.
2. Check if you are using a Java runtime environment version 1.6 or greater in the build configuration of your project.
3. Add the package **extendedOj** to the build path.
   - `http://muclipse.sourceforge.net/site/extendedOJ.jar`

## Mutant generation

1. Select the Runas operation in Eclipse and create a new running configuration for `MuClipse:Mutants`.
2. Configure the directories.
   - It is highly recommended to leave the configuration to its default.
   - So, you must change your project configuration to output the classes to a single directory (`bin`).
   - Type the name of class you want to test in the field `ClasstoMutate`.
3. Select the mutation operators to be used.
4. Execute the new run configuration!
   - If you get an `error=12,Cannotallocatememory` exception and you are using Linux, run the following command in the system shell: `echo0>/proc/sys/vm/overcommit_memory`.

## Test execution

1. Select the `Runas` operation in Eclipse and create a new running configuration for `MuClipse:Tests`.
2. Configure the directories.
   - Set the name of the directory where the test cases (their source code) are stored.
   - Select the target class (the class for which you generated the mutants).
   - Select the test case (JUnit class) to be executed.
3. Configure the testing options.
   - If no mutant was generated for a given mutant operator type (method or class), unselect it (otherwise the execution will not work correctly).
4. Execute the new run configuration.

# MuClipse
## View test results

### View test results

1. Select the `MutantsandResults` view at Windows / Show view / Other menu.
2. Click on the right yellow arrow on the right to reload the latest results of test case execution.