# Developing a simple mutation analyzer for the Major Mutation Framework


## Group 52

Divyesh Harit, Rishi Mody, Thuan Banh and Utkarsh Srivastava


### Project Goals

- Implement Mutation Analyzer as a standalone Java program
- Input: Set of mutants and a test suite.
- Output: Mutation score with a summary of mutant kill information

# Major Mutation Framework: An overview

- A mutation analysis tool comprising of
  - A compiler-integrated mutator
  - Mutation analyzer(execute test suite on mutants)
- Enables efficient and scalable mutation analysis
- Enables for both strong and weak mutation analysis

## Input specification

- Build provided source and test files using Major javac and XMutator flag(generates and embeds mutants in code)
- Create a JAR file of the resulting .class files
- Our Mutation Analyzer takes two program arguments:
  - Path to the created JAR
  - Path to the mutants.log file generated upon build

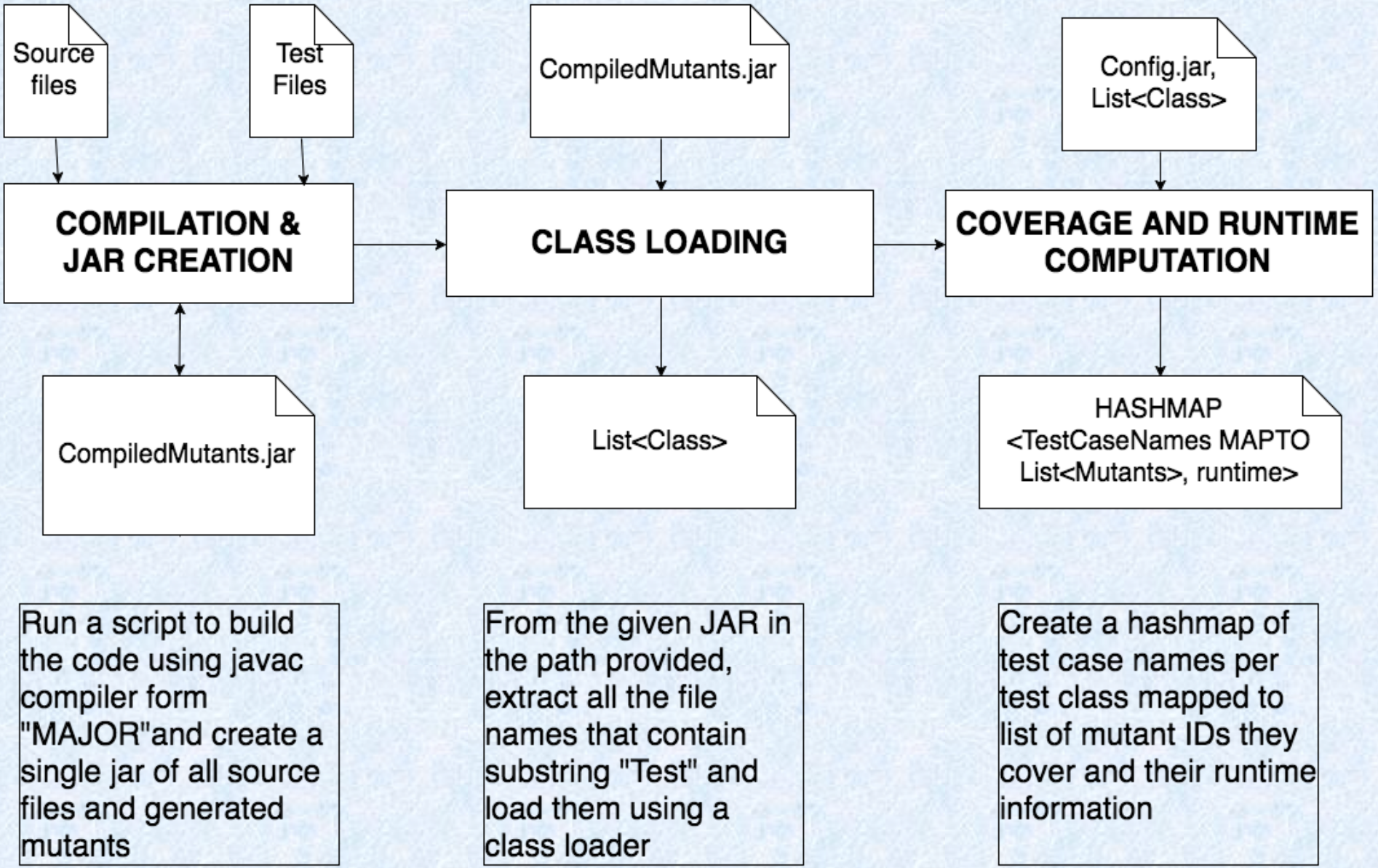
## Output specification

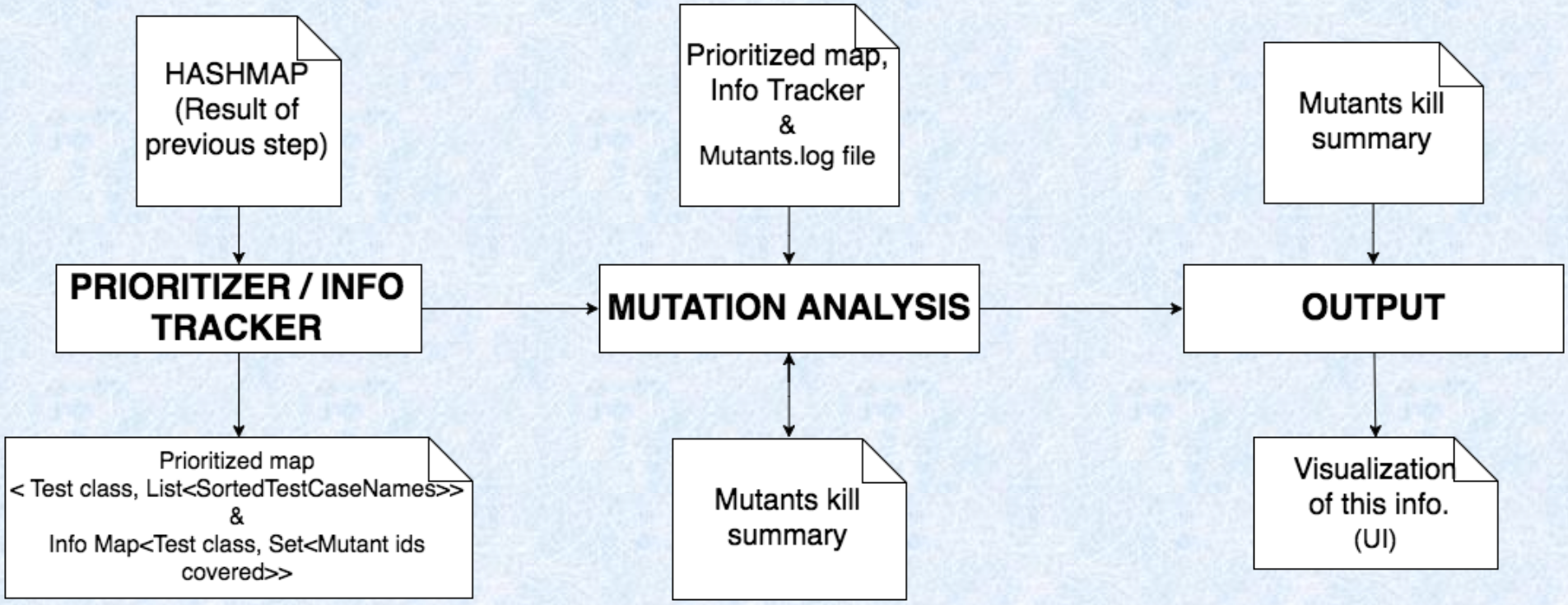
A GUI reporting information about

- Mutation kill rate
- Summary of killed mutants
- List of mutant ids not killed
- Runtime of the code

# Design choices

- Separation of Concerns: Every individual function and associated utilities have been divided into appropriate package structures by focusing distinction on the following three phases:
  - Pre-pass to record run time and coverage information per mutant for a test case
  - Prioritization of a test suite based on above information
  - Running test case on every mutant to analyze mutant kill information
- Composite pattern for output view
- Iterator pattern for executing functions per test class


Source files
Test Files
CompiledMutants.jar
Config.jar, List<Class>
COMPILATION & JAR CREATION
CLASS LOADING
COVERAGE AND RUNTIME COMPUTATION
CompiledMutants.jar
List<Class>
HASHMAP <TestCaseNames MAPTO List<Mutants>, runtime>
Run a script to build the code using javac compiler form "MAJOR"and create a single jar of all source files and generated mutants
From the given JAR in the path provided, extract all the file names that contain substring "Test" and load them using a class loader
Create a hashmap of test case names per test class mapped to list of mutant IDs they cover and their runtime information

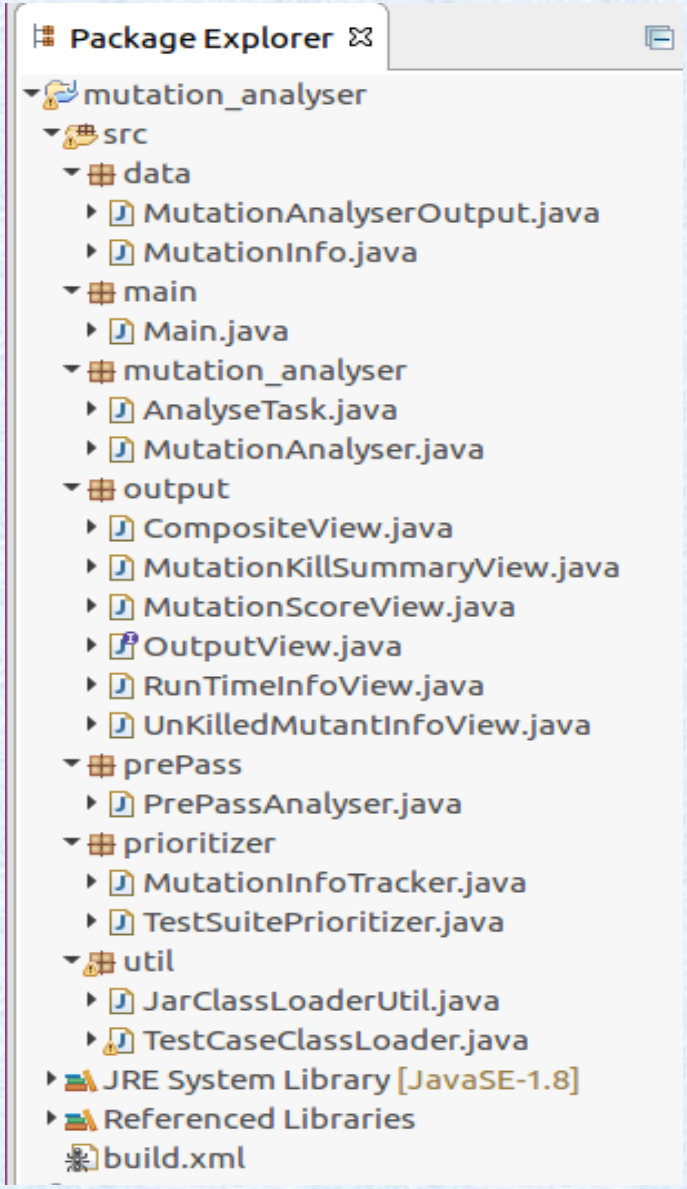



Use the hashmap from previous step to produce a prioritized map of test case names based on increasing runtime, and no.of mutants covered. Also keeps a track of ids of mutants covered by every class, so as to optimize on mutation analyzer.

Find total no. of mutants from mutants.log. Run prioritized test cases leveraging information from Info-Tracker on each mutant . Whenever a test case fails, it returns a failed status by breaking the execution for that mutant
Status code 0: Not killed
Status Code 1: Killed
Status Code 2: Killed with Timeout

Display mutants killed, mode of mutants killed (assertion, timeout, etc), list of mutants not killed, total runtime of entire code

# Code Structuring




# Libraries and Utilities

- InputStreamReader: Read contents from mutants.log file
- URLClassLoader: To load contents of mutants.jar file
- JarEntry: TO iterate over elements in the JAR
- JunitCore, Request, Result: To create a request, run and record the results of a test case in a test file
- JFrame: GUI output
- ExecutorService, Future, Callable: For handling time outs and Thread Execution.

# Data Structures

- MutationInfoClass: Holding information from Pre-Pass phase
- Mutation AnalyserOutput: Hold Mutant kill information to display output.
- Map<String, List<String>> Prioritized Tests: Class name v/s ordered test cases.
- Map<String, Set<Integer>> MutationInfoPerClass: Class name v/s Ids of Covered mutants.

## Optimizations

- Using coverage and runtime information to prioritize test case execution in each test class to reduce overall runtime of the code
- A mutant is set to be killed as soon as a test case for that iteration fails. This saves time by not calling further unnecessary test case executions.
- Using executor service with Timeout to handle test cases stuck in an infinite loop

## Limitations

- The code takes coverage and runtime into consideration but does not cater to state infection through weak-mutation analysis
- JAR as an input is essential as per design.
- To identify test class files in the JAR, we assume that the substring “Test” is present in the .class file being analyzed.

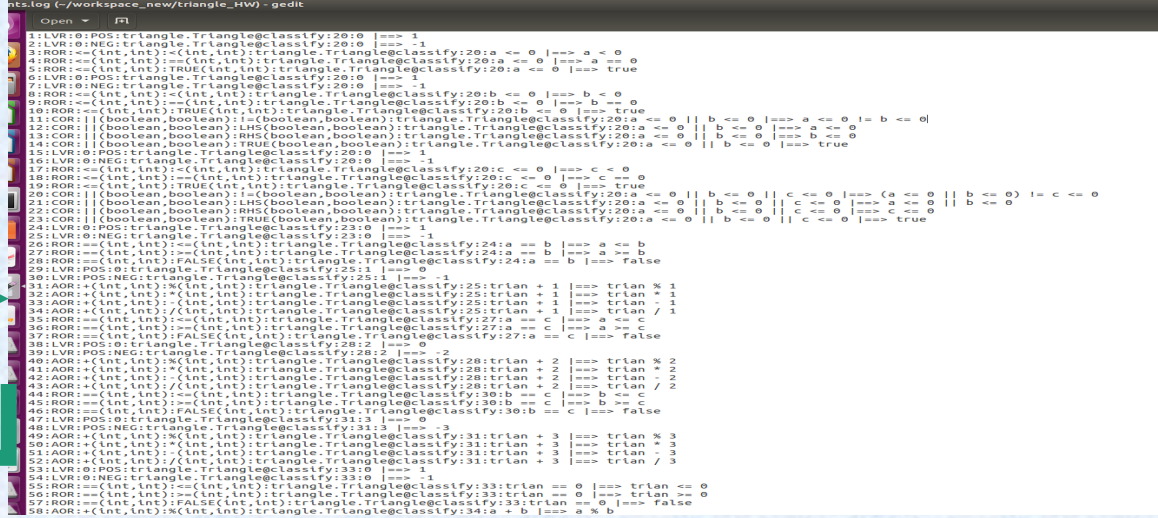
# Code screenshots

File Edit View Search Tools Documents Help

Open

```
ant clean
ant compile
cd bin
jar cf ../../Jars/Mutants.jar *
```

Using javac from "MAJOR" and XMutator




Name: Main

Main | Arguments | JRE | Classpath | Source

Program arguments:

"/home/utkarsh1404/workspace_new/Jars/Mutants.jar" "/home/utkarsh1404/workspace_new/triangle_HW/mutants.log"

Variables...

Problems | Javadoc | Declaration | Console | JUnit

Main [Java Application] /home/utkarsh1404/Downloads/eclipse-installer/jre/bin/java (Dec 13, 2016, 12:09:46 AM)

```
-----Starting Mutation Analysis-----

 STEP 1 : Pre-Pass (Collecting Mutation coverage and run-time info for each test case)

 STEP 2 : Performing Test Suite Prioritization

 STEP 3 : Performing Strong Mutation Analysis

 Total no. of mutants = 139
100 mutants analysed!!
```

Major Mutation Framework : Mutation Analyser

Total Number of Mutants Present : 139
Total Number of Mutants Killed : 130
Mutation Score : 93.5251798561151%

Total no. of Mutants killed due to TimeOut : 0

Total no. of Un-Killed Mutants : 9
Un-Killed Mutant Ids : [11, 55, 72, 83, 94, 104, 120, 127, 136]

Run Time of code (in seconds) : 1.106

# Sample Run on Real World Data Set: Numerics4J

- Xmutator Flag: Set to “ALL”
- Total number of Mutants generated: 16708