
# DSpot: how developers can amplify existing test cases.

Benjamin DANGLOT

January $30^{th}$, 2019

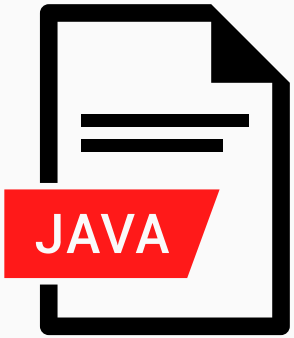
*benjamin.danglot@inria.fr*

**STAMP Workshop** INRIA - Sophia Antipolis

# Demo

# DSpot

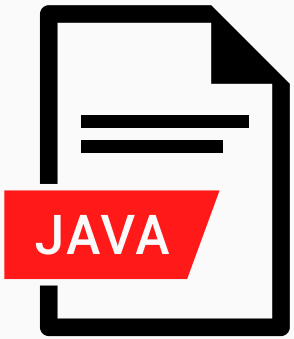
# DSpot: Principle



Program


Test suite

# DSpot: Principle

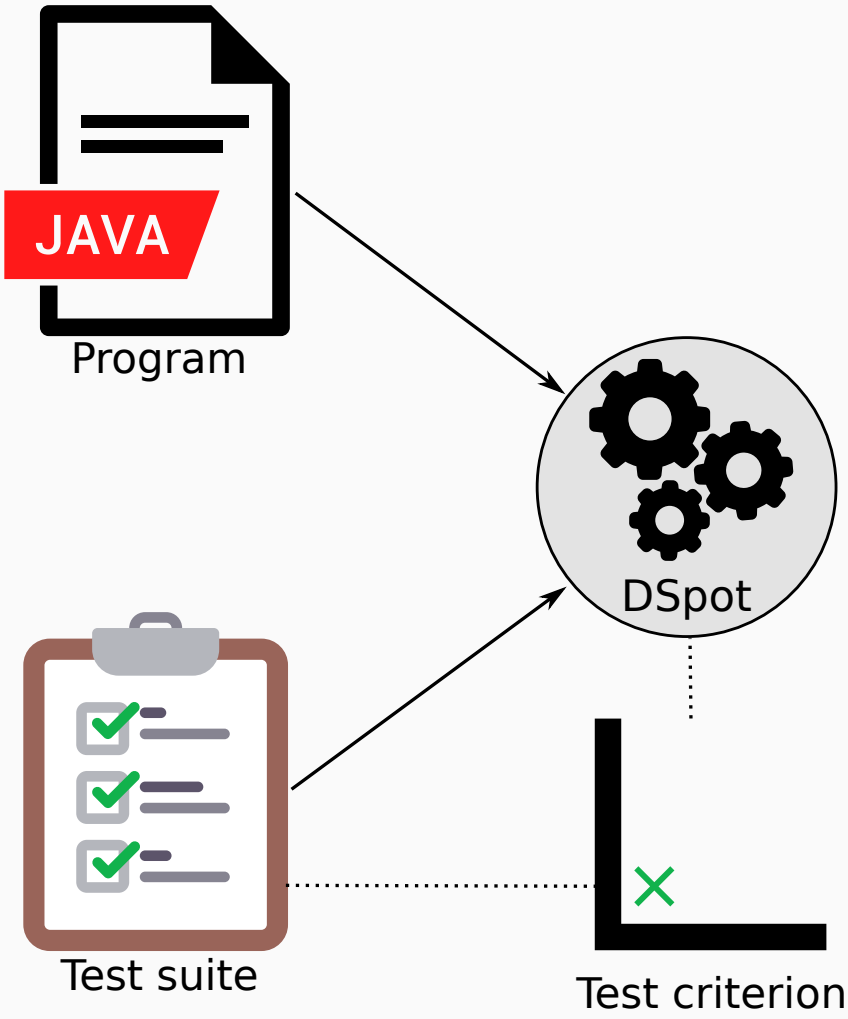



Program

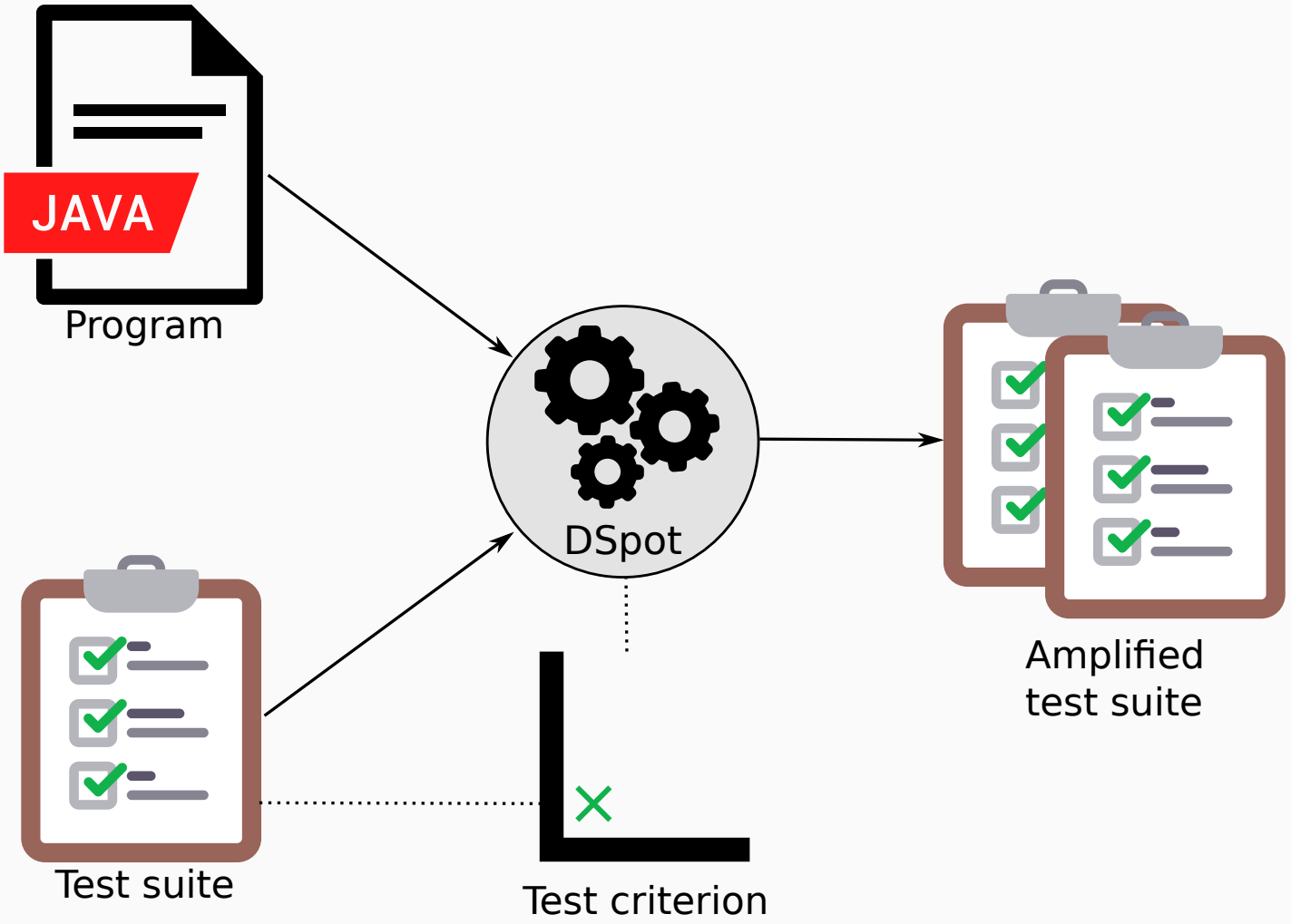
Test suite

Test criterion

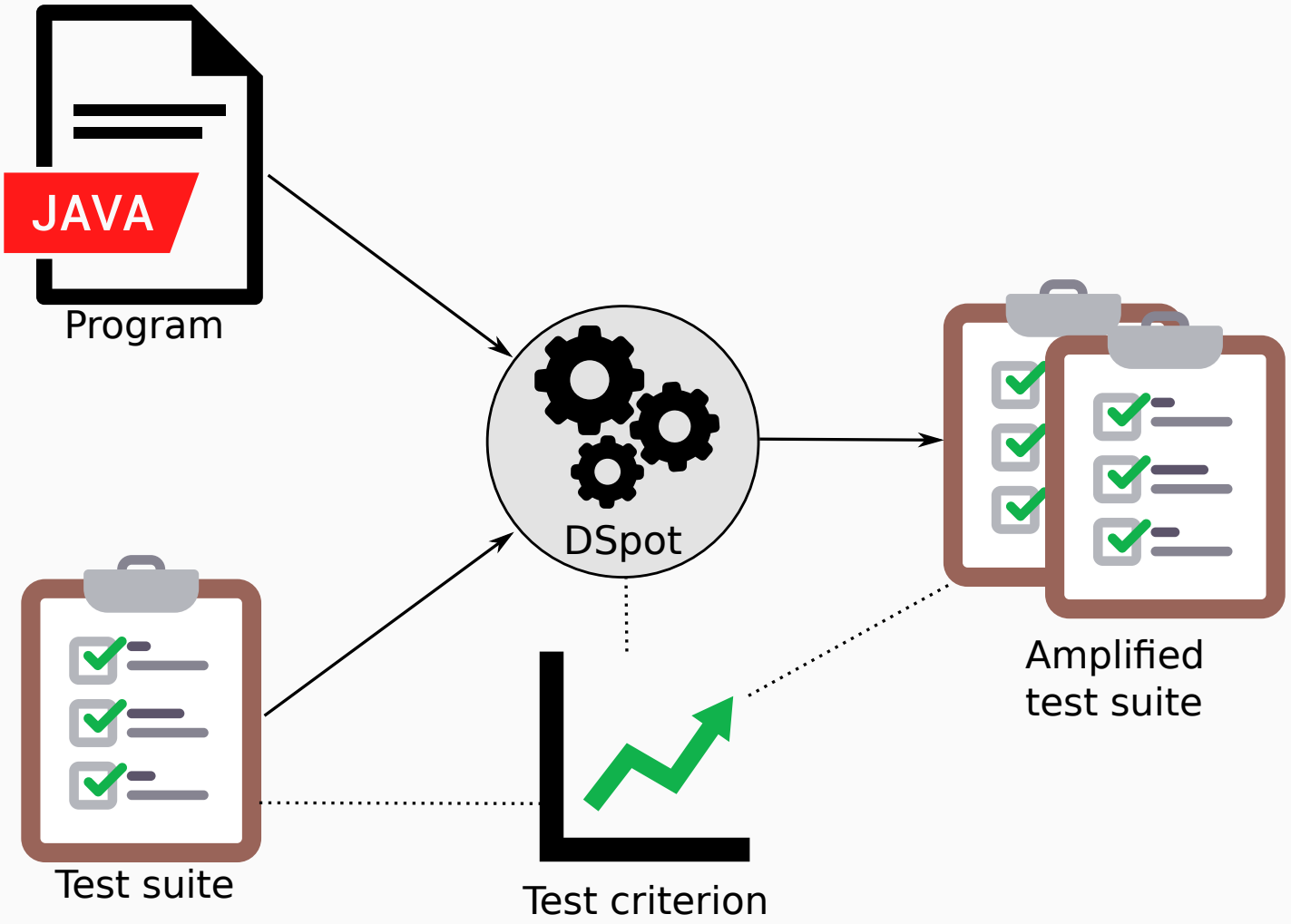
# DSpot: Principle



# DSpot: Principle



# DSpot: Principle



# Test Example

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

# Test Example

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

In blue, the input of the test

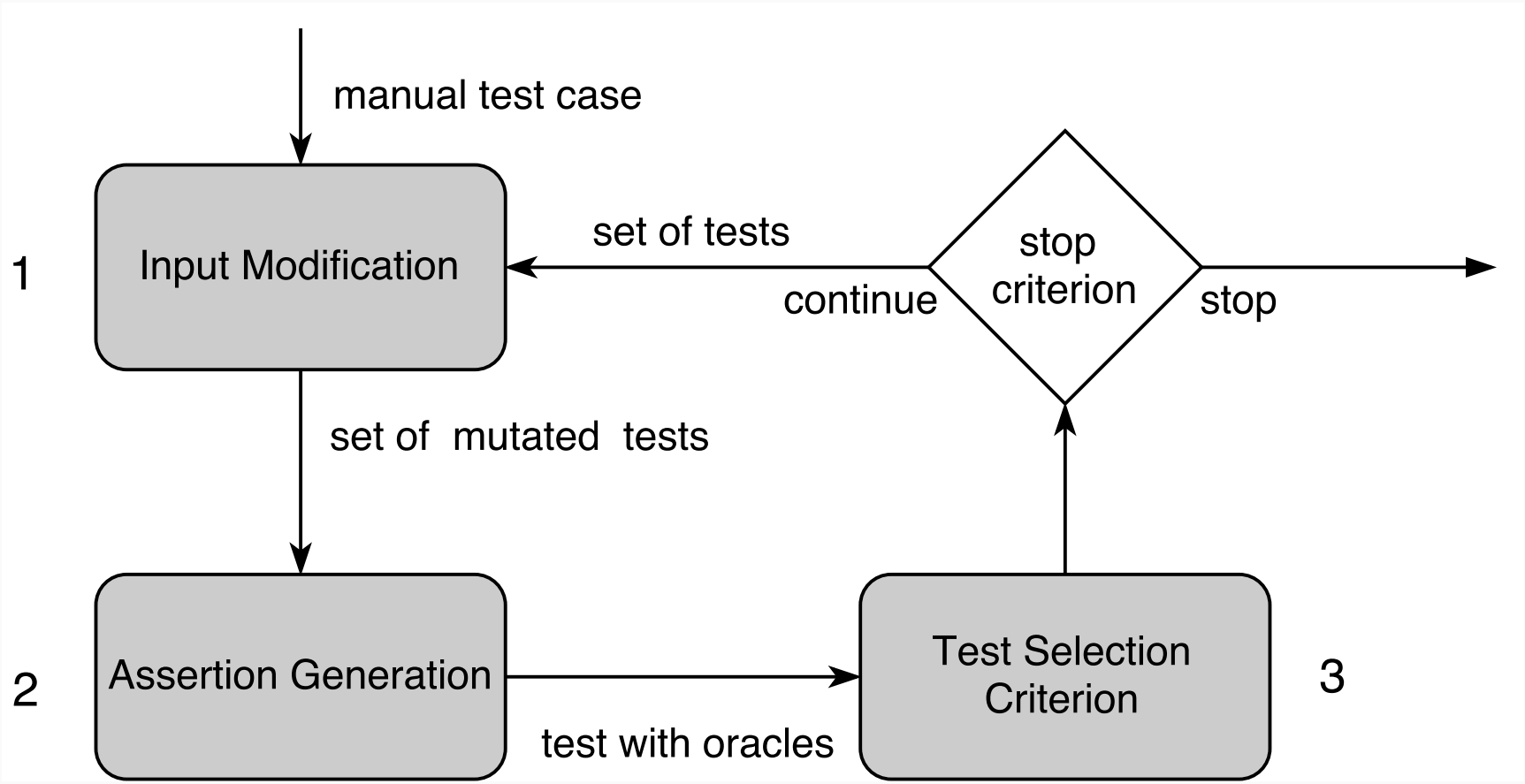
# Test Example

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

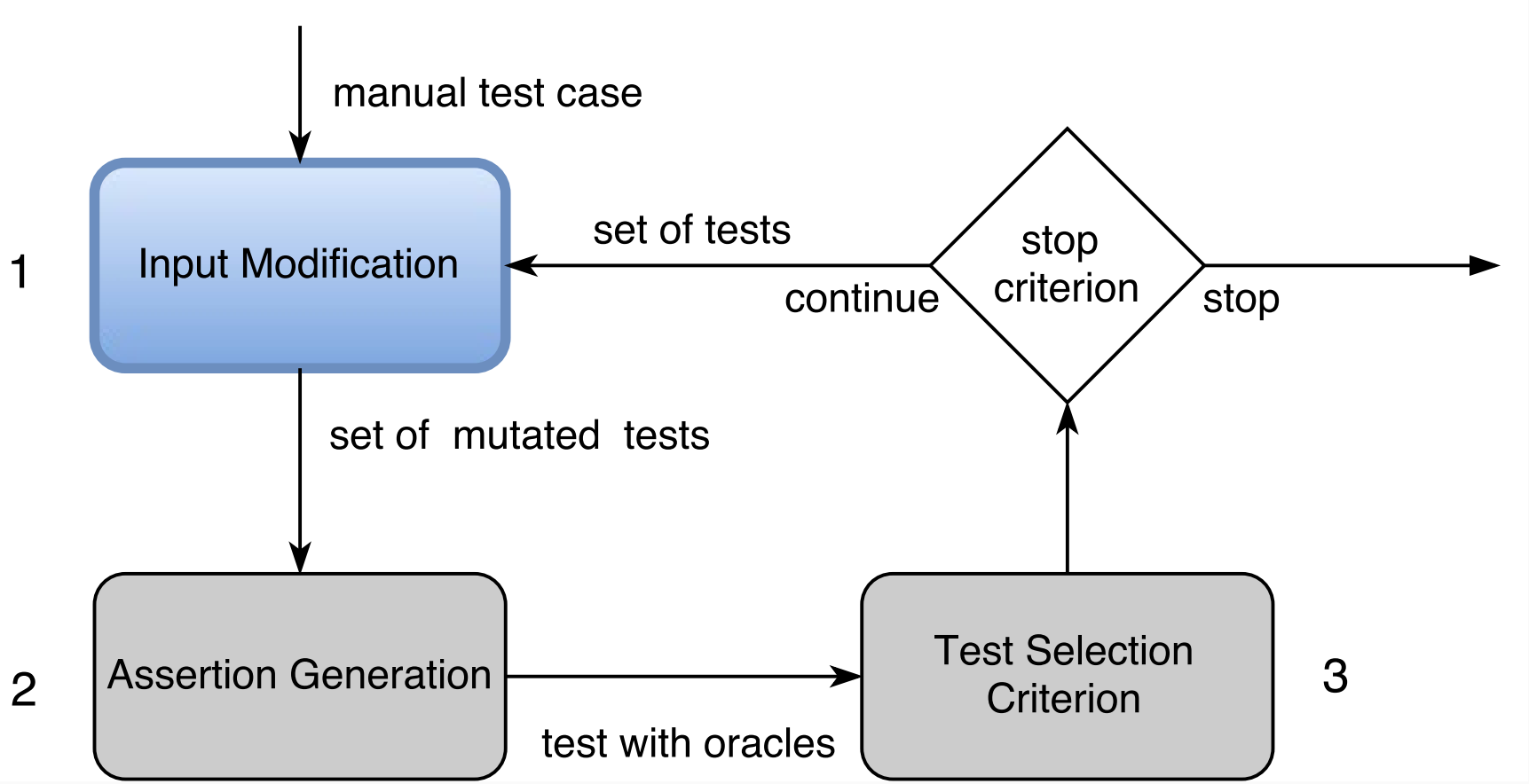
In blue, the input of the test

In green, an oracle of the test to verify the current behavior

# How DSpot works?



# DSpot: 1. Input Modification



# DSpot: 1. Input Modification

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

Modifies the input to create new state of the program

# DSpot: 1. Input Modification

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

Remove method call

Removes a method call to create a new state of the program

# DSpot: 1. Input Modification

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertFalse(benjamin.isHungry());
}
```

Removes a method call to create a new state of the program

# DSpot: 1. Input Modification

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertFalse(benjamin.isHungry());
}
```

Removes a method call to create a new state of the program

Before:

*benjamin.isHungry*() → *false*

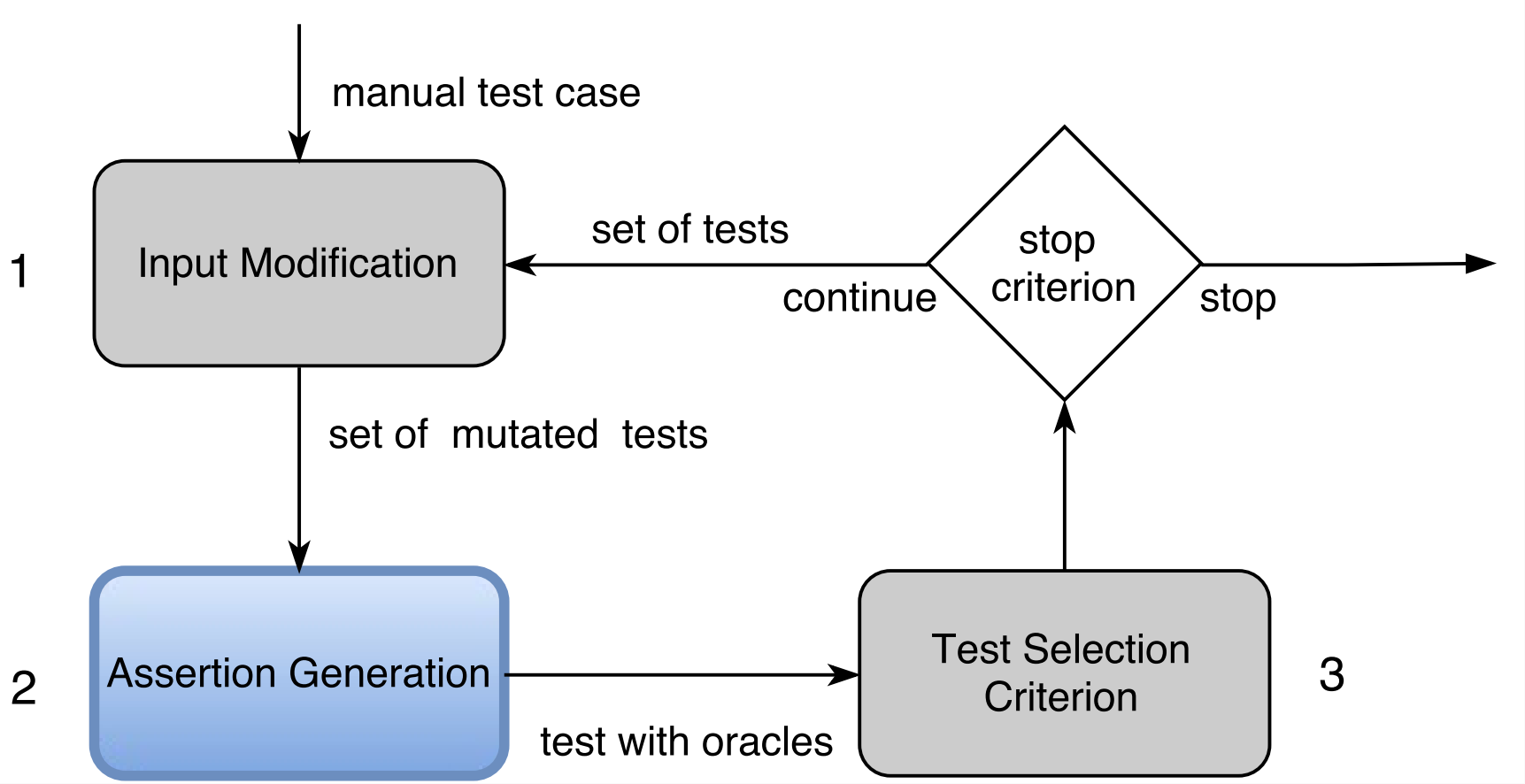
*benjamin.isHappy*() → *true*

After:

*benjamin.isHungry*() → *true*

*benjamin.isHappy*() → *false*

# DSpot: 2. Assertion Generation



# DSpot: 2. Assertion Generation

1. Remove existing assertions
2. Instrument the test
3. Run Instrumented test to collect values
4. Generate assertions based on values collected

# DSpot: 2. Assertion Generation

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertFalse(benjamin.isHungry());
}
```

A. Removes existing assertions

# DSpot: 2. Assertion Generation

```
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    benjamin.isHungry();
}
```

assertion removed

A. Removes existing assertions

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    benjamin.isHungry();
}
```

B. Instruments the code to gather new values

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    benjamin.isHungry();
    Log.log(benjamin, id: "benjamin");
}
```

B. Instruments the code to gather new values

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    benjamin.isHungry();
    Log.log(benjamin, id: "benjamin");
}
```

C. Runs the instrumented test and obtains observations

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    benjamin.isHungry();
    Log.log(benjamin, id: "benjamin");
}
```

C. Runs the instrumented test and obtains observations

Observations:

*benjamin.isHungry*() → *true*

*benjamin.isHappy*() → *false*

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertTrue(benjamin.isHungry());
    assertFalse(benjamin.isHappy());
}
```

D. Generates new assertions based on observations

Observations:

$benjamin.isHungry() \rightarrow true$

$benjamin.isHappy() \rightarrow false$

# DSpot: 2. Assertion Generation

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertTrue(benjamin.isHungry());
    assertFalse(benjamin.isHappy());
}
```

D. Generates new assertions based on observations

Observations:

*benjamin.isHungry*() → *true*

*benjamin.isHappy*() → *false*

# DSpot: Amplification of Test Results

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

Original test

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();

    assertTrue(benjamin.isHungry());
    assertFalse(benjamin.isHappy());
}
```

Amplified test

# DSpot: Amplification of Test Results

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    benjamin.eat(tacos);
    assertFalse(benjamin.isHungry());
}
```

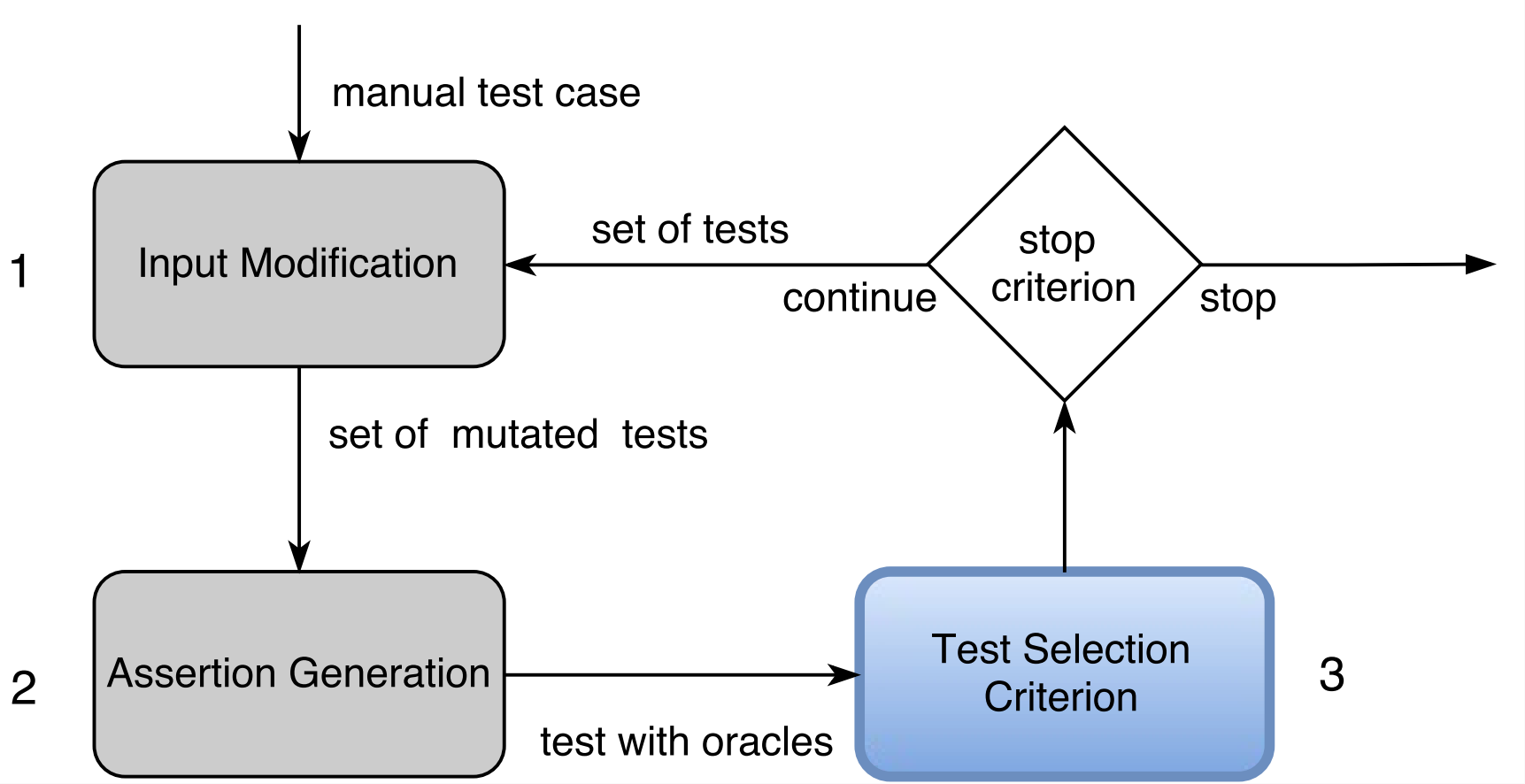
Original test

```java
@Test
public void test() {
    Tacos tacos = new Tacos();
    Benjamin benjamin = new Benjamin();
    →Method call removed
    assertTrue(benjamin.isHungry());
    assertFalse(benjamin.isHappy());
}
```

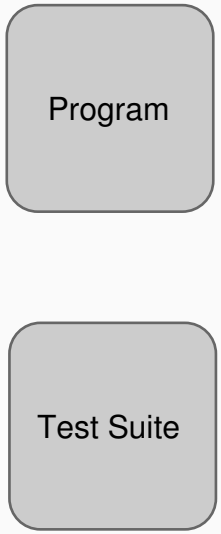
**Assertion Generated**

Amplified test

# DSpot: 3. Test Selection

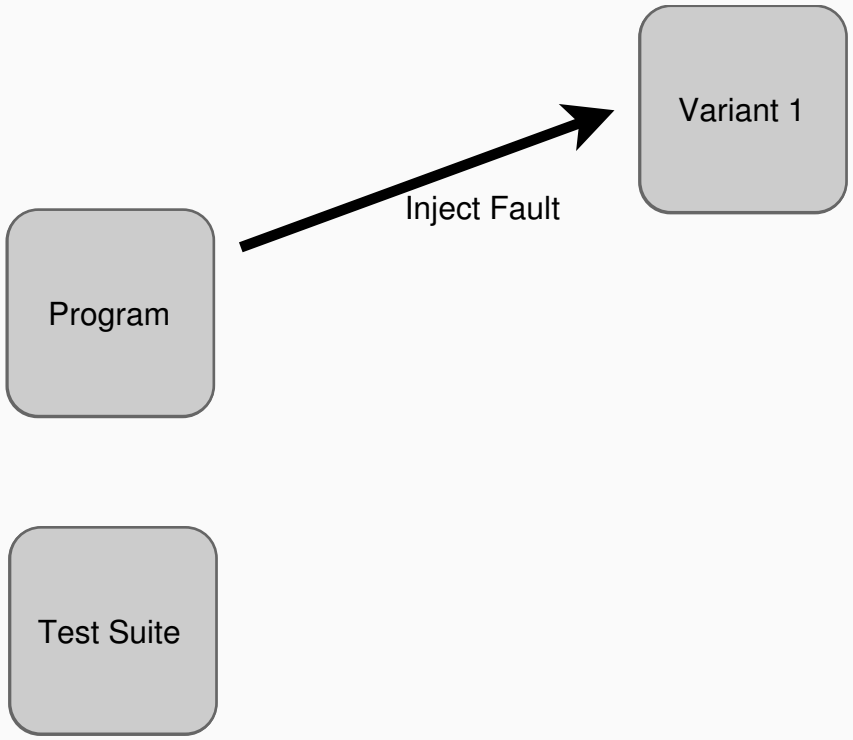


# Pre-Requisite: Mutation Analysis



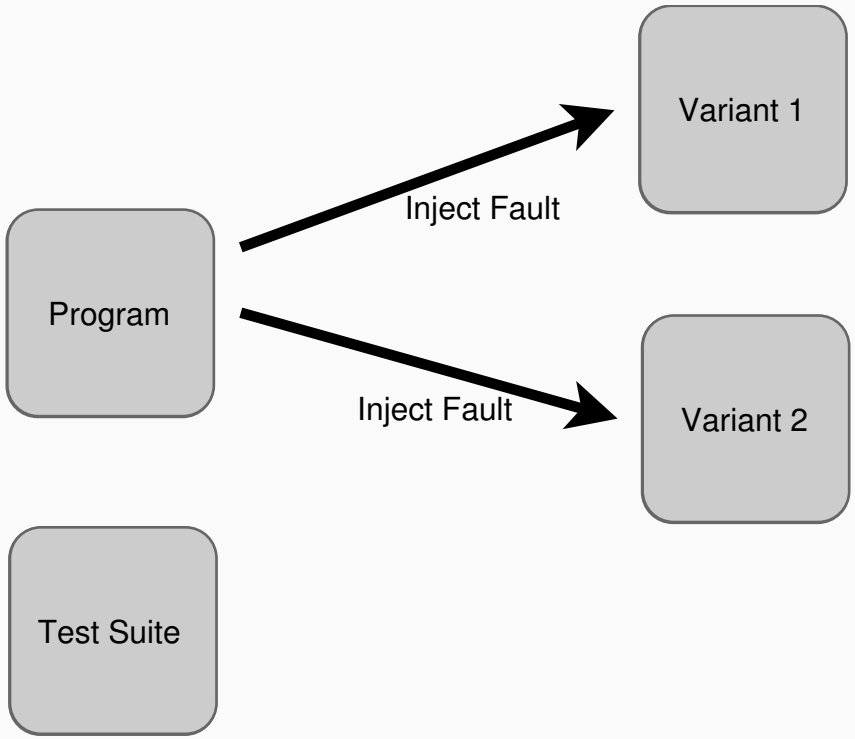

The mutation analysis computes the **mutation score**, a measure of the efficiency of tests

# Mutation Analysis



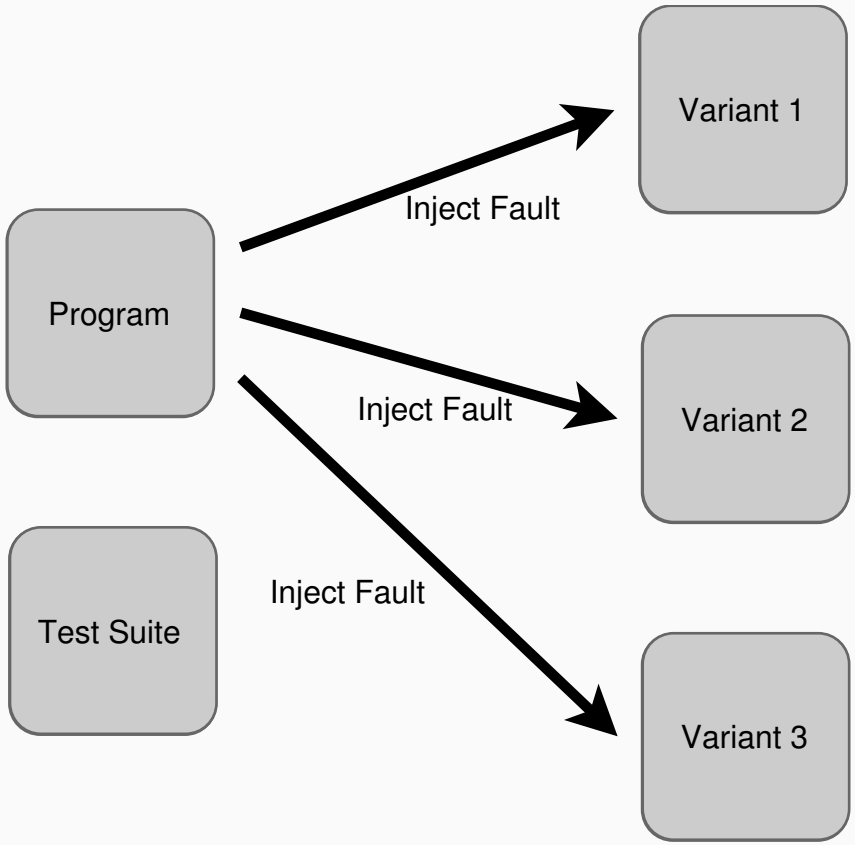

First, creates variants of program by injecting faults

# Mutation Analysis



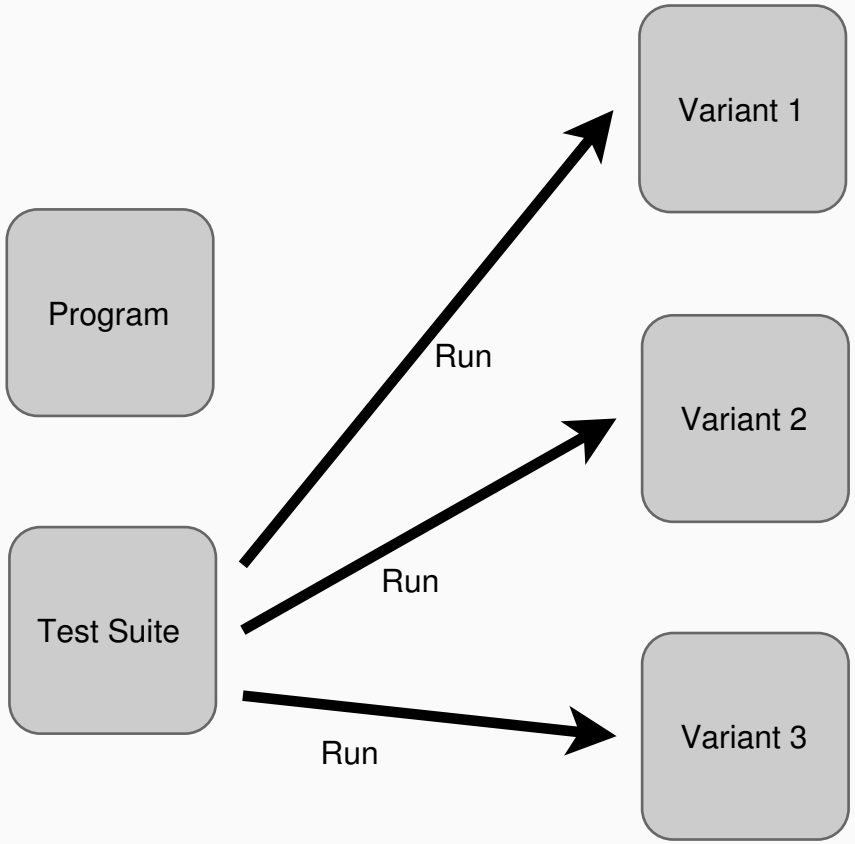

First, creates variants of program by injecting faults

# Mutation Analysis



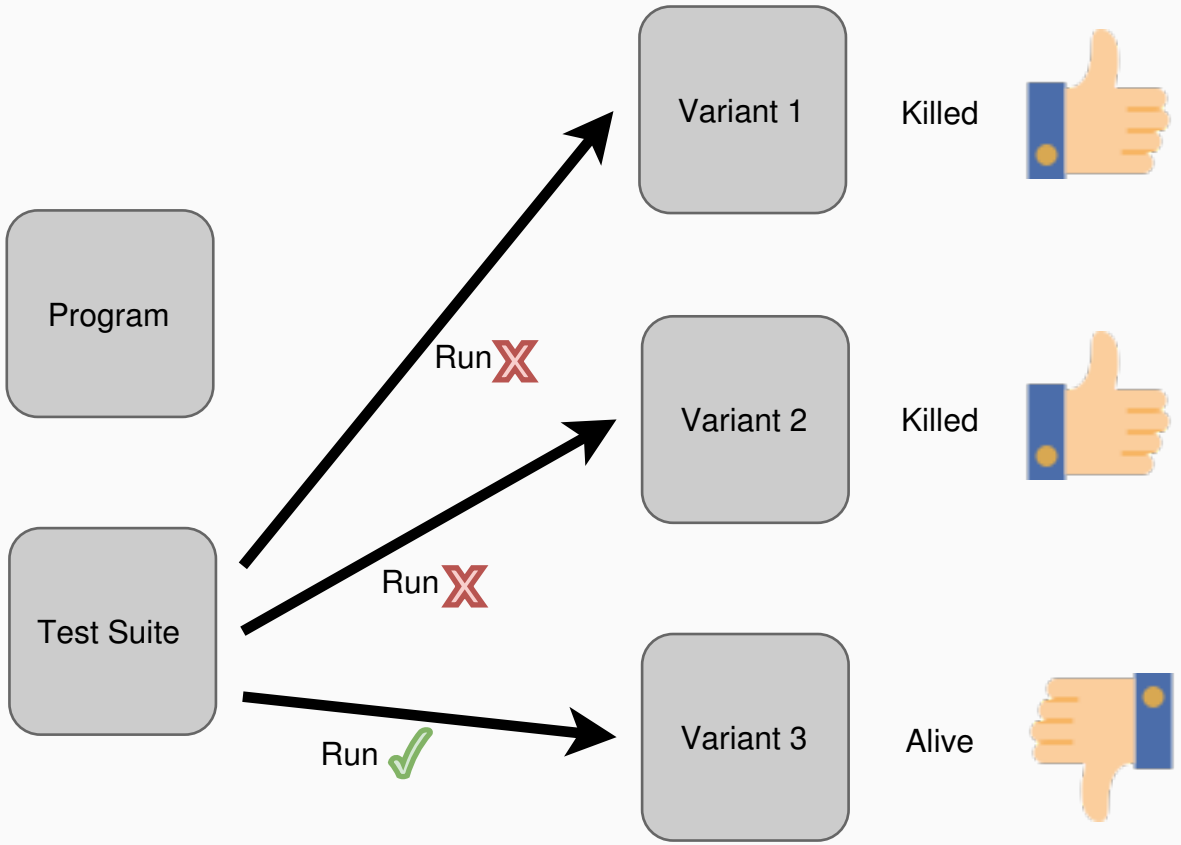

First, creates variants of program by injecting faults

# Mutation Analysis




Then, runs tests against each variant

# Mutation Analysis

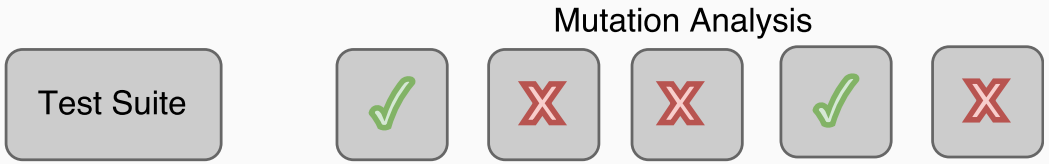



Failing test $\Leftrightarrow$ mutant killed $\Leftrightarrow$ behavior variation detected

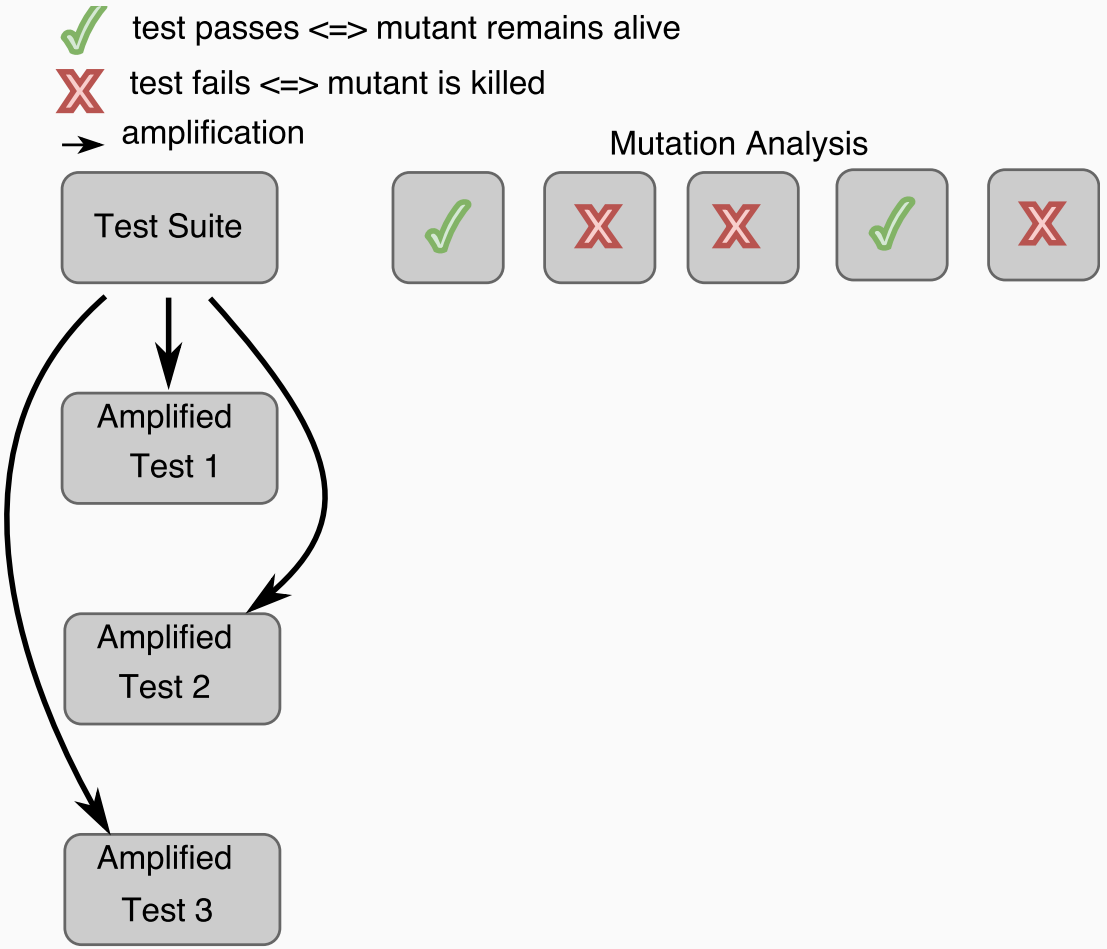
# Selection Of Test Using Mutation Analysis

✓ test passes <=> mutant remains alive

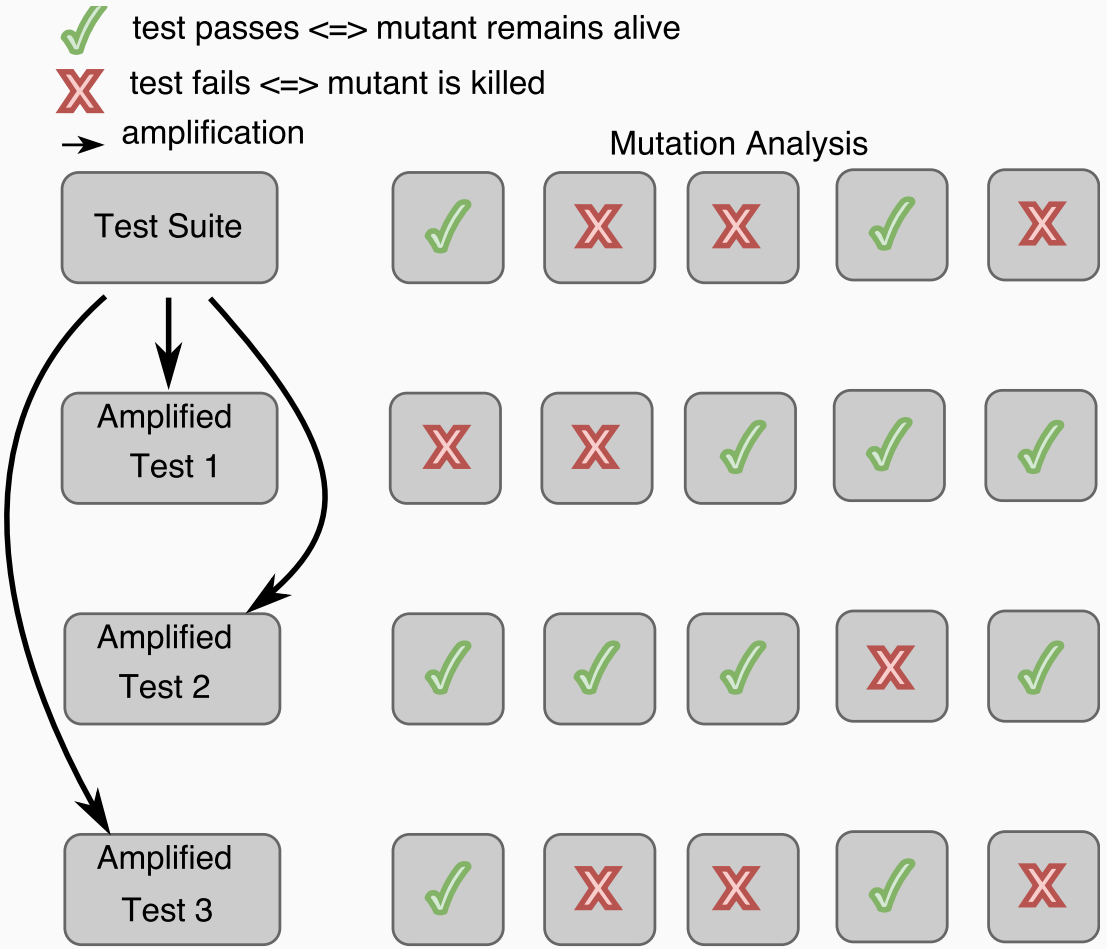
✗ test fails <=> mutant is killed



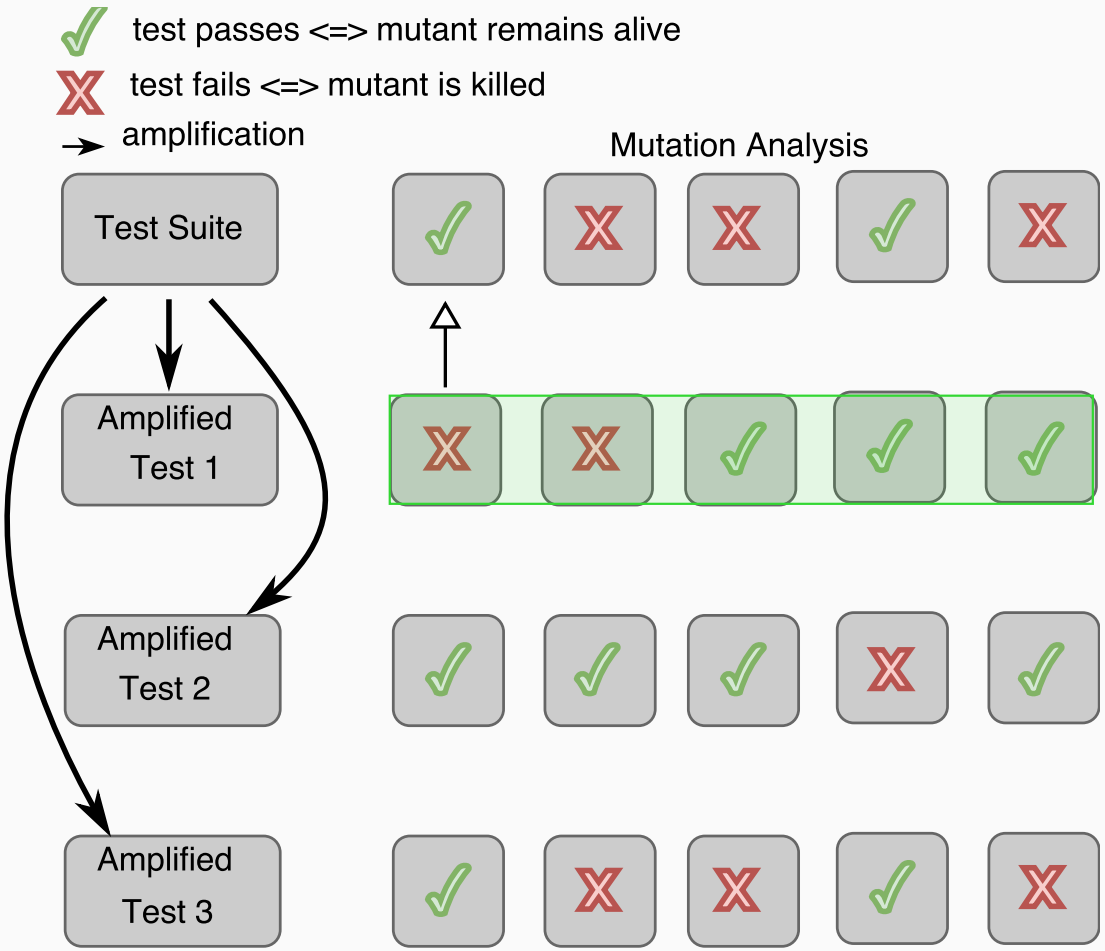
# Selection Of Test Using Mutation Analysis



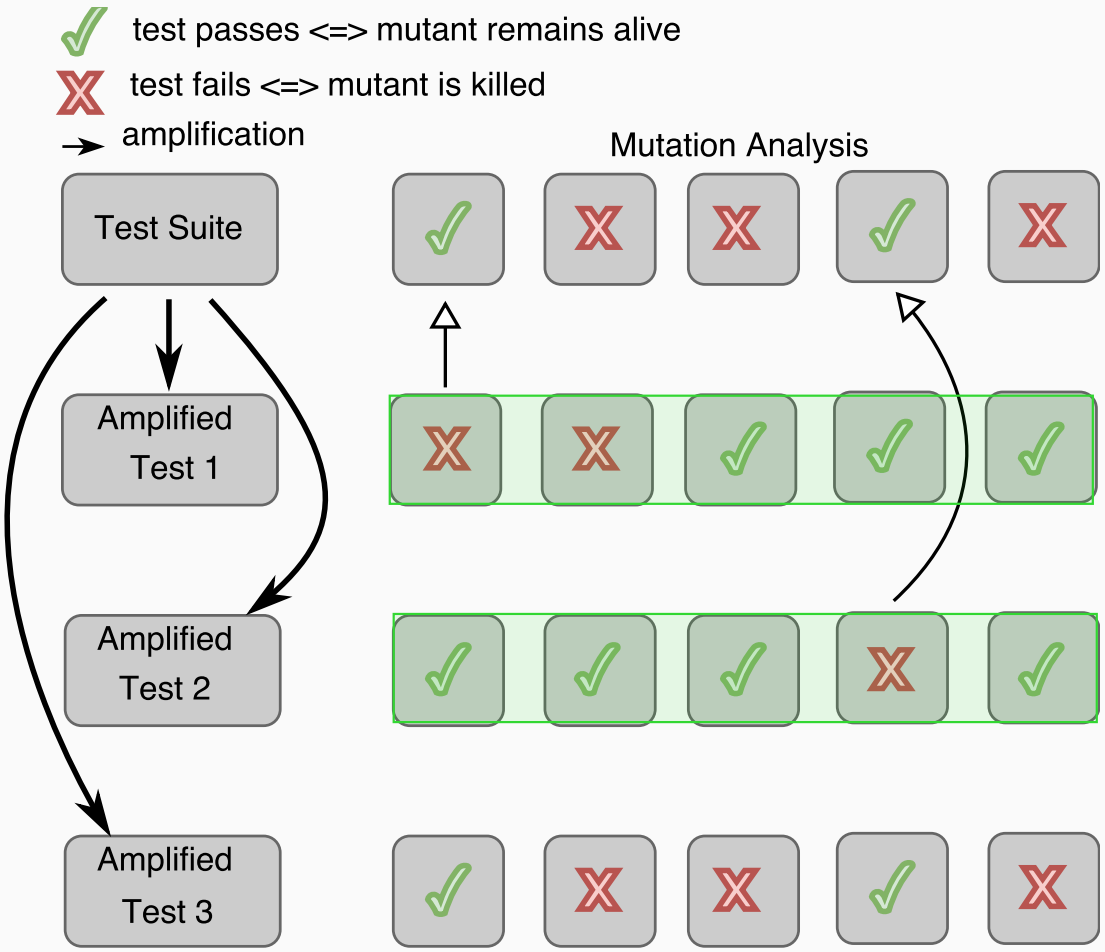
# Selection Of Test Using Mutation Analysis



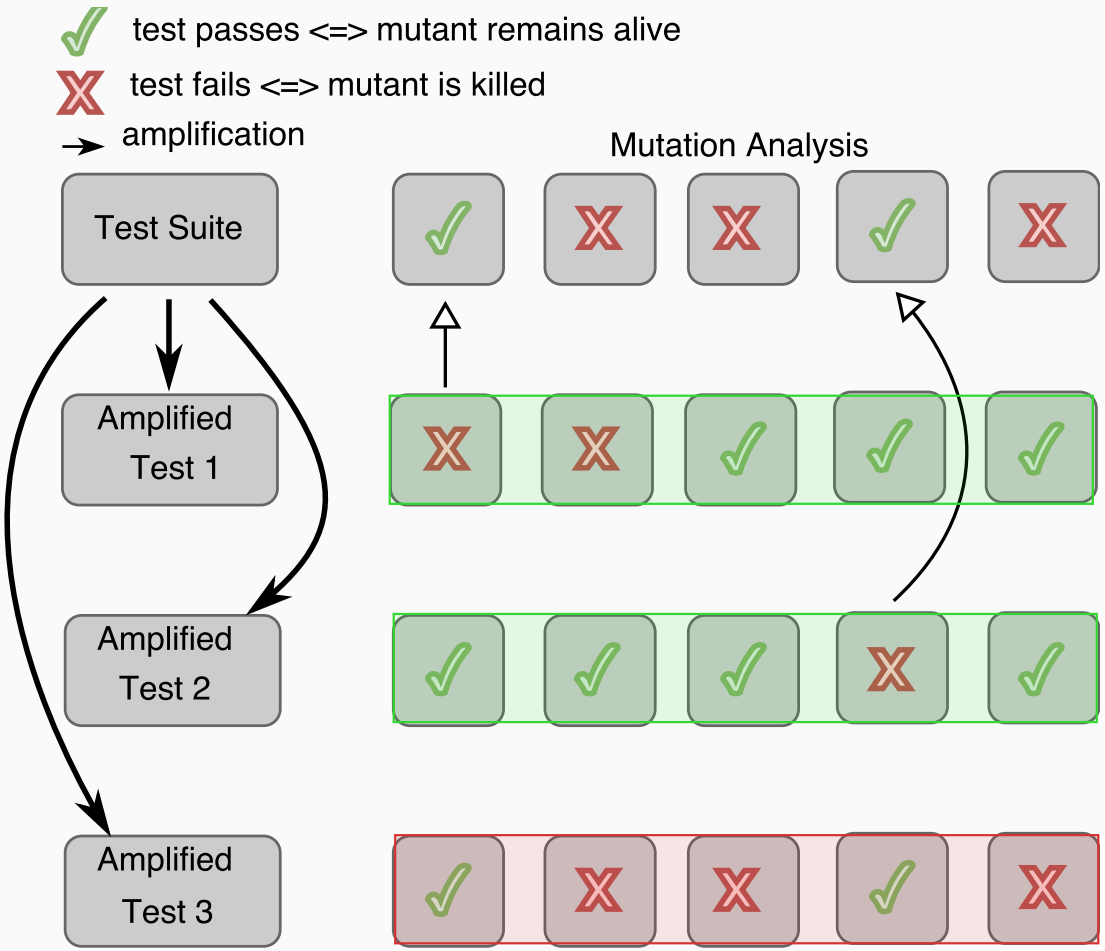
# Selection Of Test Using Mutation Analysis



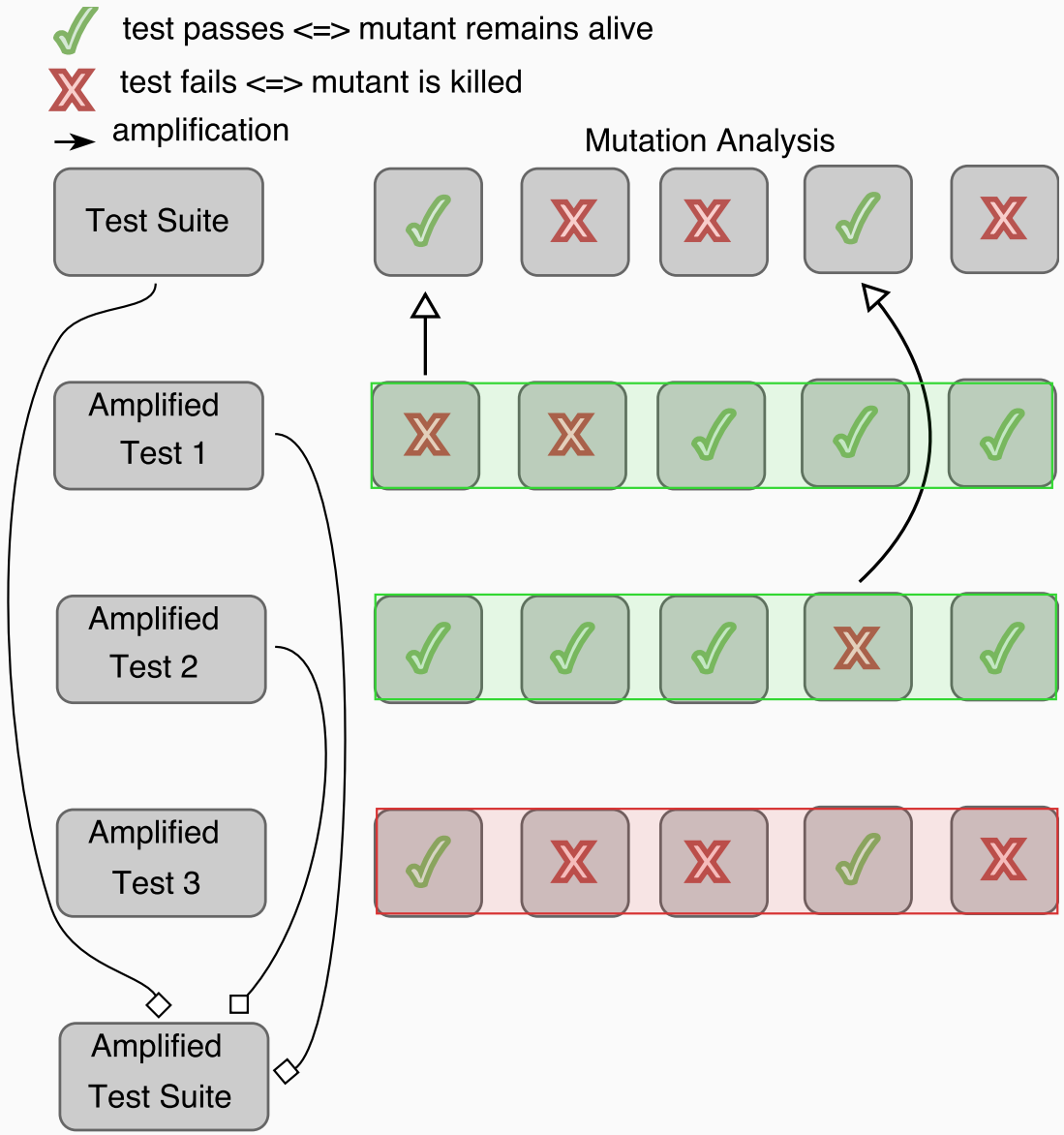
# Approach


test passes <=> mutant remains alive
test fails <=> mutant is killed
amplification
Mutation Analysis
Test Suite
Amplified Test 1
Amplified Test 2
Amplified Test 3

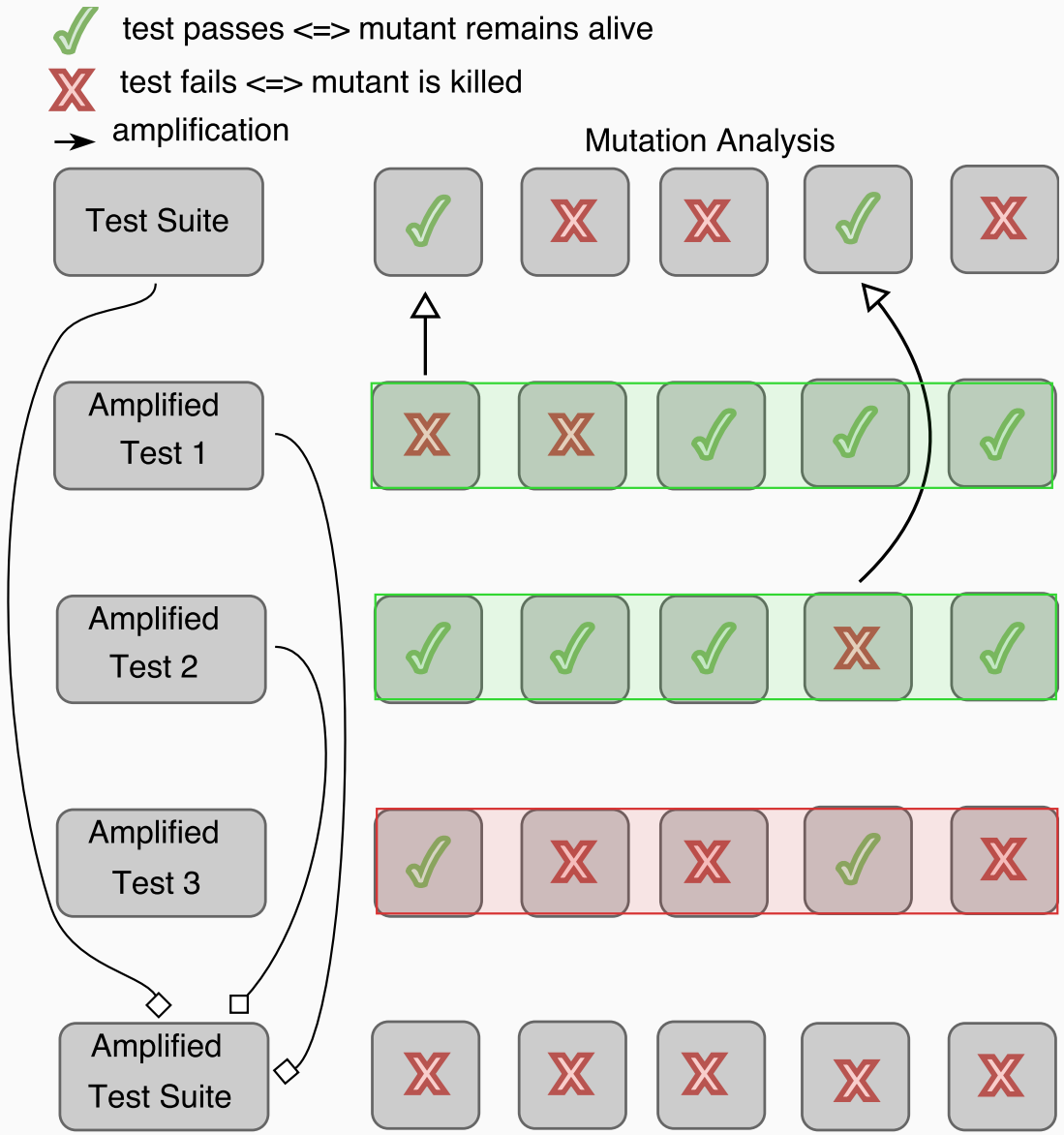
# Selection Of Test Using Mutation Analysis



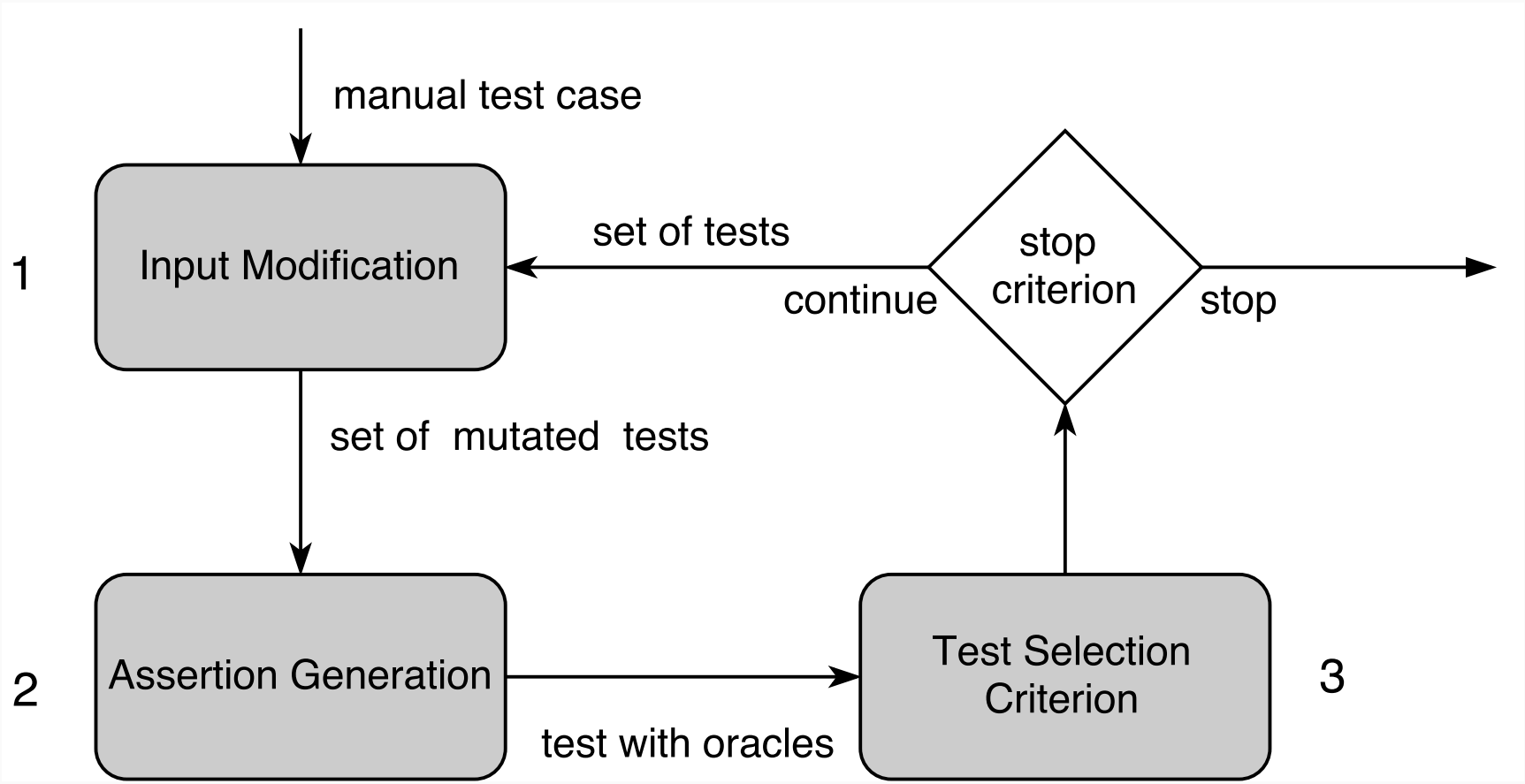
# Selection Of Test Using Mutation Analysis



# Selection Of Test Using Mutation Analysis



# DSpot: iteration



# Hands-on!

# Hands-on!

- GitHub repository:
  `https://github.com/STAMP-project/dspot.git`
- Using maven plugins (no manual downloading required)
- First, amplify a toy-project from STAMP-project repository (dhell)
- Then, amplify your own project!
- Bunch of scripts available at:

  wget https://danglotb.github.io/resources/cmd_dspot_tuto.zip

# Toy-project to play with DSpot

- Clone: `git clone https://github.com/STAMP-project/dhell.git`
- Amplify:

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests \
    -DtargetOneTestClass \
    -Damplifiers=MethodAdd,MethodRemove \
    -Diteration=1 \
    -Dtest=eu.stamp_project.examples.dhell.HelloAppTest
```

- Output:

```
======= REPORT =======
PitMutantScoreSelector:
The original test suite kills 12 mutants
The amplification results with 1 new tests
it kills 2 more mutants
```

- Look output in `target/dspot/output/`.
- You should find some reports (.txt, .json), and the java file

# On your own project

Amplify only assertions:

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests \
    -DtargetOneTestClass \
    -Dtest=your.TestClass
```

Amplify input: option -Damplifiers=amplifier1,amplifier2

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests \
    -DtargetOneTestClass \
    -Damplifiers=MethodAdd \
    -Diteration=1 \
    -Dtest=your.TestClass
```

values: MethodAdd, MethodRemove, AllLiteralAmplifiers, MethodGeneratorAmplifier, ReturnValueAmplifier

`https://github.com/STAMP-project/dspot.git`

# Backup hands-on

- Compute the original mutation score of class with:

```
mvn clean test -DskipTests \
    eu.stamp-project:pitmp-maven-plugin:descartes \
    -DoutputFormats=XML \
    -DtargetTests=your.TestClass \
    (-DtargetModules=myModule)
```

- Copy: `cp target/pit-reports/AAAAMMDDHHMM/mutations.xml ./mutations.xml`
- Amplify with `mutations.xml`:

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests \
    -Dpath-pit-result=mutations.xml \
    -Dtest=your.TestClass
```

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests \
    -Dpath-pit-result=mutations.xml \
    -Damplifiers=MethodAdd \
    -Dtest=your.TestClass
```

- Iterations: `-Diteration=<int>`, 1, 2, 3...
- Amplifiers: `-Damplifiers=<a1,a2,...an>`, MethodAdd, MethodRemove, TestDataMutator, MethodGeneratorAmplifier