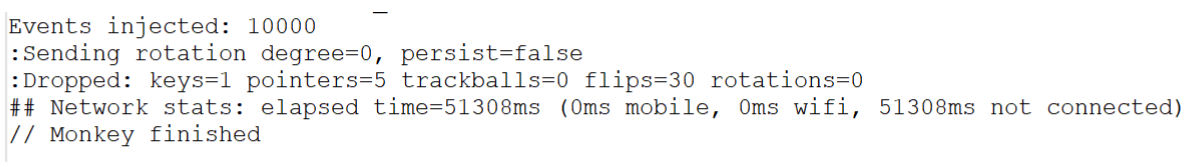
CPEN522　Assignment 4

Ziyan Gong/94478161, Zhen Wang/98552169

1　Automated Test Generation

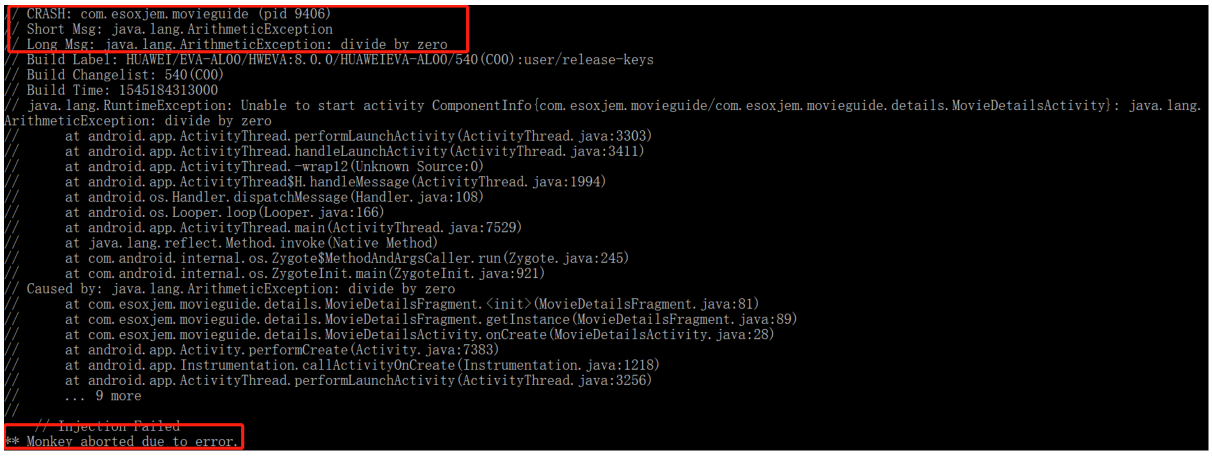
Monkey



```
Events injected: 10000
:Sending rotation degree=0, persist=false
:Dropped: keys=1 pointers=5 trackballs=0 flips=30 rotations=0
## Network stats: elapsed time=51308ms (0ms mobile, 0ms wifi, 51308ms not connected)
// Monkey finished
```

Time: As the log shows, it takes 51308ms to generate 10000 random tests.

FindBug: We introduced an Arithmetic Bug in “MovieDetailsFragment.java”. Monkey found the bug and here is the output report.



```
// CRASH: com.esoxjem.movieguide (pid 9406)
// Short Msg: java.lang.ArithmeticException
// Long Msg: java.lang.ArithmeticException: divide by zero
// Build Label: HUAWEI/EVA-AL00/HWEVA:8.0.0/HUAWEIEVA-AL00/540(C00):user/release-keys
// Build Changelist: 540(C00)
// Build Time: 1545184313000
// java.lang.RuntimeException: Unable to start activity ComponentInfo{com.esoxjem.movieguide/com.esoxjem.movieguide.details.MovieDetailsActivity}: java.lang.
ArithmeticException: divide by zero
//      at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3303)
//      at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3411)
//      at android.app.ActivityThread.-wrap12(Unknown Source:0)
//      at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1994)
//      at android.os.Handler.dispatchMessage(Handler.java:108)
//      at android.os.Looper.loop(Looper.java:166)
//      at android.app.ActivityThread.main(ActivityThread.java:7529)
//      at java.lang.reflect.Method.invoke(Native Method)
//      at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:245)
//      at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:921)
// Caused by: java.lang.ArithmeticException: divide by zero
//      at com.esoxjem.movieguide.details.MovieDetailsFragment.<init>(MovieDetailsFragment.java:81)
//      at com.esoxjem.movieguide.details.MovieDetailsFragment.getInstance(MovieDetailsFragment.java:89)
//      at com.esoxjem.movieguide.details.MovieDetailsActivity.onCreate(MovieDetailsActivity.java:28)
//      at android.app.Activity.performCreate(Activity.java:7383)
//      at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1218)
//      at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3256)
//      ... 9 more
//
    // Injection Failed
** Monkey aborted due to error.
```

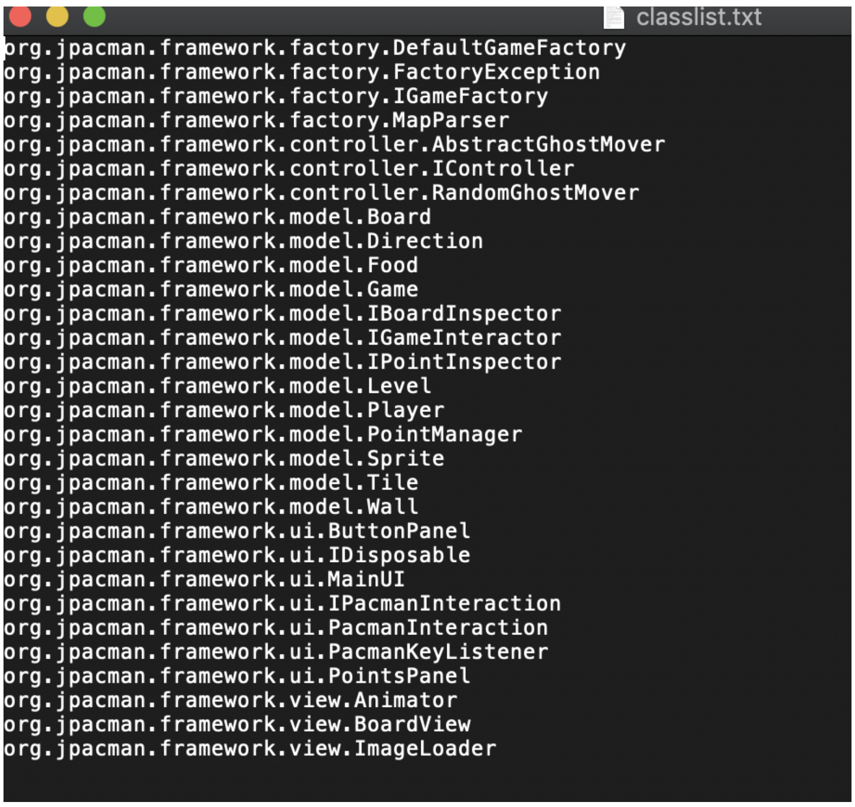
Evaluation: In general, monkey is easy to use for me. It is a command-line tool that you can run on any emulator instance or on a device with the target APK installed. Monkey is included in Android Studio, so you can launch it in the adb shell directly. This tool help us avoid some boring and repetitive testing operations.

Limitations: After several tests, I found some limitations of Monkey. Firstly, the target testing object for Monkey must be apk. Secondly, the stream of user events is generated randomly. If a developer wants to reproduce the error, he must use the same pseudo-random seed. Thirdly, a tester need to unlock the screen before the testing, otherwise Monkey couldn't enter the app correctly. Besides, when interacting with the system, the tools in toolbar are easily mistouched, which might influence the testing, like turning off WiFi.

Improvements: If Monkey could take screenshot when detecting errors, that would be better for developers to understand abnormal scenarios.

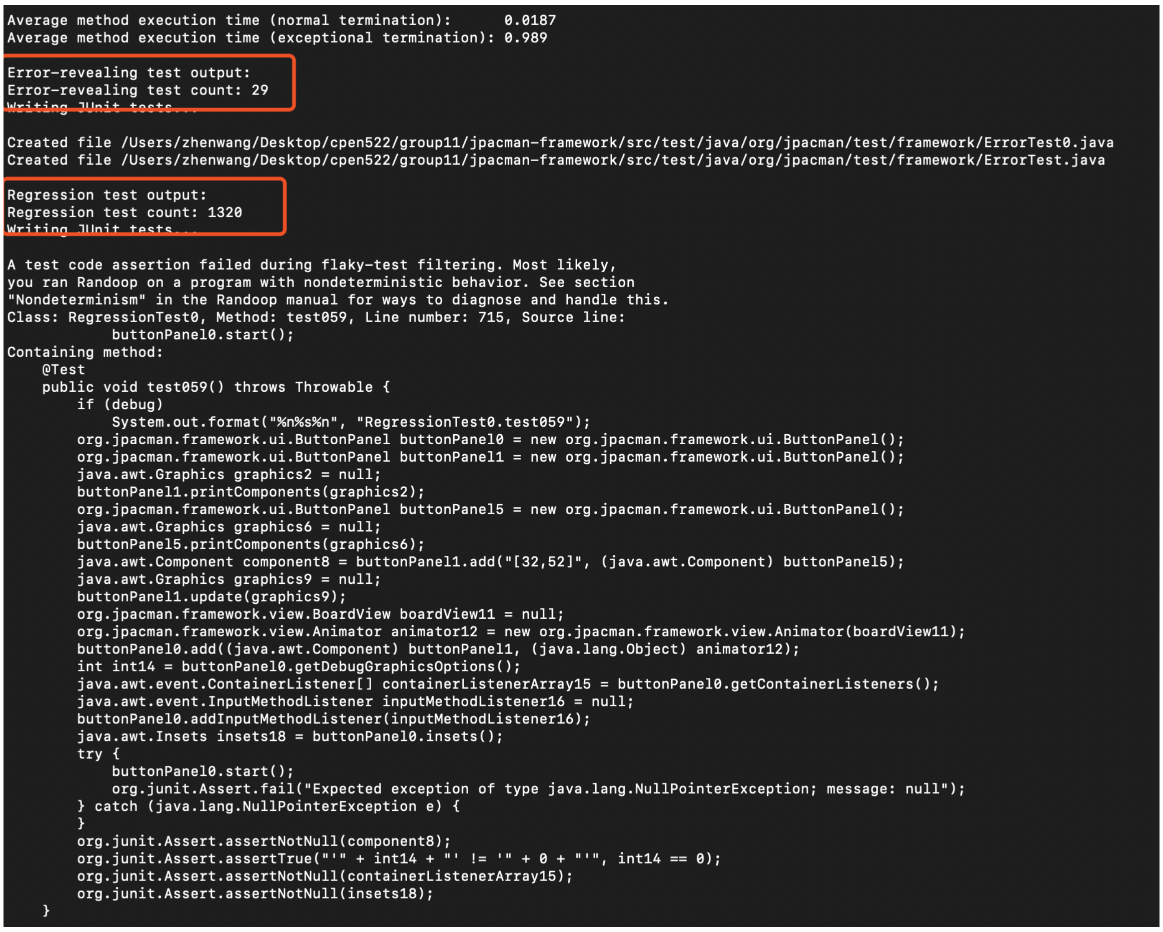
Randoop

Time: The class list is set to include all the classes in JPacman shown as below.



```
org.jpacman.framework.factory.DefaultGameFactory
org.jpacman.framework.factory.FactoryException
org.jpacman.framework.factory.IGameFactory
org.jpacman.framework.factory.MapParser
org.jpacman.framework.controller.AbstractGhostMover
org.jpacman.framework.controller.IController
org.jpacman.framework.controller.RandomGhostMover
org.jpacman.framework.model.Board
org.jpacman.framework.model.Direction
org.jpacman.framework.model.Food
org.jpacman.framework.model.Game
org.jpacman.framework.model.IBoardInspector
org.jpacman.framework.model.IGameInteractor
org.jpacman.framework.model.IPointInspector
org.jpacman.framework.model.Level
org.jpacman.framework.model.Player
org.jpacman.framework.model.PointManager
org.jpacman.framework.model.Sprite
org.jpacman.framework.model.Tile
org.jpacman.framework.model.Wall
org.jpacman.framework.ui.ButtonPanel
org.jpacman.framework.ui.IDisposable
org.jpacman.framework.ui.MainUI
org.jpacman.framework.ui.IPacmanInteraction
org.jpacman.framework.ui.PacmanInteraction
org.jpacman.framework.ui.PacmanKeyListener
org.jpacman.framework.ui.PointsPanel
org.jpacman.framework.view.Animator
org.jpacman.framework.view.BoardView
org.jpacman.framework.view.ImageLoader
```

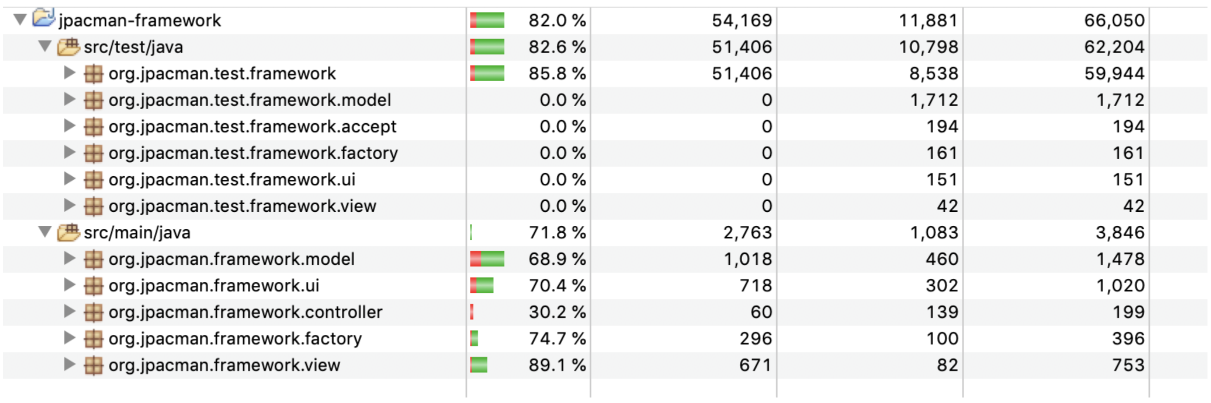
By setting the time limit to 1 minute, it generates 29 error-revealing tests and 1320 regression tests shown as below.



```
Average method execution time (normal termination):      0.0187
Average method execution time (exceptional termination): 0.989

Error-revealing test output:
Error-revealing test count: 29
Writing JUnit tests...

Created file /Users/zhenwang/Desktop/cpen522/group11/jpacman-framework/src/test/java/org/jpacman/test/framework/ErrorTest0.java
Created file /Users/zhenwang/Desktop/cpen522/group11/jpacman-framework/src/test/java/org/jpacman/test/framework/ErrorTest.java

Regression test output:
Regression test count: 1320
Writing JUnit tests...

A test code assertion failed during flaky-test filtering. Most likely,
you ran Randoop on a program with nondeterministic behavior. See section
"Nondeterminism" in the Randoop manual for ways to diagnose and handle this.
Class: RegressionTest0, Method: test059, Line number: 715, Source line:
            buttonPanel0.start();
Containing method:
    @Test
    public void test059() throws Throwable {
        if (debug)
            System.out.format("%n%s%n", "RegressionTest0.test059");
        org.jpacman.framework.ui.ButtonPanel buttonPanel0 = new org.jpacman.framework.ui.ButtonPanel();
        org.jpacman.framework.ui.ButtonPanel buttonPanel1 = new org.jpacman.framework.ui.ButtonPanel();
        java.awt.Graphics graphics2 = null;
        buttonPanel1.printComponents(graphics2);
        org.jpacman.framework.ui.ButtonPanel buttonPanel5 = new org.jpacman.framework.ui.ButtonPanel();
        java.awt.Graphics graphics6 = null;
        buttonPanel5.printComponents(graphics6);
        java.awt.Component component8 = buttonPanel1.add("[32,52]", (java.awt.Component) buttonPanel5);
        java.awt.Graphics graphics9 = null;
        buttonPanel1.update(graphics9);
        org.jpacman.framework.view.BoardView boardView11 = null;
        org.jpacman.framework.view.Animator animator12 = new org.jpacman.framework.view.Animator(boardView11);
        buttonPanel0.add((java.awt.Component) buttonPanel1, (java.lang.Object) animator12);
        int int14 = buttonPanel0.getDebugGraphicsOptions();
        java.awt.event.ContainerListener[] containerListenerArray15 = buttonPanel0.getContainerListeners();
        java.awt.event.InputMethodListener inputMethodListener16 = null;
        buttonPanel0.addInputMethodListener(inputMethodListener16);
        java.awt.Insets insets18 = buttonPanel0.insets();
        try {
            buttonPanel0.start();
            org.junit.Assert.fail("Expected exception of type java.lang.NullPointerException; message: null");
        } catch (java.lang.NullPointerException e) {
        }
        org.junit.Assert.assertNotNull(component8);
        org.junit.Assert.assertTrue("'" + int14 + "' != '" + 0 + "'", int14 == 0);
        org.junit.Assert.assertNotNull(containerListenerArray15);
        org.junit.Assert.assertNotNull(insets18);
    }
```

Coverage: The generated tests are saved as .java file with 29 tests for error-revealing test file and 500 for one regression test file. They are put into the JPacman project and the jar packages of JUnit and Hamcrest are put imported to the class path of eclipse.

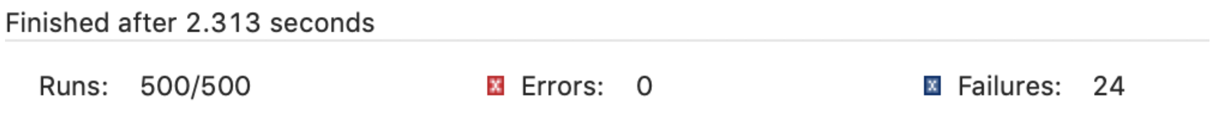
By running the tests of error-revealing and regression test together, the coverage is as shown below.



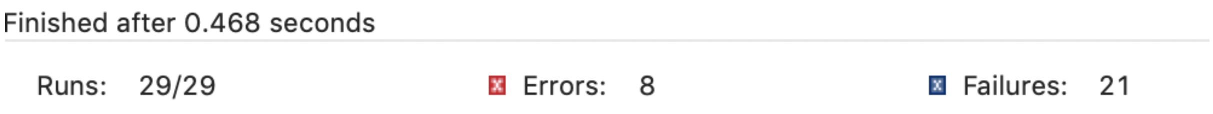
jpacman-framework	82.0 %	54,169	11,881	66,050
src/test/java	82.6 %	51,406	10,798	62,204
org.jpacman.test.framework	85.8 %	51,406	8,538	59,944
org.jpacman.test.framework.model	0.0 %	0	1,712	1,712
org.jpacman.test.framework.accept	0.0 %	0	194	194
org.jpacman.test.framework.factory	0.0 %	0	161	161
org.jpacman.test.framework.ui	0.0 %	0	151	151
org.jpacman.test.framework.view	0.0 %	0	42	42
src/main/java	71.8 %	2,763	1,083	3,846
org.jpacman.framework.model	68.9 %	1,018	460	1,478
org.jpacman.framework.ui	70.4 %	718	302	1,020
org.jpacman.framework.controller	30.2 %	60	139	199
org.jpacman.framework.factory	74.7 %	296	100	396
org.jpacman.framework.view	89.1 %	671	82	753

The coverage for the source code is 71.8% which is not bad but not very good, especially for the controller, the coverage rate is only 30.2% and for model which contribute the most codes, the coverage rate is only 68.9%. Also a lot of failures are found during the process which are all caused by nullPointerException.

FindBug: By running the error revealing test and the regression test separately, we can see the expected result that no pass for error revealing test and no error for the regression test.

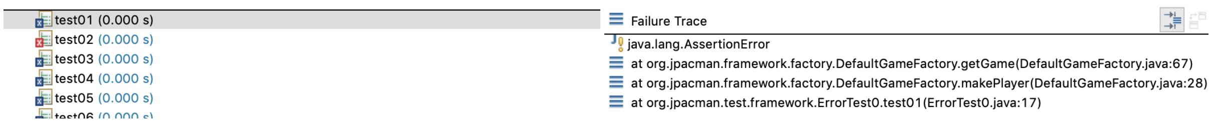


```
Finished after 2.313 seconds

Runs:  500/500        Errors:  0        Failures:  24
```



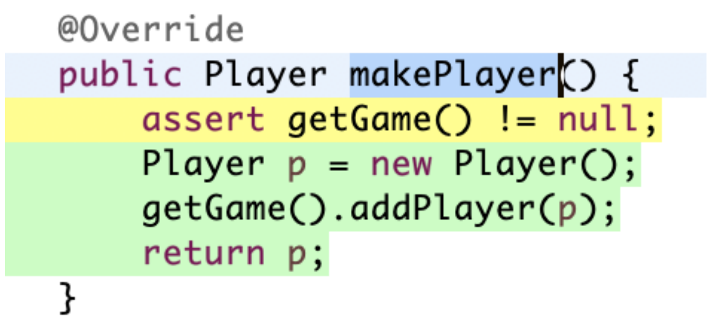
```
Finished after 0.468 seconds

Runs:  29/29        Errors:  8        Failures:  21
```

To dig deeper into why the failures occur, the example is given to explain.



```
test01 (0.000 s)
test02 (0.000 s)
test03 (0.000 s)
test04 (0.000 s)
test05 (0.000 s)
test06 (0.000 s)
```

```
Failure Trace
java.lang.AssertionError
at org.jpacman.framework.factory.DefaultGameFactory.getGame(DefaultGameFactory.java:67)
at org.jpacman.framework.factory.DefaultGameFactory.makePlayer(DefaultGameFactory.java:28)
at org.jpacman.test.framework.ErrorTest0.test01(ErrorTest0.java:17)
```

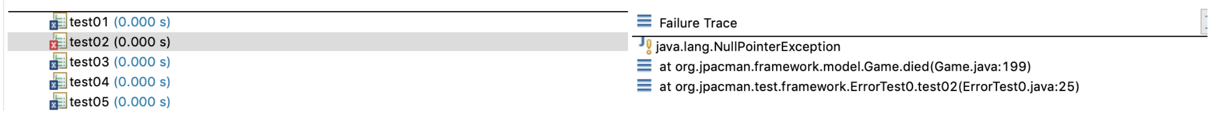


```java
@Override
public Player makePlayer() {
    assert getGame() != null;
    Player p = new Player();
    getGame().addPlayer(p);
    return p;
}
```

Obviously, it is caused by the assertions in the source code.

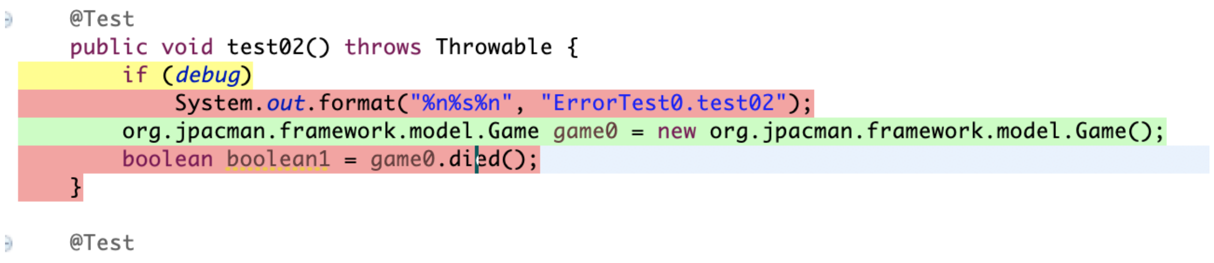
It definitely can find errors with error-revealing test.

For the errors it found, the example is shown below to analyze the reason.

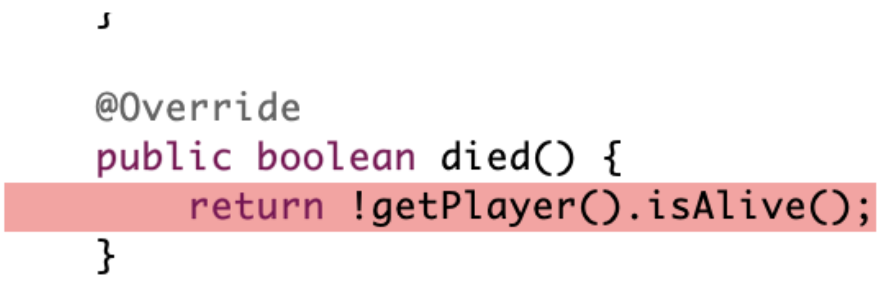


```
test01 (0.000 s)
test02 (0.000 s)
test03 (0.000 s)
test04 (0.000 s)
test05 (0.000 s)
```

```
Failure Trace
java.lang.NullPointerException
at org.jpacman.framework.model.Game.died(Game.java:199)
at org.jpacman.test.framework.ErrorTest0.test02(ErrorTest0.java:25)
```



```java
@Test
public void test02() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "ErrorTest0.test02");
    org.jpacman.framework.model.Game game0 = new org.jpacman.framework.model.Game();
    boolean boolean1 = game0.died();
}

@Test
```



```java
}

@Override
public boolean died() {
    return !getPlayer().isAlive();
}
```

Test02 tests the game.died() method, but there is no player created and getPlayer() method response with a reference referring to no object. The reference points to an address with java default null, and null do not have isAlive() method so the exception is raised and the failure is caused.

Yes, it is easy to use. The users just need to set the arguments of the command line properly according to the document and the error-revealing test and regression test will be generated.

Limitations:

It does have a decent coverage which is shown in the previous questions, but the coverage is not high enough compared to what we do to use manually created test suites in the previous assignments. By analyzing the coverage in the source code, the most obvious reason that leads to low coverage is that it does not cover all the paths considering conditioning.

Also according to the previous question, the error-revealing test does find bugs (or errors). But under complicated implementation conditions provided in the source code, most of the nullPointerException will never occur in real runtime.

Improvements:

Improving the coverage should be possible. If it can integrate randoop with other systematic testing method properly or provide user-friendly methods for people to use it together with other popular systematic testing tools, the coverage may get improved.