TEST PLAN

Flappy Bird Game

Abstract

This document provides an overview of Flappy Bird Desktop Game and the product test strategy, a list of testing deliverables and plan for development

Akiba Amrin
[amrin.akiba@northsouth.edu]

Md Shihab Uddin
[shihab.uddin@northsouth.edu]

Table of Contents

INTRODUCTION

The Test Plan is designed to prescribe the scope, approach, resources, and schedule of all testing activities of Flappy Bird.

The plan identify the items to be tested, the features to be tested, the types of testing to be performed, the personnel responsible for testing, the resources and schedule required to complete testing, and the risks associated with the plan.

1 TEST STRATEGY

1.1 Project Overview:

Flappy Bird is a side-scrolling Java 2D retro style graphics. The objective was to direct a flying bird, named "Faby", who moves continuously to the right, between sets of Mario-like pipes. If the player touches the pipes, they lose. Faby briefly flaps upward each time if the player presses up key from the keyboard; if up key is not pressed, Faby falls because of gravity, each pair of pipes that he navigates between earns the player a single point. And Game is over if the bird touches the ground or any wall. This game was developed in Java as our CSE215 final project in 2015. In CSE427 Project, our main goal is to develop the test cases especially using input space partitioning and graph partitioning and complete test suite using Java JUnit and SikuliX that covers unit, integration, functional and Graphical User Interface Tests. The purpose of this project is to fully implement all test design concept learned in CSE427: Software Testing & Quality Assurance course.

1.2 Scope of Testing

1.2.1 Feature to be tested

All the feature of Flappy Bird Game which were defined in software requirement specs are need to tested

Module	Applicable Rules	Description
BirdFlap	User	This module provides the bird with the ability to flap.

Component	Game	All of the components is drawn with the help of this module.
GamePanel	User	This module is responsible for graphics and movement of them.
GameSound	User	This module provides audio feedback of the game.

1.1.2 Feature not to be tested

These feature are not be tested because they are not included in the software requirement specs

- ❏ Hardware Interfaces
- ❏ Security
- ❏ Performance

1.2 Test Type

In this project, we developed the complete test suite with help JUnit. Later, SikuliX API was integrated with JUnit for GUI testing. The aspects included in the test suite are:

- ❏ **Unit Testing:** Each method of class Component, GamePanel, and GameSound was tested with respect to unit testing methodology.
- ❏ **Integration Testing:** Individual units tested in the previously mentioned class were combined together and tested as a group.
- ❏ **Functional Testing:** Functional Testing of this game was also tested by testing the functionality of each method UI.
- ❏ **GUI Testing:** GamePanel is mainly responsible for GUI. The game GUI was tested with SikuliX API.

1.3 Used Tools/Frameworks:

The framework '***JUnit 5***' from maven is responsible for structuring the test cases, checking the outputs and the tests themselves. Roughly, comparing expected values with values returned by the software, informing us whether the tests passed or failed. '***JUnit Params***' was used for parameterized tests, JUnit Theory is deprecated in latest version 'JUnit 5' so implementation of it is excluded. Meanwhile, '***SikuliX API***' of **Sikuli** was used for Graphical User Interface testing.

02 Input Space Partitioning

2.1 Component Class:

List of all input variables tested with ISP including variables that are covered in Graph partitioning:

- `private int x;` `// X Coordinate of Screen`
- `private int y;` `// Y Coordinate of Screen`
- `private int width, height;` `// Tested in Graph Partition`
- `private String imagePath;` `// Tested in Graph Partition`

Test cases for width, height and imagePath using input space partition is generated only for demonstration purpose to show that graph partition is more feasible. Test cases for width, height and image using ISP is not used in the JUnit test code. 'Graph partition' is used for in-depth test design for these three. Therefore, test cases for these variables are moved to 'Graph Partition'.

Characteristics of input variables defined by applying the domain knowledge of screen components. List below is characteristics of each input variable:

- `int x` can be any valid negative or 0 or any positive integer.
- `int y` can be any valid negative or 0 or any positive integer.
- `int width` can be any valid positive number starting from 0.
- `int height` can be any valid positive number starting from 0.
- `int imagePath` can be null or any valid image file path.

Blocks of the characteristics for each state variable is defined by partitioning them. Tables below shows blocks defined for each characteristics.

Partition	b1	b2	b3
Q1: Value of `int x`	negative	0	positive

Partition	b1	b2	b3
Q1: Value of `int y`	negative	0	positive

Partition	b1	b2
Q1: Value of `int width`	0	positive

Partition	b1	b2
Q1: Value of `int height`	0	positive

Partition	b1	b2
Q1: Whether `imagePath` is null or not.	true	false

After defining the block of characteristics for each input variables, random possible values for each them are chosen. Tables below shows all values picked for all input variables.

Parameter	b1	b2	b3
`x`	-110	0	200

Parameter	b1	b2	b3
`y`	-220	0	200

Parameter	b1	b2
`width`	0	200

Parameter	b1	b2
`height`	0	200

Parameter	b1	b2
`imagePath`	“images//background.png”	“invalid”

From these tables, input space partitioning test cases are developed for each block of each variables. As it was mentioned earlier, test cases for width, height, and imagePath variables are

not used in ISP since Graph partition was used to generate more test in-depth than ISP. The list of test method signature with parameterized test using input space partitioning are given below:

```
@ParameterizedTest
@MethodSource("providePartitionedValueForSetXTest")
void setXTest(int input, int expected) {
// Test For X
}
private                    static                    Stream<Arguments>
providePartitionedValueForSetXTest(){
   return Stream.of(
           Arguments.of(200, 200),
           Arguments.of(0, 0),
           Arguments.of(-110, -110)
   );
}

@ParameterizedTest
@MethodSource("providePartitionedValueForSetYTest")
void setYTest(int input, int expected) {
// Test For Y
}
private                    static                    Stream<Arguments>
providePartitionedValueForSetYTest(){
   return Stream.of(
           Arguments.of(200, 200),
           Arguments.of(0, 0),
           Arguments.of(-220, -220)
   );
}

@Test
void getXTest() {
   ComponentTestObject.setX(100);
   assertEquals(100, ComponentTestObject.getX());
}

@Test
void getYTest() {
   ComponentTestObject.setY(200);
   assertEquals(200, ComponentTestObject.getY());
}
```

2.2 GamePanel Class:

List of all input variables tested with ISP including variables that are covered in Graph partitioning:

- ➔ `Component bird.y;  // Implicit State Variable`
- ➔ `Double points;`
- ➔ `Double amount;`
- ➔ `Boolean gameOver;`
- ➔ `Boolean started;`

'Graph partition' was used for in-depth Test Design for variable background, bird, walls, and ground. Therefore, Test cases for these four variables are generated from 'Graph Partition'.

Characteristics of input variables defined by applying the domain knowledge of screen components. List below is characteristics of each input variable:

- ➔ `Component bird.y` can be any valid negative or 0 or any positive integer.
- ➔ `Double points` can be any valid positive floating point number starting from 0.
- ➔ `Double amount` can be any valid negative or 0 or any positive floating point number.
- ➔ `Boolean gameOver` can be either true or false.
- ➔ `Boolean started` can be either true or false.

Blocks of the characteristics for each state variable is defined by partitioning them. Tables below shows blocks defined for each characteristics.

Characteristics	b1	b2	b3
Q1: Value of `Component bird.y`	negative	0	positive

Characteristics	b2	b3
Q1: Value of `double points`	0	positive

Characteristics	b1	b2	b3
Q1: Value of `double amount`	negative	0	positive

Characteristics	b1	b2
Q1: Whether `gameOver` is true or false	true	false

Characteristics	b1	b2
Q1: Whether `started` is true or false	true	false

After defining the block of characteristics for each input variables, random possible values for each blocks are chosen. Tables below shows all values picked for all blocks from b1 to b3.

Parameter	b1	b2	b3
`bird.y`	-10	0	10

Parameter	b1	b2
`points`	0.0	1.0

Parameter	b1	b2	b3
`amount`	-220.0	0	200

Parameter	b1	b2
`gameOver`	true	false

Parameter	b1	b2
`started`	true	false

From these tables, input space partitioning test cases are developed for each block of each variables. As it mentioned earlier, test cases for some variables are not used in ISP since Graph partition was used to generate more tests for them.

Test cases by ISP was developed for each and every method that uses these state variables. The list of test method signature with parameterized test using input space partitioning are given below:

Test Methods For bird.y:

- `gravityPullWithPositiveStateVariableTest(){..}`
- `gravityPullWithZeroStateVariableTest() {..}`
- `gravityPullWithNegativeStateVariableTest() {..}`
- `flapWithPartitionOneTest() {..}`
- `flapWithPartitionTwoTest() {..}`
- `flapWithPartitionThreeTest() {..}`

Test Methods For `points` and `amount`:

```
@ParameterizedTest
@MethodSource("providePartitionedValueForIncrementScoreTest")
void incrementScorePositiveTest(double input, double expected) {
   assertEquals(expected+1.0, gamePanelTestObject.incrementScore(1.0,
input), 1);
}

@ParameterizedTest
@MethodSource("providePartitionedValueForIncrementScoreTest")
void incrementScoreZeroTest(double input, double expected) {
   assertEquals(expected, gamePanelTestObject.incrementScore(0.0,
input), 1);
}
```

Test Methods For gameOver:

```
@ParameterizedTest
@MethodSource("providePartitionedValueForIsGameOverTest")
void isGameOverTest(boolean input, boolean expected) {
   gamePanelTestObject.setGameOver(input);
   assertEquals(expected, gamePanelTestObject.isGameOver());
}

setGameOverTrueTest(){..}
setGameOverFalseTest(){..}
```

Test Methods For started:

```
@ParameterizedTest
@MethodSource("providePartitionedValueForIsStartedAndSetStartedCombin
edTest")
void isStartedAndSetStartedCombinedTest(boolean input, boolean
expected) {
   gamePanelTestObject.setStarted(input);
   assertEquals(expected, gamePanelTestObject.isStarted());
}
```

03 Graph partitioning

3.1 GamePanel Class:

The method `GamePanel.scrollWalls()` is tested by Graph partition. The figure below shows the graph of the method `GamePanel.scrollWalls()`

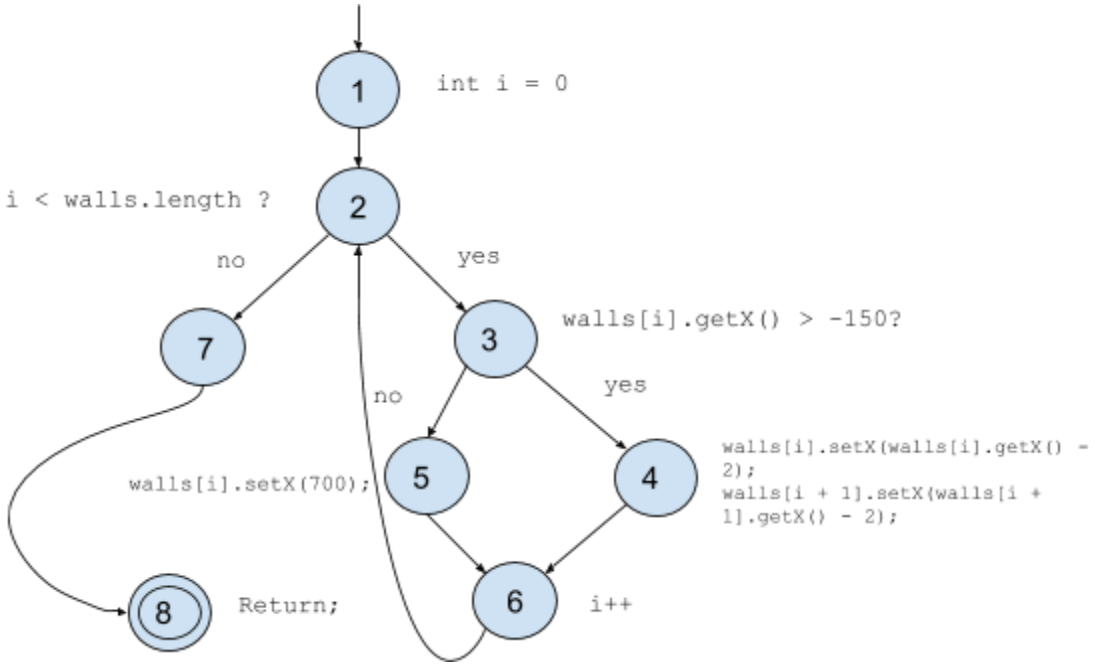

Figure: Graph of `scrollWalls()` Method

Test cases were designed using Edge-Pair, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 4, 6, 7, 8), (1, 2, 3, 5, 6, 7, 8) }

Both of these paths were visited by the test methods. Random values were chosen for the state variables to visit both paths. Two test cases were developed for these paths.

The method `GamePanel.scrollBG()` is tested by Graph partition. The figure below shows the graph of the method `GamePanel.scrollBG()`

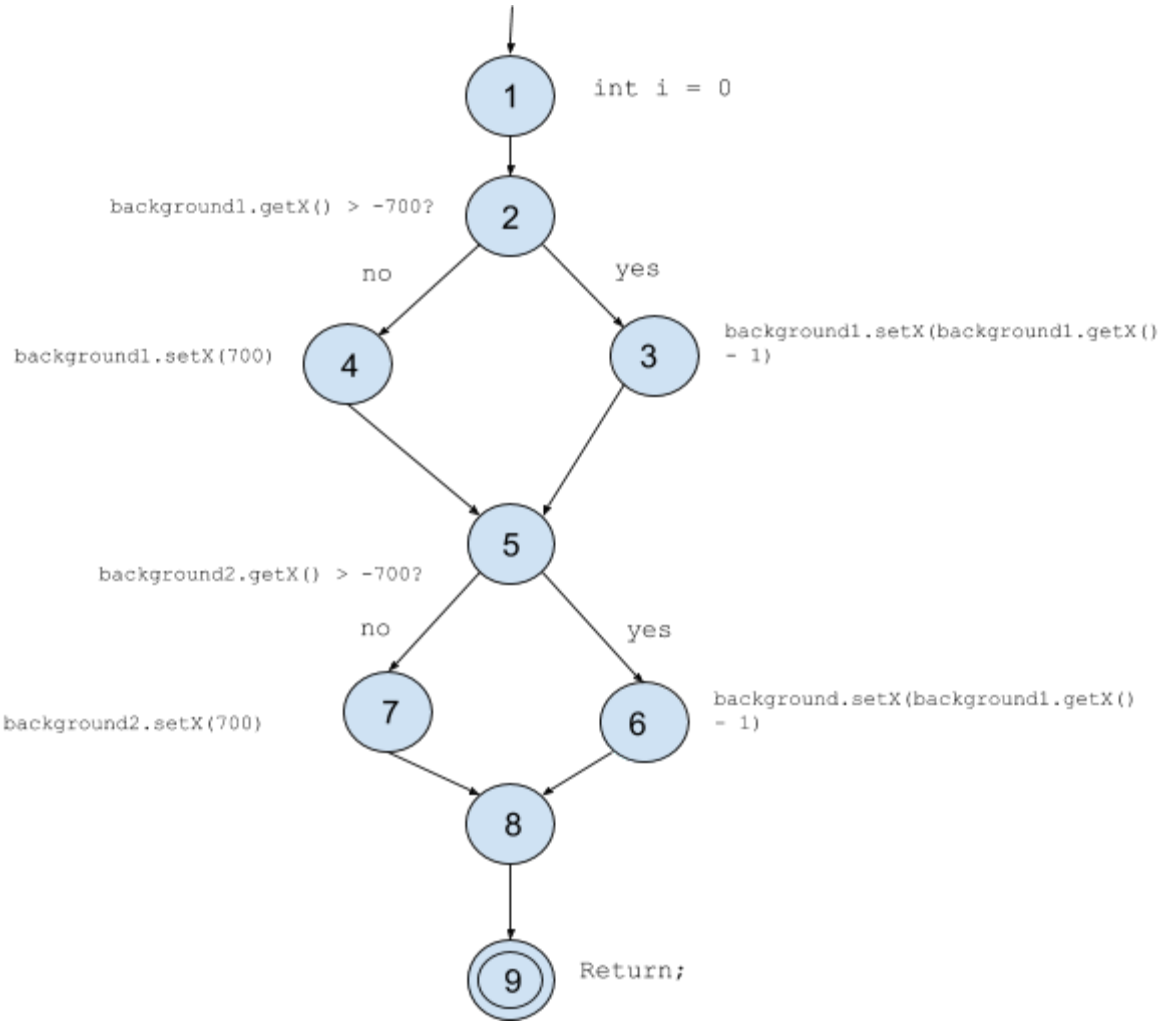

Figure: Graph of `scrollBG()` Method

Test cases were designed using Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 5, 6, 8, 9), (1, 2, 3, 5, 7, 8, 9), (1, 2, 3, 5, 6, 8, 9), (1, 2, 4, 5, 6, 7, 8, 9) }

Both of these paths were visited by the test methods. Random values were chosen for the state variables to visit all paths. Four test cases were developed for these paths.

The method `GamePanel.detectCollision()` is tested by Graph partition. The figure below shows the graph of the method `GamePanel.detectCollision()`

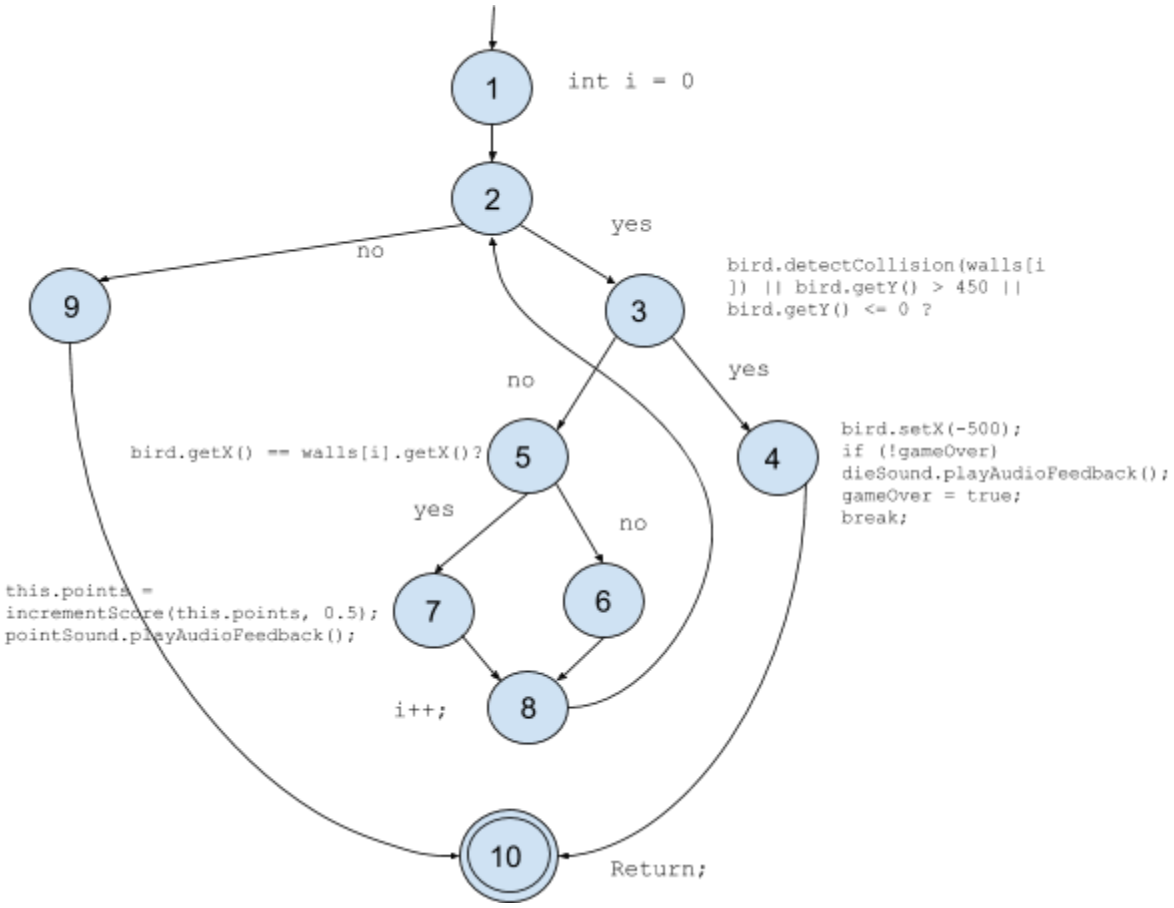

Figure: Graph of `detectCollision()` Method

Test cases were designed using Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph. Some side trip was included to completely test the method.

TR = { (1, 2, 3, 5, 7, 8, 2, 3, 5, 6, 8, 9, 10), (1, 2, 3, 5, 6, 8, 2, 3, 4, 10) }

Both of these paths were visited by the test methods. Random values were chosen for the state variables to visit both paths. Two test cases were developed for these paths.

The method `GamePanel.scrollGround()` is tested by Graph partition. The figure below shows the graph of the method `GamePanel.scrollGround()`

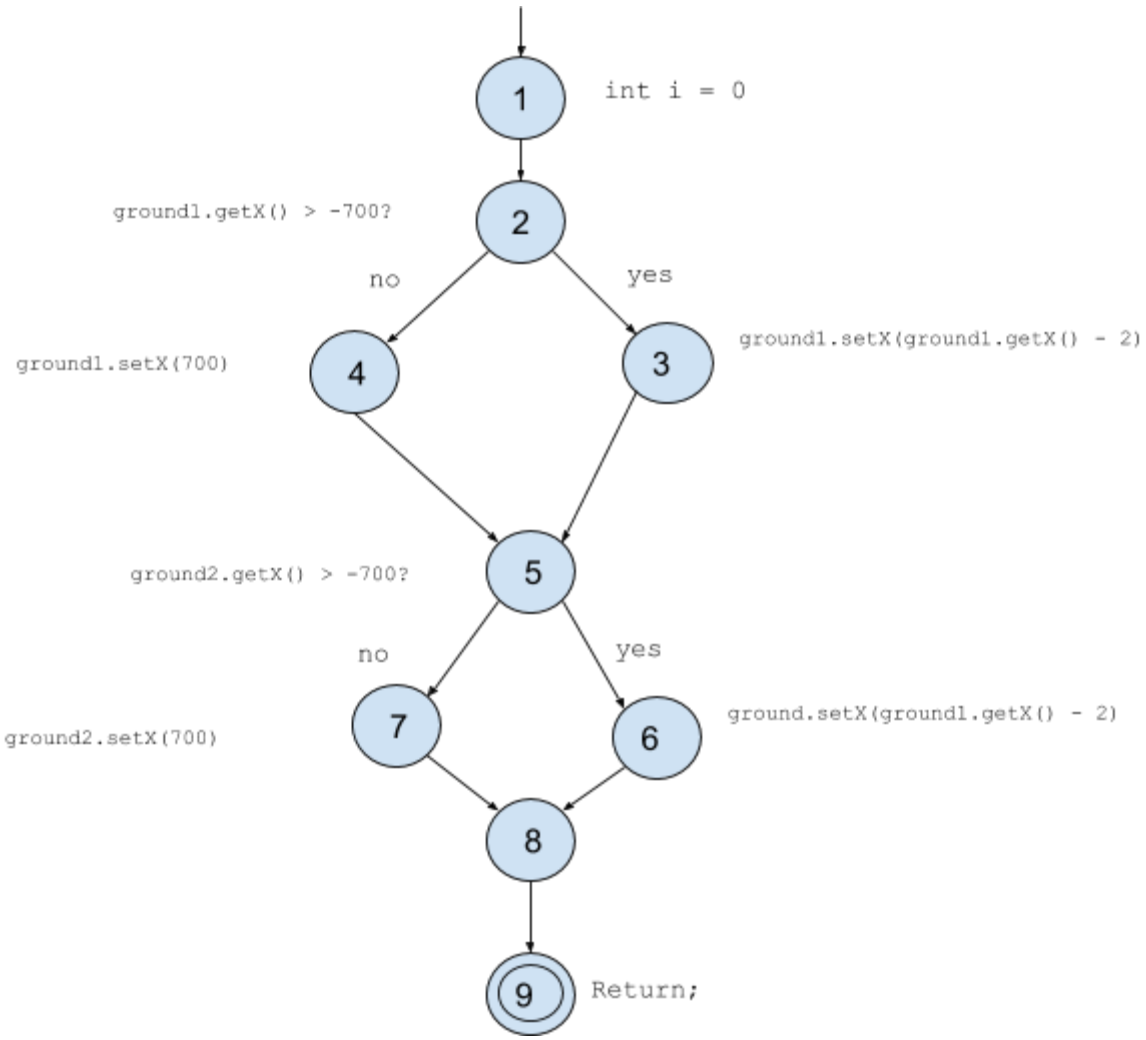

Figure: Graph of `scrollGround()` Method

Test cases were designed using Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 5, 6, 8, 9), (1, 2, 3, 5, 7, 8, 9), (1, 2, 3, 5, 6, 8, 9), (1, 2, 4, 5, 6, 7, 8, 9) }

Both of these paths were visited by the test methods. Random values were chosen for the state variables to visit all paths. Four test cases were developed for these paths.

3.2 Component Class:

The method `Component.detectCollision()` is tested by Graph partition. The figure below shows the graph of the method `Component.detectCollision()`

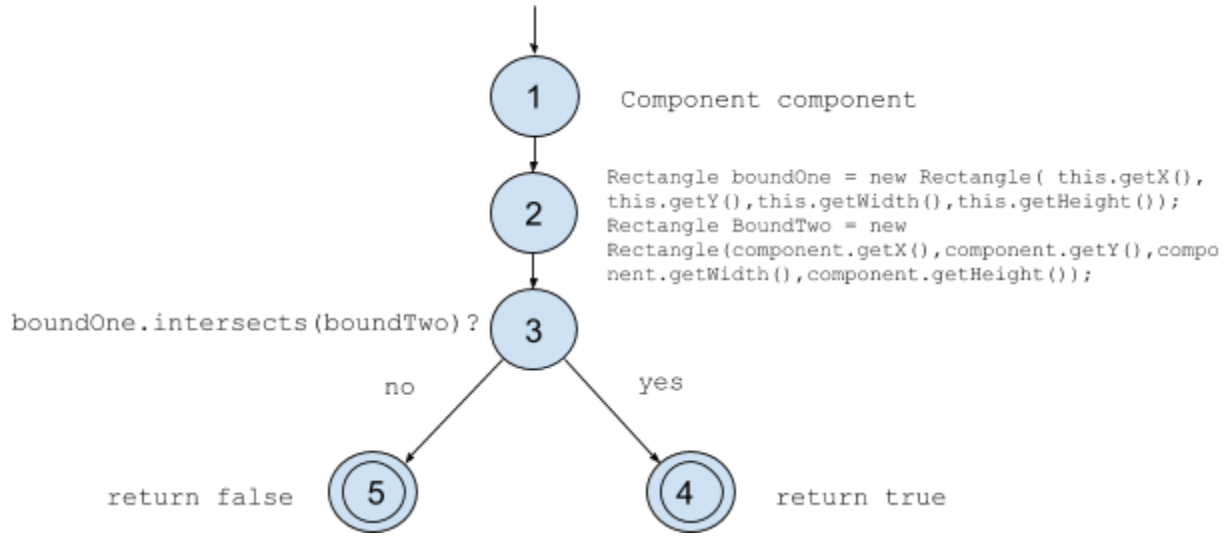

Figure: Graph of `detectCollision()` Method

Test cases were designed using Prime-Path Coverage, Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 4), (1, 2, 3, 5) }

Both of these paths were visited by the test methods. Random values were chosen for the state variables of the method to visit all paths. In total, two test cases were developed for these two paths.

The method `Component.setWidth()` is tested by Graph partition. The figure below shows the graph of the method `Component.setWidth()`

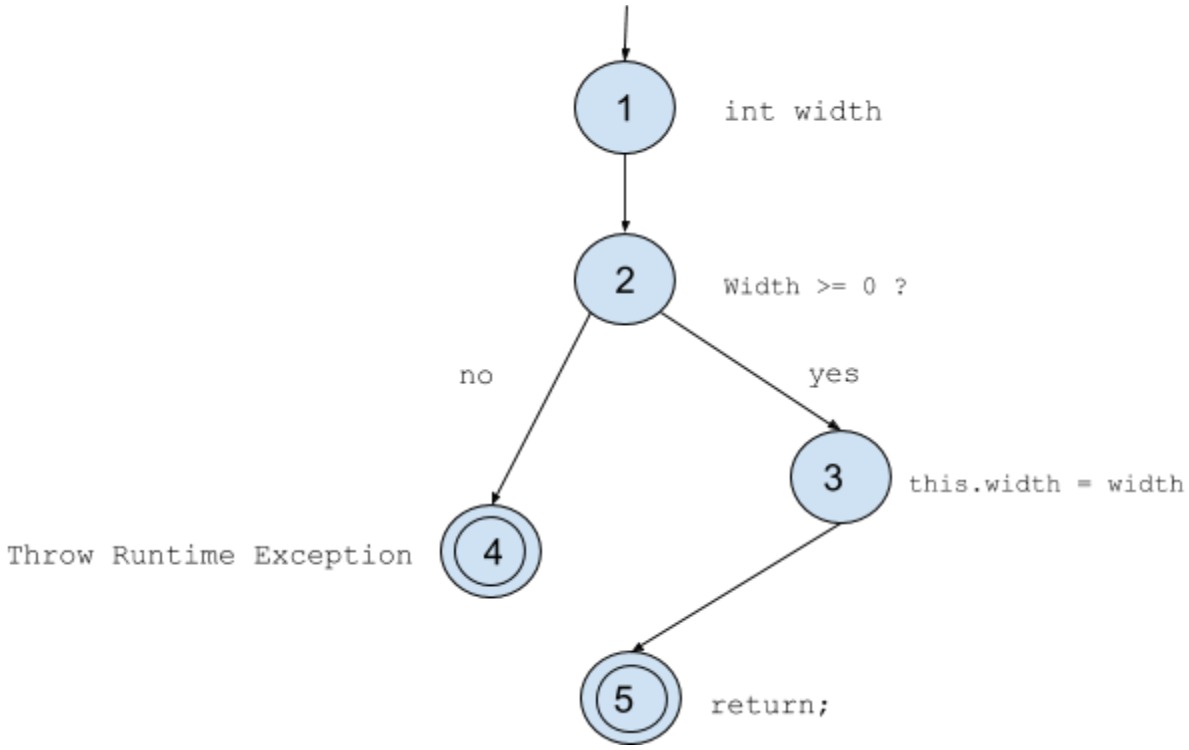

Figure: Graph of `setWidth()` Method

Test cases were designed using Prime-Path Coverage, Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 5), (1, 2, 4) }

Both of these paths were visited by the test methods. Random values were chosen for the input variables of the method to visit all paths. In total, two test cases were developed for these two paths. `Component.getWidth()` is completely overlaps with these test cases. Therefore, redundant test cases were excluded.

The method `Component.setHeight()` is tested by Graph partition. The figure below shows the graph of the method `Component.setHeight()`

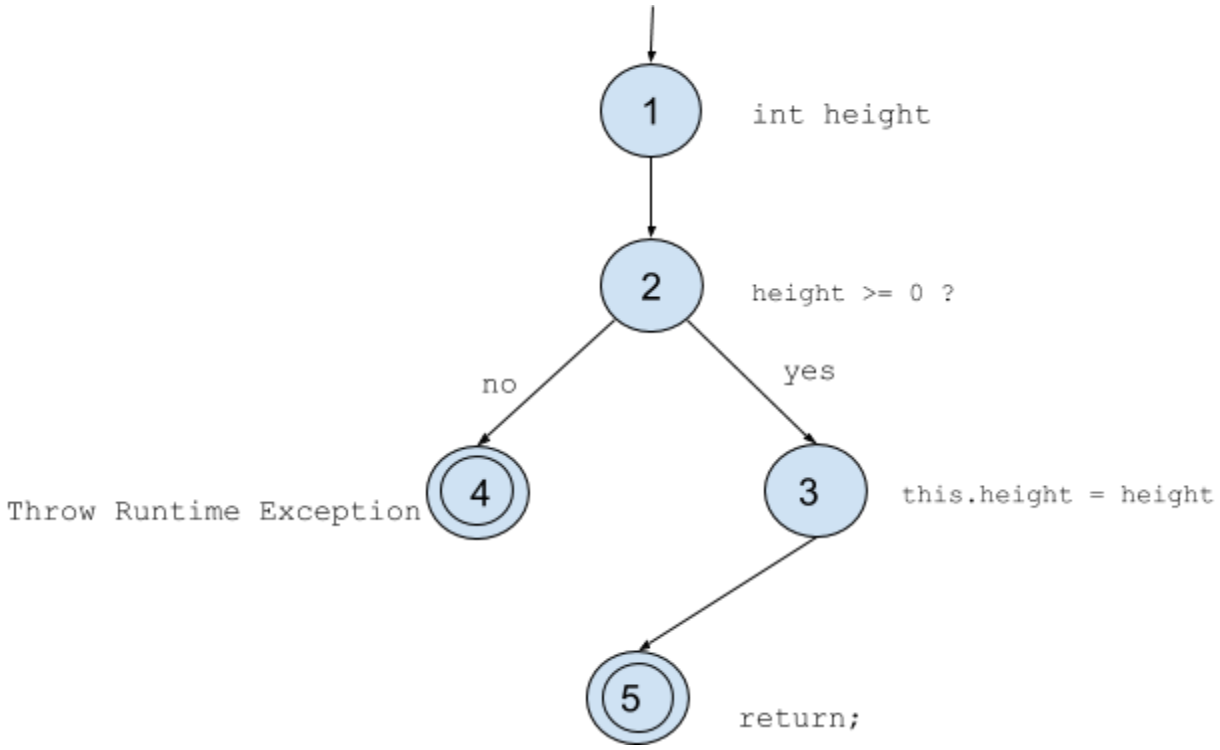

Figure: Graph of `setHeight()` Method

Test cases were designed using Prime-Path Coverage, Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 5), (1, 2, 4) }

Both of these paths were visited by the test methods. Random values were chosen for the input variables of the method to visit all paths. In total, two test cases were developed for these two paths. `Component.getHeight()` is completely overlaps with these test cases. Therefore, redundant test cases were excluded.

The method `Component.setImagePath()` is tested by Graph partition. The figure below shows the graph of the method `Component.setImagePath()`

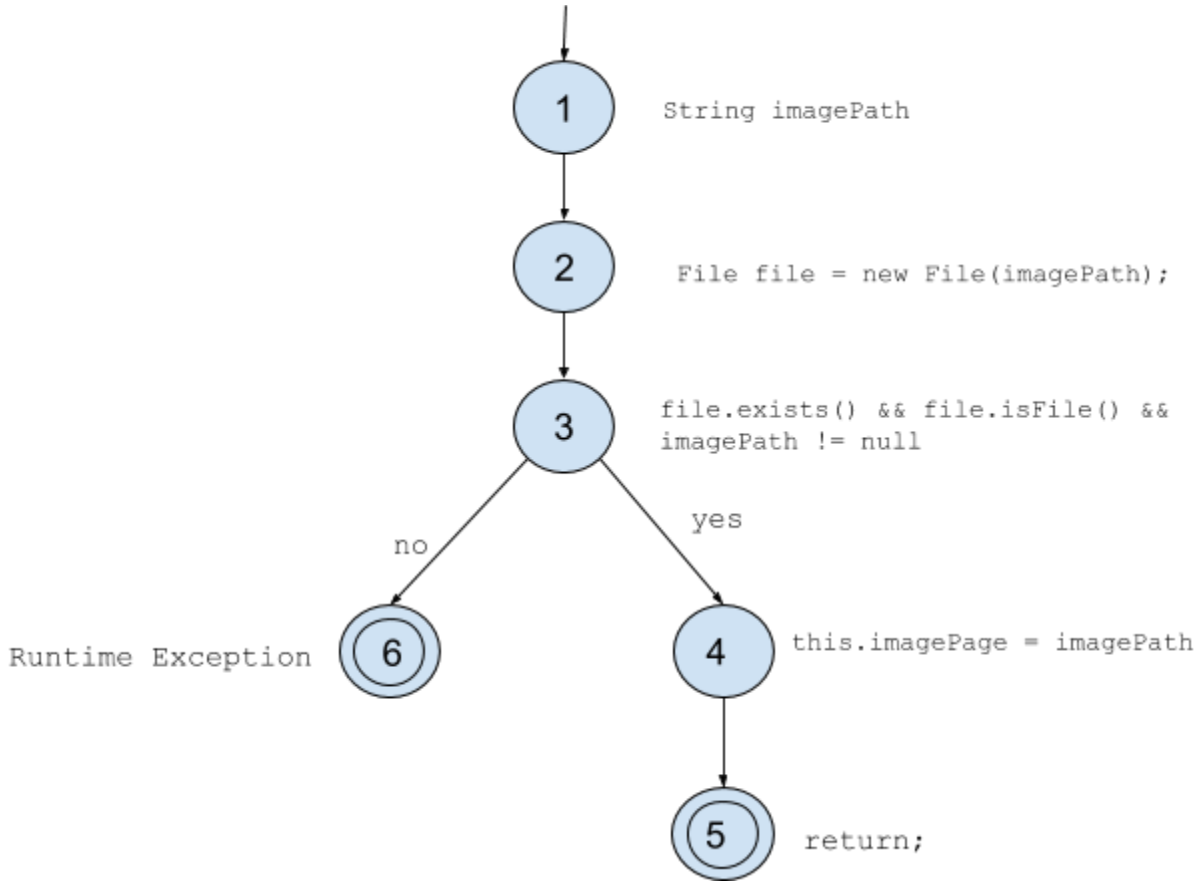

Figure: Graph of `setImagePath()` Method

Test cases were designed using Prime-Path Coverage, Edge-Pair coverage, Edge coverage and Node coverage from this graph. To cover all the nodes and edges and edge-pairs two test paths was derived from the graph.

TR = { (1, 2, 3, 4, 5), (1, 2, 3, 6) }

Both of these paths were visited by the test methods. Random values were chosen for the input variables of the method to visit all paths. In total, two test cases were developed for these two paths. `Component.getImagePath()` is completely overlaps with these test cases. Therefore, redundant test cases were excluded.

3.3 GameSound Class:

The GameSound has one method which is tested by Graph partition. The figure below shows the graph of the method `GameSound.playAudioFeedback()`

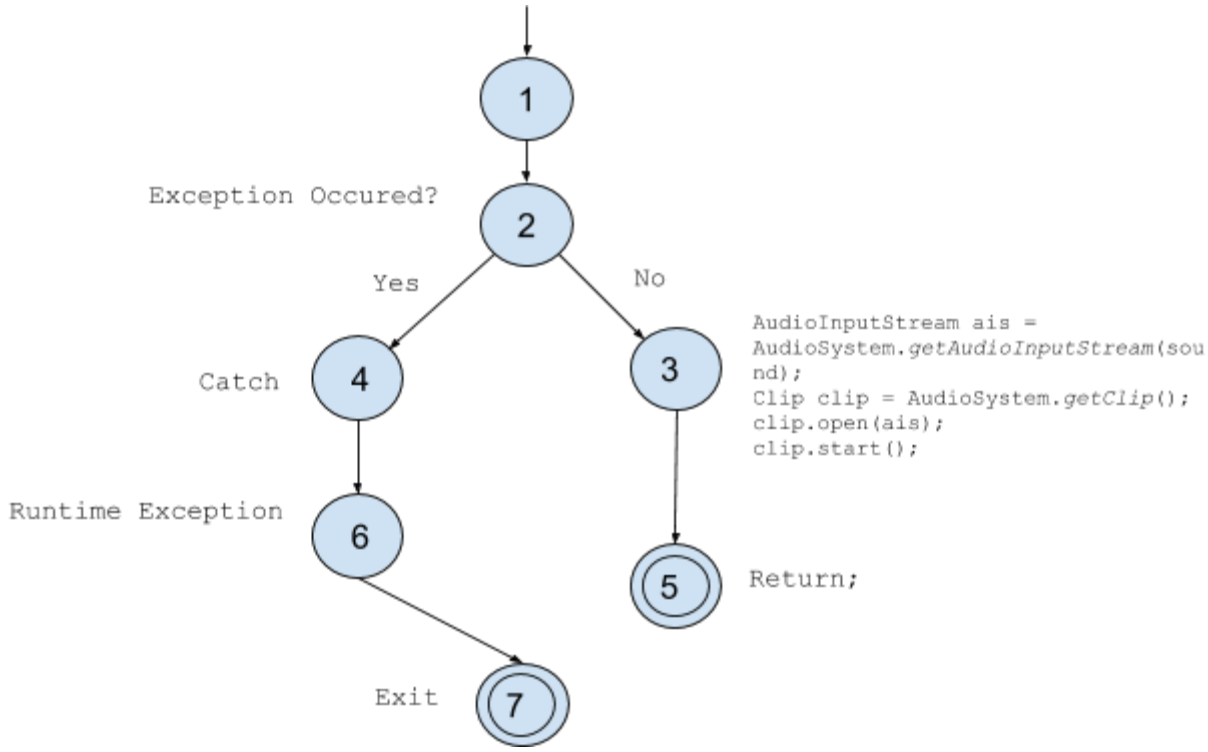

Figure: Graph of playAudioFeedback() Method

Test cases were designed for Prime Path, Edge and Node coverage from this graph. To cover all the nodes and edges two test paths was derived from the graph.

TR = { (1, 2, 4, 6, 7), (1, 2, 3, 5) }

Both of these paths were visited by the test method below. Random values were chosen for the state variables to visit both paths. Paramaterized test was used to generate test cases:

```
@Test
public void playAudioFeedbackTest() {
     assertEquals(expected, testObject.playAudioFeedback());
   }
}
```

```
@Test
public void playAudioFeedbackTestWithException() {
   Exception exception = assertThrows(RuntimeException.class, () ->
testObject.playAudioFeedback());
   assertEquals("Sound file loading error!", exception.getMessage());
}
```

04. Conclusion

In this project, we tested every possible aspects of the flappy bird game with the help of JUnit, Junit Params, SikuliX API. Test cases for every class and and every GUI object was designed and implemented while carefully removing redundant test cases. Finally, we developed a test suite combining all the unit tests, integration tests, functional tests and GUI test.