Mummy Maze Plus

Test Report

Group 10

Abhishek Singh Rathore
Jeff Grandt
William Montgomery
Zichen Wang

The Testing Process

The testing of Mummy Maze Plus was divided into multiple phases. While we would have preferred to implement JUnit testing during the implementation phase, we decided it would not be feasibly to write unit tests for all of the code. This meant that the implementation relied on the manual testing of code, which, though not ideal, resulted in an acceptable final product.

The first phase of testing outside of development was a usability test with an independent user. The goal of this test was to uncover issues related to the usability of the software. It was imperative that this phase occur shortly after initial development, because flaws in the usability could have a major impact on the code and therefore further testing. It makes no sense to complete rigorous testing on software that does not meet its usability goals.

For the second phase of testing, we chose four representative pieces of code, one from each team member. As a team, we completed an initial inspection of the code, with the aid of an Inspection Checklist. Any deficiencies in naming, style, or comments were immediately addressed. The checklist also included some checks for control flow, I/O defects, and computational defects. Any defects that were uncovered at this point were corrected before proceeding to the next phase.

In the next step, we combined black-box and white-box testing to test parts of the representative pieces of code. Our approach utilized equivalence testing, boundary testing, path testing, and state-based testing.

Finally, after defects were found, we met as a team to re-inspect the code to verify that the identified defects had been corrected. In the real world, we would expect that new defects would require new unit tests that identify the defects.

Since our implementation was rapid and we were continuously integrating code as it became available, we decided that instead of performing integration testing, we would move directly to

system testing. This was performed by developing use cases and running through the uses cases manually.

Requirements Traceability

Ideally, a software development project would be able to trace its requirements. However this was not realistic in this project. The requirements set out in the requirements document were not obtainable with the limited time and manpower that were used to develop the project. As such, we used the requirements document as a springboard for development instead of a rigorous document to abide by. In the real world, we would certainly work to maintain traceability with the requirements document.

Tested Items

As noted previously, we limited our testing to representative portions of code, one from each team member. These include:

- Animation Class (Jeffrey Grandt)
- Button Class (William Montgomery)
- ProgressBar Class (Abhishek Singh Rathore)
- GameSaved (Zichen Wang)

Testing Schedule

The software was tested in the following schedule:

Wednesday, November 20, 2013: Usability Testing

Monday, November 25, 2013: Initial Code Inspection

Wednesday, November 27, 2013 - Friday, November 29, 2013: Defect Resolution

Saturday, November 30, 2013 - Sunday, December, 1, 2013: Test Report Generation

Monday, December 2, 2013: Follow-Up Code Inspection

Test Recording Procedures

All tests were collaborative documented in a Google Docs document as the tests progressed. This allowed for collaborative testing as well as up to the minute results of the testing.

Hardware and Software Requirements

The tests require Java JRE version 1.7 or greater and Processing 2.0 installed. Processing 2.0 is supported on the following operating systems:

- Mac OS X 10.8
- Windows XP (latest service pack only)
- Windows 7 (32-bit and 64-bit)

- Ubuntu Linux 10.04 (32-bit)
- Ubuntu Linux 12.04 (64-bit)

Constraints

Time was severely limited during testing, so testing was not as thorough as would be expected on a commercial piece of software. The completion of the requirements document, along with the development of a presentation diverted much manpower away from the testing. We would have liked to be able to develop actual unit tests, instead of mocking out the tests, however this was not possible.

Testing Results

Usability Test Results

Our usability test results were generally very positive. Users had a positive impression of the game, and the play was very intuitive. There were several issues that were uncovered during this phase.

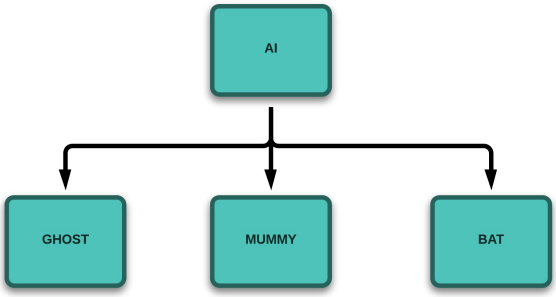
1. When adversaries collide, they disappear. This was unintuitive for the player as there was no indication of what happened.
2. In the main menu, the loading of saved games worked, but there was no indication of what level the saved game was on.
3. In the pause menu, when a player saved a game, there was no indication that the game was saved. We went ahead and implemented this code; it is evaluated below in the GameSaved class.It is not enough simply to run tests, the results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

Animation Class (Jeffrey Grandt) (Polymorphism Testing Example)

Original Code

```
public Boolean checkForItems(Coordinate newPosition) {
      for (int j = 0; j < items.size(); j++) {
             if (items.get(j).isABarrier()) {
               items.get(j).setCanWalkOver(false);

               for (int k = 0; k < ai.size(); k++) {
                   if (!ai.get(k).canWalkThroughWalls() &&
                        ai.get(k).position.equals(items.get(j).position)) {
                     ai.remove(k);
                     k--;
                   }
               }
             }
```

```
}
```



In this example, we present a test case of function employing polymorphism. The function checkForItems comes from class AI which is used to describe the behavior and states of different actors, a ghost, a bat, etc. Function checkForItems takes the position in terms of x and y as parameters, determines what type of that item is and send corresponding messages to the item receiver. Class Item is the superclass of the set of items defined in this game, class life, booby trap, and barrier switch are subclass of item for concrete item used in this game. We created and conducted polymorphism testing for this function to verify the compatibility among all subclasses.

Polymorphic Test Class for Actor

```
public Boolean checkForItems(Coordinate newPosition) {
for (int j = 0; j < items.size(); j++) {
      if (items.get(j).isABarrier()) {
        items.get(j).setCanWalkOver(false);

        for (int k = 0; k < ai.size(); k++) {
            Boolean canWalkThroughWalls = false;
            Boolean equalPosition = false;
            if ( ai.get(k).canWalkThroughWalls() instanceof Ghost) {
              Ghost ghost = ai.get(k);
              canWalkThroughWalls = ghost.canWalkThroughWalls();
              equalPosition = ghost.position.equals(items.get(j).position);
            }
            if ( ai.get(k).canWalkThroughWalls() instanceof Bat) {
              Bat bat = ai.get(k);
              canWalkThroughWalls = bat.canWalkThroughWalls();
              equalPosition = bat.position.equals(items.get(j).position);
            }
```

```
            if ( ai.get(k).canWalkThroughWalls() instanceof Mummy) {
              Mummy mummy = ai.get(k);
              canWalkThroughWalls = mummy.canWalkThroughWalls();
              equalPosition = mummy.position.equals(items.get(j).position);
            }

            if (!canWalkThroughWalls && equalPosition) {
              ai.remove(k);
              k--;
            }
          }
        }
      }
```

Code Checklist

1. Variable and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions? Yes
2. Are there variables with confusingly similar names? No
3. Is every variable properly initialized? Yes
4. Could any non-local variables be made local? No
5. Are there literal constants that should be named constants? No
6. Are there macros that should be constants? No
7. Are there variables that should be constants? No

2. Function Definition Defects (FD)

8. Are descriptive function names used in accord with naming conventions? Yes
9. Is every function parameter value checked before being used? No. The only parameter is a instance of class coordination which has two variable. Values os those variables should be checked before use ensuing the values are within acceptable ranges.
10. For every function: Does it return the correct value at every function return point? Yes

5. Comparison/Relational Defects (CR)

18. Are the comparison operators correct? Yes
19. Is each boolean expression correct? Yes
20. Are there improper and unnoticed side-effects of a comparison? No

6. Control Flow Defects (CF)

21. For each loop: Is the best choice of looping constructs used? Yes

22. Will all loops terminate? Yes

23. When there are multiple exits from a loop, is each exit necessary and handled properly? Yes

26. Is the nesting of loops and branches too deep, and is it correct? Not too deep. And is correct.

27. Can any nested if statements be converted into a switch statement? No

29. Does every function terminate? Yes

30. Are goto statements avoided? Yes

8. Module Interface Defects (MI)

37. Are the number, order, types, and values of parameters in every function call in agreement with the called function's declaration? Yes

38. Do the values in units agree (e.g., inches versus yards)? Yes

9. Comment Defects (CM)

39. Does every function, class, and file have an appropriate header comment? No

40. Does every variable or constant declaration have a comment? No

41. Is the underlying behavior of each function and class expressed in plain language? Yes

42. Is the header comment for each function and class consistent with the behavior of the function or class? No

43. Do the comments and code agree? No

44. Do the comments help in understanding the code? No

45. Are there enough comments in the code No

46. Are there too many comments in the code? No

11. Modularity Defects (MO)

50. Is there a low level of coupling between packages (classes)? No

51. Is there a high level of cohesion within each package? Yes

52. Is there duplicate code that could be replaced by a call to a function that provides the behavior of the duplicate code? Yes

12. Performance Defects (PE) [Optional]

54. Can better data structures or more efficient algorithms be used? Yes. Most of the matching and searching processing can be improved by introduce high efficiency algorithm. The produce of loading pictures into this game can be replaced by a lazy loading strategy, which only loads pictures when it is necessary rather than loading all the pictures once at the beginning of the

game.

55. Are logical tests arranged such that the often successful and inexpensive tests precede the more expensive and less frequently successful tests? Yes

56. Can the cost of recomputing a value be reduced by computing it once and storing the results? No.

57. Is every result that is computed and stored actually used? Yes

58. Can a computation be moved outside a loop? No

59. Are there tests within a loop that do not need to be done? No

60. Can a short loop be unrolled? No

61. Are there two loops operating on the same data that can be combined into one? No

Button Class (William Montgomery)

Original Code

```
/*!
 *  Based on code from: http://processing.org/examples/button.html
 */
public class Button {
  protected int buttonWidth;
  protected int buttonLength;
  protected int rectX, rectY;      // Position of square button
  protected String word;
  protected color rectColor, circleColor, baseColor;
  protected color rectHighlight, circleHighlight;
  protected color currentColor;
  protected boolean rectOver = false;
  protected int currentAction;
  protected int action;
  protected int nextAction;
  protected PImage bgImage;

  public Button(int startX, int startY, int startWidth, int startLength,
String startWord, int current, int next) {
    buttonWidth = startWidth;
    buttonLength = startLength;
    rectX = startX;
    rectY = startY;
    word = startWord;
    rectColor = color(0);
    rectHighlight = color(51);
```

```
    baseColor = color(102);
    currentColor = baseColor;
    currentAction = current;
    action = current;
    nextAction = next;
    bgImage = null;
  }

  public Button(int startX, int startY, PImage bgImage, int current, int next)
{
    buttonWidth = bgImage.width;
    buttonLength = bgImage.height;
    rectX = startX;
    rectY = startY;
    word = "";
    rectColor = color(0);
    rectHighlight = color(51);
    baseColor = color(102);
    currentColor = baseColor;
    currentAction = current;
    action = current;
    nextAction = next;
    this.bgImage = bgImage;
  }

  public void draw() {
    update(mouseX, mouseY);
    if (bgImage != null){
      image(bgImage, rectX, rectY);
    }
    else {
      if (rectOver) {
        fill(rectHighlight);
      }
      else {
        fill(rectColor);
      }
      stroke(255);
      rect(rectX, rectY, buttonWidth, buttonLength);
      textBox();
    }
    if (mousePressed && rectOver) {
      action = nextAction;
    }
  }

  public void reset(){
    action = currentAction;
```

```
  }

  public void update(int x, int y) {
    if ( overRect(rectX, rectY, buttonWidth, buttonLength) ) {
      rectOver = true;
    }
    else {
      rectOver = false;
    }
  }

  public void mousePressed() {
    if (rectOver) {
      currentColor = rectColor;
      action = nextAction;
    }
  }

  public boolean overRect(int x, int y, int width, int height) {
    if (mouseX >= x && mouseX <= x+width &&
      mouseY >= y && mouseY <= y+height) {
      return true;
    }
    else {
      return false;
    }
  }

  public void textBox () {
    textAlign(CENTER);
    textSize(32);
    fill(255);
    text(word, rectX + 10, rectY + 10, buttonWidth - 20, buttonLength - 20);
  }

  public int getAction () {
    return action;
  }
}
```

Code Checklist

1. Variable and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions? **yes**.
2. Are there variables with confusingly similar names? **no**.
3. Is every variable properly initialized? **yes**.

4. Could any non-local variables be made local? **no**.

5. Are there literal constants that should be named constants? **no**.

6. Are there macros that should be constants? **no**.

7. Are there variables that should be constants? **no**.

2. Function Definition Defects (FD)

8. Are descriptive function names used in accord with naming conventions? **yes**.

9. Is every function parameter value checked before being used? **No. But this is alright in such an application because all game state is controlled by the programmer / using input via mouse and directional pad**

10. For every function: Does it return the correct value at every function return point? **yes.**

3. Class Definition Defects (CD)

11. Does each class have an appropriate constructor and destructor? **yes. and n/a no destructors in Java.**

12. For each member of every class: Could access to the member be further restricted? **no.**

13. Do any derived classes have common members that should be in the base class? **n/a**

14. Can the class inheritance hierarchy be simplified? **n/a**

4. Computation/Numeric Defects (CN)

15. Is overflow or underflow possible during a computation? **Yes, however as mentioned before, in this type of application the control flow is generated by the programmer and these checks are not necessary here.**

16. For each expressions with more than one operator: Are the assumptions about order of evaluation and precedence correct? **yes.**

17. Are parentheses used to avoid ambiguity? **yes.**

5. Comparison/Relational Defects (CR)

18. Are the comparison operators correct? **yes.**

19. Is each boolean expression correct? **yes.**

20. Are there improper and unnoticed side-effects of a comparison? **no.**

6. Control Flow Defects (CF)

21. For each loop: Is the best choice of looping constructs used? **yes.**

22. Will all loops terminate? **yes.**

23. When there are multiple exits from a loop, is each exit necessary and handled properly?

yes. yes.

24. Does each switch statement have a default case? **n/a**

25. Are missing switch case break statements correct and marked with a comment? **n/a**

26. Is the nesting of loops and branches too deep, and is it correct? **no. yes.**

27. Can any nested if statements be converted into a switch statement? **no. Not in a well formed way, that is.**

28. Are null bodied control structures correct and marked with braces or comments? **n/a**

29. Does every function terminate? **yes.**

30. Are goto statements avoided? **yes.**

7. Input-Output Defects (IO)

31. Have all files been opened before use? **n/a**

32. Are the attributes of the open statement consistent with the use of the file?**n/a**

33. Have all files been closed after use? **n/a**

34. Is buffered data flushed? **n/a**

35. Are there spelling or grammatical errors in any text printed or displayed? **n/a**

36. Are error conditions checked?**n/a**

8. Module Interface Defects (MI)

37. Are the number, order, types, and values of parameters in every function call in agreement with the called function's declaration? **n/a**

38. Do the values in units agree (e.g., inches versus yards)? **yes.**

9. Comment Defects (CM)

39. Does every function, class, and file have an appropriate header comment? **no. These should be added, perhaps.**

40. Does every variable or constant declaration have a comment? **no. Not every variable needs one though since well formed and useful names are used.**

41. Is the underlying behavior of each function and class expressed in plain language? **No.**

42. Is the header comment for each function and class consistent with the behavior of the function or class? **n/a**

43. Do the comments and code agree? **yes.**

44. Do the comments help in understanding the code? **yes.**

45. Are there enough comments in the code? **yes.**

46. Are there too many comments in the code? **no.**

10. Packaging Defects (LP)

47. For each file: Does it contain only one class? **yes.**

48. For each function: Is it no more than about 60 lines long? **yes.**

49. For each class: Is no more than 2000 lines long (Sun Coding Standard) ? **yes.**

11. Modularity Defects (MO)

50. Is there a low level of coupling between packages (classes)? **n/a**

51. Is there a high level of cohesion within each package? **n/a**

52. Is there duplicate code that could be replaced by a call to a function that provides the behavior of the duplicate code? **no.**

53. Are framework classes used where and when appropriate? **n/a**

12. Performance Defects (PE) [Optional]

54. Can better data structures or more efficient algorithms be used? **no.**

55. Are logical tests arranged such that the often successful and inexpensive tests precede the more expensive and less frequently successful tests? **yes.**

56. Can the cost of recomputing a value be reduced by computing it once and storing the results? **no.**

57. Is every result that is computed and stored actually used? **n/a**

58. Can a computation be moved outside a loop? **no.**

59. Are there tests within a loop that do not need to be done? **no.**

60. Can a short loop be unrolled? **no.**

61. Are there two loops operating on the same data that can be combined into one? **no.**

Button Testing

To test the button class we made a sample sketch in Processing which emulated the use of an instance of Button and the visual effects and functionality that were required.

The code for this mock-up test is:

```
Button button;

void setup() {
  background(0);
  size(600, 600);
  button = new Button(200, 200, 200, 200, "200", 200, 300);
}
```

```
void draw() {
  button.draw();
}

void mouseMoved() {
  println("X: " + (new Integer(mouseX)).toString() + ", Y: " + (new
Integer(mouseY)).toString());
}
```

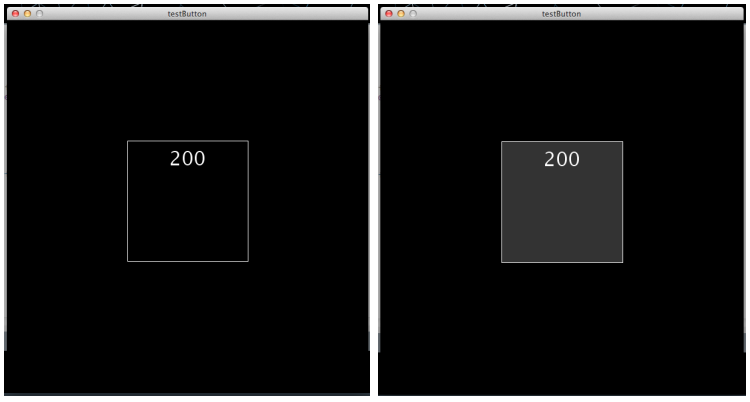
A few notes about the expected behavior of the constructor in this case: 1) the button should be 200 by 200 pixels. 2) the button should be 200 by 200 pixels from the top left corner of the screen. 3) the text displayed by the button should be “200” 4) the current and next actions for this button are 200 and 300, respectively; yielding a change in internal state when the button is clicked.

With this test we were able to verify that the button’s visual style was correct given the respective input (i.e. mouse movements). We were able to verify that when the mouse was hovering over the exact correct positions (as are printed to the console in the mouseMove() event), the button changed its fill color. We also verified that the text was displaying to the screen as expected.

The expected visual behavior of an instance of the Button class is that when the mouse hovers over any part of the button, the instance changes its color to notify the user that the button is currently clickable.

Here are some screenshot of this sample “mock-up” application in its different states:

Mouse not over: **Mouse over:**

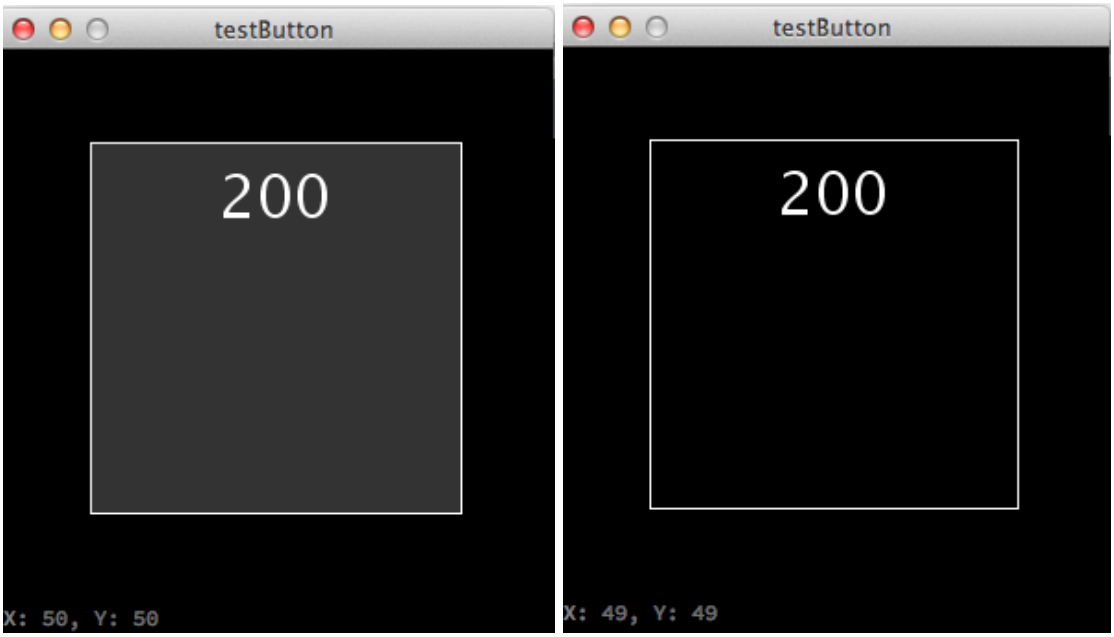


This “mock-up” application / test driver is the best way, when using a visual medium such as Processing, to “black box” test the behavior of a simple visual class such as this Button class and its visual representation. Here we see that the visual performance of this class is working correctly under the correct inputs (i.e. location of the mouse is inside that of the borders of the Button class instance).

To further test this classes' visual representation, the boundary conditions (i.e. "boundary testing") must be taken into consideration. The boundary conditions necessary here are the exact coordinates of the mouse when the button starts to change its color. To test this we printed out the exact coordinates of the mouse whenever the mouse moves so that we could track the change in color of the button at that particular moment.

To demonstrate this idea for this report I changed the starting location of the button to (50, 50) on the screen and took a screenshot of the console:

Button state at (50, 50): **Button state at (49, 49):**



Note that you can see the location of the mouse ^ in the console screen below the application. This displays the exact coordinates (x, y) of the mouse when the box became shaded, thus testing this boundary condition.

This same "boundary condition" testing process was done for many different cases and for all of the surrounding points of access where this Button instance should change color (i.e. the mouse is inside the walls of the Button instance).

We also tested the "image button" type instance of the Button class. Here instead of displaying a box that has a hover over "color change" we have a static image.

Here is a screenshot of that application:

Here the whole “money bag” picture with the white box around it is the bounds of the Button class instance. Note that similar boundary tests were performed as mentioned before but since there was no visual feedback (i.e. no change of color) this had to be tested more thoroughly during the button click tests mentioned below.

To test the buttons functionality (i.e. its “clickability” => response to the click event) we implemented a visual verification to our sample application that indicated that a button has been clicked. The visual verification is that a word appears (e.g. “Clicked”) saying that this instance of the Button class has been clicked.

Here are the changes to the test code => sample “mock-up” application:

```
Button button;

void setup() {
  background(0);
  size(300, 300);
  PImage buttonImage = loadImage("MOney-Bag1.jpg");
  buttonImage.resize(200, 200);
  button = new Button(50, 50, buttonImage, 200, 300);
}

void draw() {
  background(0);
  button.draw();
  if (button.action > 200) {
    textAlign(CENTER);
    textSize(32);
```

```
    fill(255);
    text("Clicked!", 75, 255, 150, 150);
    button.action = 200;
  }
}

void mouseMoved() {
  println("X: " + (new Integer(mouseX)).toString() + ", Y: " + (new
Integer(mouseY)).toString());
}
```

Note that this code also displays the change in logic that was needed to change the button from a boxed button with text to a button that uses an image as its visual representation.

Here is a screenshot of this new test code in action:

Note that here we as testers are notified of a click when the mouse has been clicked by the word “Clicked!” appearing on the screen below the instance of the Button class.

Here is a screenshot of the functionality of the sample applications click event when using the boxed text instance of the Button class:

Again, here is the draw function for the Button class:

```
public void draw() {
    update(mouseX, mouseY); // path 0...
    if (bgImage != null){ // path 1...
      image(bgImage, rectX, rectY);
    }
    else { // path 2...
      if (rectOver) { // path 2.1...
        fill(rectHighlight);
      }
      else { // path 2.2...
        fill(rectColor);
      }
      stroke(255);
      rect(rectX, rectY, buttonWidth, buttonLength);
      textBox();
    }
    if (mousePressed && rectOver) { // path 3...
      action = nextAction;
    }
  }
```

Note that the paths used for path testing have been put in this code sample as comments on their respective lines.

To test this function, first we use path testing and the test cases provided above when we tested the visual aspects of the system (this makes sense to use these tests since the “draw()”

method / function has the sole responsibility of drawing the instance of the Button class to the screen.

Path 0 [`update(mouseX, mouseY);`]:

This function call leads the sequential control flow of this program into the update() method. This method is very trivial in its implementation and can be left out, however it must be mentioned that this methods functionality has been tested by the visual representation tests above since its sole responsibility is to update the mouse coordinates and the rectOver variable.

Path 1 [`if (bgImage != null)`]:

We know that from the tests of the visual representation that this has been hit when an image has been provided because the image has been drawn to the screen. The only way that an image can be drawn to the screen in Processing (that we know of) is by using the “image()” function and this is the only occurrence of this function call in this class.

Path 2 [`else // i.e. not path 1`]:

We know that from the tests of the visual representation that this has been hit when an image has not been provided because the Button class then draws instead a boxed text button instead of an image button.

Path 2.1 [`rectOver`]:

We know that from the tests of the visual representation that this path has been hit because when we are hovering over this image with the mouse the fill is a different, darker grey color. Here in this code this is exemplified by the `fill(rectHighlight);` function call in this portion of the code.

Path 2.2 [`else // i.e. not path 2.1`]:

We know that from the tests of the visual representation that this path has been hit because when we are not hovering over this image with the mouse the fill is black in color. Here in this code this is exemplified by the `fill(rectColor);` function call in this portion of the code.

Path 3 [`mousePressed && rectOver`]:

We know that from the tests of the visual representation that this path has been hit because when we click (i.e. mousedPressed is set to “true”) and we are hovering the mouse over the instance of the Button class (i.e. rectOver is set to “true”) the text “Clicked!” appeared. This click event changes the action variable (i.e. `action = nextAction;` => note in the test driver / sample application action = 200 and nextAction = 300 as set in the constructor) of the Button class which is checked by the draw() method of our testing driver and the word “Clicked!” appears as expected. This part was therefore tested during the functionality test part of our testing.

Therefore we have exhausted the paths of this method and have completed the path testing scheme of testing this function.

The tests of the overRect(), update(), textBox(), getAction(), reset(), and constructor methods were all tested in a similar fashion and were compared with the visual representation tests in a very similar way. These functions are much simpler than the draw method and only aid in its functionality and were thus proven inductively by proving the draw method was fully functional. We have accomplished this in full.

Progress Bar (Abhishek Singh Rathore)

Original Source Code

```
public class ProgressBar {
  private float lineFill = 0;
  private int xpos, ypos, xwidth, yheight;
  private float startTime = millis();
  private float endTime = startTime + PROGRESSDELAY;
  private boolean complete = false;
  private float percent = 0;

  public ProgressBar(int startX, int startY, int startWidth, int startHeight)
{
    xpos = startX;
    ypos = startY;
    xwidth = startWidth;
    yheight = startHeight;
  }

  public void draw() {
    noFill();
    stroke(255);
    rect(xpos, ypos, xwidth, yheight);

    if (!complete) {
      percent = (millis() - startTime) / (endTime - startTime);
      if (percent > 1) {
        percent = 1;
        complete = true;
      }
    }
    for (float i = 0; i < xwidth; i++) {
      fill(lineFill + percent * 255);
      rect(xpos + 2, ypos + 2, percent * (xwidth - 4), yheight - 4);
    }
```

```
    if (complete) {
      writeComplete("100% Loaded");
    }
    else {
      Integer done = int(percent * 100);
      writeComplete(done.toString() + "% Loaded");
    }
  }

  private void writeComplete(String txt) {
    textAlign(CENTER, CENTER);
    textSize(32);
    fill(128);
    text(txt, xpos + xwidth / 2, ypos + yheight / 2);
  }

  public boolean isComplete() {
    return complete;
  }
}
```

Code Checklist

1. Variable and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions? **yes**.
2. Are there variables with confusingly similar names? **no**.
3. Is every variable properly initialized? **yes**.
4. Could any non-local variables be made local? **no**.
5. Are there literal constants that should be named constants? **no**.
6. Are there macros that should be constants? **no**.
7. Are there variables that should be constants? **no**.

2. Function Definition Defects (FD)

8. Are descriptive function names used in accord with naming conventions? **yes**.
9. Is every function parameter value checked before being used? **No. But this is alright in such an application because all game state is controlled by the programmer / using input via mouse and directional pad**
10. For every function: Does it return the correct value at every function return point? **yes.**

3. Class Definition Defects (CD)

11. Does each class have an appropriate constructor and destructor? **yes. and n/a no destructors in Java.**

12. For each member of every class: Could access to the member be further restricted? **no.**

13. Do any derived classes have common members that should be in the base class? **n/a**

14. Can the class inheritance hierarchy be simplified? **n/a**

4. Computation/Numeric Defects (CN)

15. Is overflow or underflow possible during a computation? **Yes, however as mentioned before, in this type of application the control flow is generated by the programmer and these checks are not necessary here.**

16. For each expressions with more than one operator: Are the assumptions about order of evaluation and precedence correct? **yes.**

17. Are parentheses used to avoid ambiguity? **yes.**

5. Comparison/Relational Defects (CR)

18. Are the comparison operators correct? **yes.**

19. Is each boolean expression correct? **yes.**

20. Are there improper and unnoticed side-effects of a comparison? **no.**

6. Control Flow Defects (CF)

21. For each loop: Is the best choice of looping constructs used? **yes.**

22. Will all loops terminate? **yes.**

23. When there are multiple exits from a loop, is each exit necessary and handled properly? **yes. yes.**

24. Does each switch statement have a default case? **n/a**

25. Are missing switch case break statements correct and marked with a comment? **n/a**

26. Is the nesting of loops and branches too deep, and is it correct? **no. yes.**

27. Can any nested if statements be converted into a switch statement? **no. Not in a well formed way, that is.**

28. Are null bodied control structures correct and marked with braces or comments? **n/a**

29. Does every function terminate? **yes.**

30. Are goto statements avoided? **yes.**

7. Input-Output Defects (IO)

31. Have all files been opened before use? **n/a**

32. Are the attributes of the open statement consistent with the use of the file?**n/a**

33. Have all files been closed after use? **n/a**

34. Is buffered data flushed? **n/a**

35. Are there spelling or grammatical errors in any text printed or displayed? **n/a**

36. Are error conditions checked?**n/a**

8. Module Interface Defects (MI)

37. Are the number, order, types, and values of parameters in every function call in agreement with the called function's declaration? **n/a**

38. Do the values in units agree (e.g., inches versus yards)? **yes.**

9. Comment Defects (CM)

39. Does every function, class, and file have an appropriate header comment? **no. These should be added, perhaps.**

40. Does every variable or constant declaration have a comment? **no. Not every variable needs one though since well formed and useful names are used.**

41. Is the underlying behavior of each function and class expressed in plain language? **No.**

42. Is the header comment for each function and class consistent with the behavior of the function or class? **n/a**

43. Do the comments and code agree? **yes.**

44. Do the comments help in understanding the code? **yes.**

45. Are there enough comments in the code? **yes.**

46. Are there too many comments in the code? **no.**

10. Packaging Defects (LP)

47. For each file: Does it contain only one class? **yes.**

48. For each function: Is it no more than about 60 lines long? **yes.**

49. For each class: Is no more than 2000 lines long (Sun Coding Standard) ? **yes.**

11. Modularity Defects (MO)

50. Is there a low level of coupling between packages (classes)? **n/a**

51. Is there a high level of cohesion within each package? **n/a**

52. Is there duplicate code that could be replaced by a call to a function that provides the behavior of the duplicate code? **no.**

53. Are framework classes used where and when appropriate? **n/a**

12. Performance Defects (PE) [Optional]

54. Can better data structures or more efficient algorithms be used? **no.**

55. Are logical tests arranged such that the often successful and inexpensive tests precede the more expensive and less frequently successful tests? **yes.**

56. Can the cost of recomputing a value be reduced by computing it once and storing the results? **no.**

57. Is every result that is computed and stored actually used? **n/a**

58. Can a computation be moved outside a loop? **no.**

59. Are there tests within a loop that do not need to be done? **no.**

60. Can a short loop be unrolled? **no.**

54. Can better data structures or more efficient algorithms be used? no.
55. Are logical tests arranged such that the often successful and inexpensive tests precede the more expensive and less frequently successful tests? yes.
56. Can the cost of recomputing a value be reduced by computing it once and storing the results? no.
57. Is every result that is computed and stored actually used? n/a
58. Can a computation be moved outside a loop? no.
59. Are there tests within a loop that do not need to be done? no.
60. Can a short loop be unrolled? no.
61. Are there two loops operating on61. Are there two loops operating on the same data that can be combined into one? **no.**

ProgressBar Testing

To test the button class we made a sample sketch in Processing which emulated the use of an instance of ProgressBar and the visual effects and functionality that were required. Note that this class is somewhat trivial in that it does not perform a very complex action.

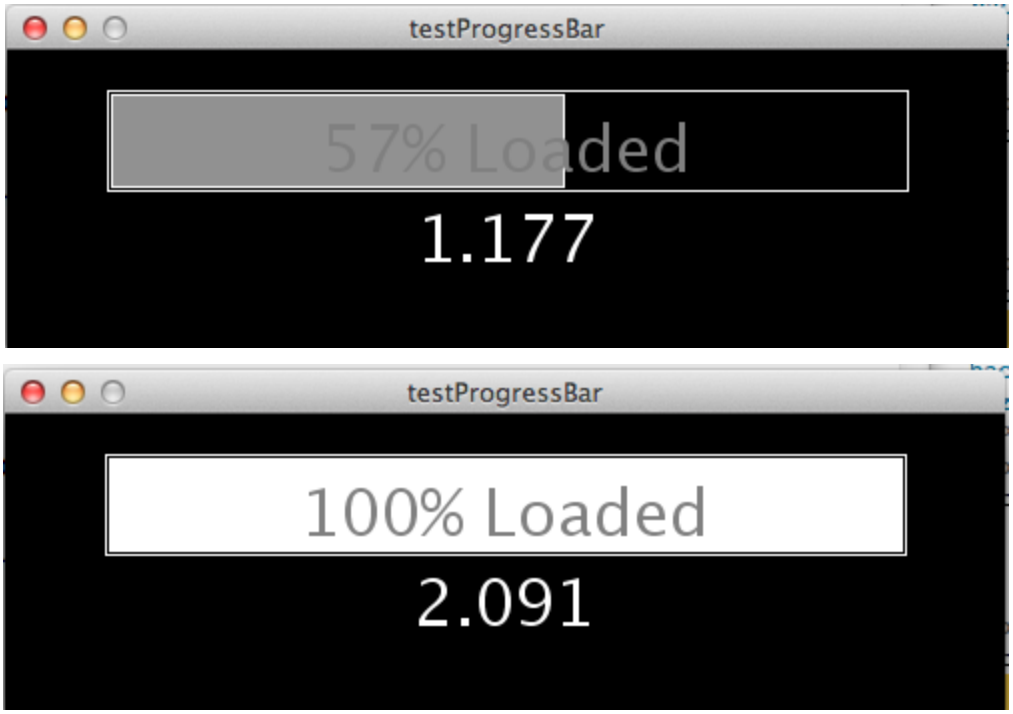
The code for this mock-up / test driver is:

```
public final Integer PROGRESSDELAY = 2000;
float startTime = 0.00;

ProgressBar progressBar;

void createProgressBar() {
  progressBar = new ProgressBar(50, 20, 400, 50);
  startTime = millis();
}

void setup() {
  background(0);
  size(500, 150);
  createProgressBar();
```

```
}

void draw() {
  background(0);
  progressBar.draw();

  String text = "";

  float currentTime = millis();

  text = (new Float((currentTime - startTime) / 1000)).toString();

  textAlign(CENTER);
  textSize(32);
  fill(255);
  text(text, 175, 75, 150, 150);
}

void mousePressed() {
  createProgressBar();
}

void mouseMoved() {
  println("X: " + (new Integer(mouseX)).toString() + ", Y: " + (new
Integer(mouseY)).toString());
}
```

We should note here that the initial coordinates for the starting position of the Progress bar in this test driver is (50, 20). Also the length of the progress bar is 400 pixels by 50 pixels.

Below are some screenshot of this sample application in action:



Note that the progress-bar is loaded to 100% after 2 seconds. This is what is expected when the variable `PROGRESSDELAY` is set to 2000 (i.e. because milliseconds == 2 seconds).

GameSaved (Zichen Weng)

Original Source Code

```
public class GameSaved extends Screen {
  private Button ok;

  public GameSaved() {
    ok = new Button(width /2  - 275/2, height /2 + 75/2, 275, 75, "Continue",
GAMESAVED, INGAMEMENU);
  }

  public void draw() {
    background(GAMEBG);
    ok.draw();
    if ( true ) {
      tint(random(255));
      image(GAMESAVE, width/2 - 424/2, height/2 -81/2);
    }
  }

  public int getAction() {
    if (ok.getAction() == INGAMEMENU) {
```

```
      tint(255);
      game.unPause();
      ok.reset();
      return GAME;
    }
    else
      return GAMESAVED;
  };
}
```

Code Checklist

1. Variable and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions? **no**
2. Are there variables with confusingly similar names? no
3. Is every variable properly initialized? yes
4. Could any non-local variables be made local? no
5. Are there literal constants that should be named constants? **yes, position elements**
6. Are there macros that should be constants? no
7. Are there variables that should be constants? no

2. Function Definition Defects (FD)

8. Are descriptive function names used in accord with naming conventions? yes
9. Is every function parameter value checked before being used? yes (n/a) no parameters used
10. For every function: Does it return the correct value at every function return point? yes

3. Class Definition Defects (CD)

11. Does each class have an appropriate constructor and destructor? yes, the default de-constructor is valid
12. For each member of every class: Could access to the member be further restricted? no
13. Do any derived classes have common members that should be in the base class? n/a
14. Can the class inheritance hierarchy be simplified? no

4. Computation/Numeric Defects (CN)

15. Is overflow or underflow possible during a computation? **possible**
16. For each expressions with more than one operator: Are the assumptions about order of evaluation and precedence correct? yes

17. Are parentheses used to avoid ambiguity? **no**

5. Comparison/Relational Defects (CR)

18. Are the comparison operators correct? yes

19. Is each boolean expression correct? yes

20. Are there improper and unnoticed side-effects of a comparison? none

6. Control Flow Defects (CF)

21. For each loop: Is the best choice of looping constructs used? n/a

22. Will all loops terminate? n/a

23. When there are multiple exits from a loop, is each exit necessary and handled properly? n/a

24. Does each switch statement have a default case? n/a

25. Are missing switch case break statements correct and marked with a comment? n/a

26. Is the nesting of loops and branches too deep, and is it correct? n/a

27. Can any nested if statements be converted into a switch statement? n/a

28. Are null bodied control structures correct and marked with braces or comments? n/a

29. Does every function terminate? **If we assume that ok.draw() terminate, then yes, otherwise this could be a problem.**

30. Are goto statements avoided? yes

7. Input-Output Defects (IO)

31. Have all files been opened before use? n/a

32. Are the attributes of the open statement consistent with the use of the file? n/a

33. Have all files been closed after use? n/a

34. Is buffered data flushed? n/a

35. Are there spelling or grammatical errors in any text printed or displayed? no

36. Are error conditions checked? no

8. Module Interface Defects (MI)

37. Are the number, order, types, and values of parameters in every function call in agreement with the called function's declaration? yes

38. Do the values in units agree (e.g., inches versus yards)? n/a

9. Comment Defects (CM)

39. Does every function, class, and file have an appropriate header comment? **no**

40. Does every variable or constant declaration have a comment? **no**

41. Is the underlying behavior of each function and class expressed in plain language? yes

42. Is the header comment for each function and class consistent with the behavior of the function or class? **n/a**

43. Do the comments and code agree? **n/a**

44. Do the comments help in understanding the code? **n/a**

45. Are there enough comments in the code? **n/a**

46. Are there too many comments in the code? **n/a**

10. Packaging Defects (LP)

47. For each file: Does it contain only one class? yes

48. For each function: Is it no more than about 60 lines long? yes

49. For each class: Is no more than 2000 lines long (Sun Coding Standard) ? yes

11. Modularity Defects (MO)

50. Is there a low level of coupling between packages (classes)? yes

51. Is there a high level of cohesion within each package? yes

52. Is there duplicate code that could be replaced by a call to a function that provides the behavior of the duplicate code? no

53. Are framework classes used where and when appropriate? yes

12. Performance Defects (PE) [Optional]

54. Can better data structures or more efficient algorithms be used? n/a

55. Are logical tests arranged such that the often successful and inexpensive tests precede the more expensive and less frequently successful tests? n/a

56. Can the cost of recomputing a value be reduced by computing it once and storing the results? n/a

57. Is every result that is computed and stored actually used? n/a

58. Can a computation be moved outside a loop? n/a

59. Are there tests within a loop that do not need to be done? n/a

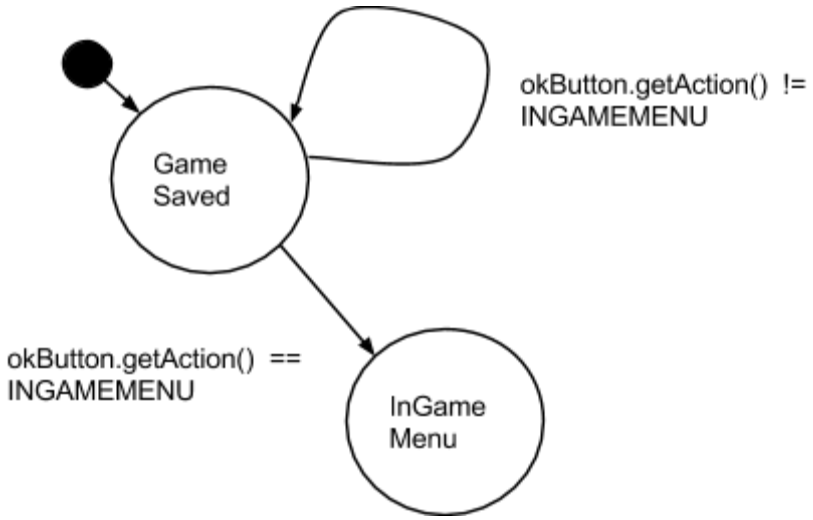
60. Can a short loop be unrolled? n/a

61. Are there two loops operating on the same data that can be combined into one? n/a

GameSaved Testing

There were several points in the checklist that the code did not pass. First off, there was an unnecessary if (true) statement in the draw() method, that was removed because it had no effect on the code. Second, the the literals 275 and 75 were pulled out of the code and new constants, BUTTONWIDTH and BUTTONHEIGHT were introduced. Third, the positioning of the image in the draw method() utilized the calculations 424 / 2 and 81 / 2. It was unclear why the results of these calculations were not used explicitly. They were changed to 400 and 40 respectively.

In order to analyze the getAction() method, we developed a state machine diagram. When the okButton is initialized in the constructor, the state of the button is set to GAMESAVED. While the okButton is in this state, getAction() should return GAMESAVED. When the button is clicked, the internal state of the button transitions to INGAMEMENU. At this point, getAction() should return INGAMEMENU. It was apparent that this was not the case in the initial code. In the initial code, the game transitioned back to the GAME. This was incorrect. We modified the source code to point to the correct INGAMEMENU, and removed the call to unpause the game.



Modified Source Code

```
public class GameSaved extends Screen {
  private Button okButton;
  private final int BUTTONWIDTH = 275;
  private final int BUTTONHEIGHT = 75;

  public GameSaved() {
    okButton = new Button(width /2  - BUTTONWIDTH /2, height /2 + BUTTONHEIGHT
/2, BUTTONWIDTH, BUTTONHEIGHT, "Continue", GAMESAVED, INGAMEMENU);
  }

  public void draw() {
    background(GAMEBG);
    okButton.draw();
```

```
    tint(random(255));
    image(GAMESAVE, width/2 - 200, height/2 - 40);
  }

  public int getAction() {
    if (okButton.getAction() == INGAMEMENU) {
      tint(255);
      okButton.reset();
      return INGAMEMENU;
    }
    else
      return GAMESAVED;
  };
}
```

Summary

Our testing process generated higher quality code than we started with. Several defects were uncovered and corrected. We certainly see the value of unit testing during development and would have liked to incorporate some of these methodologies into our project. Obviously we could not test all of the code in the project in such a short time period. In a real world project, we would expect to spend considerable resources in testing. We are satisfied however with the code we were able to test.