Advanced Models and Programs

Project Work

PONG

by Benjamin Irani and Nedyalko Kargov

email: irani@itu.dk email: nvka@itu.dk
CPR: 120467-3345 CPR: 170982-4089

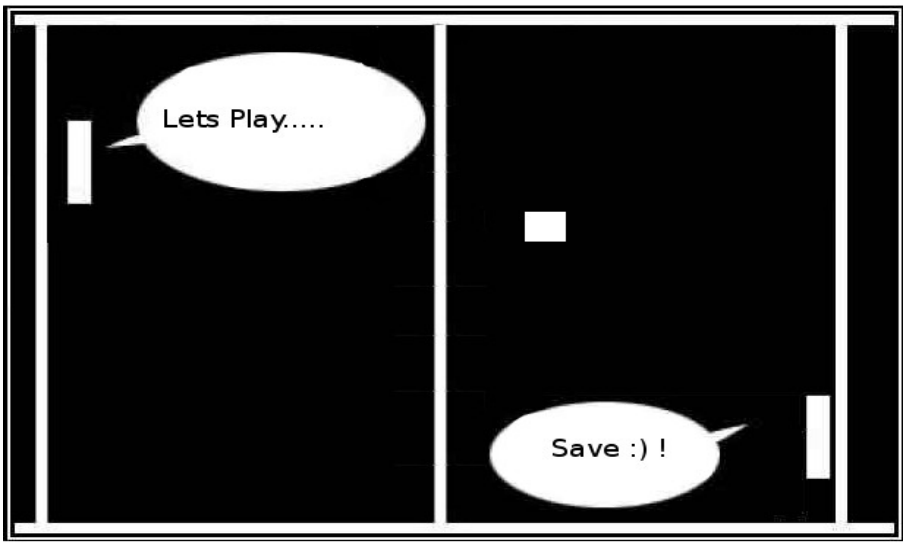


Table of Contents

1. Purpose

This project aims to present the implementation of a small 8bit computer game (PONG) by using some of the tools and verification approaches learned in the Advanced Models and Programs course. In the next sections we will explore some key parts of the system and explain our decisions in order to argue for the correctness and the quality of our code.

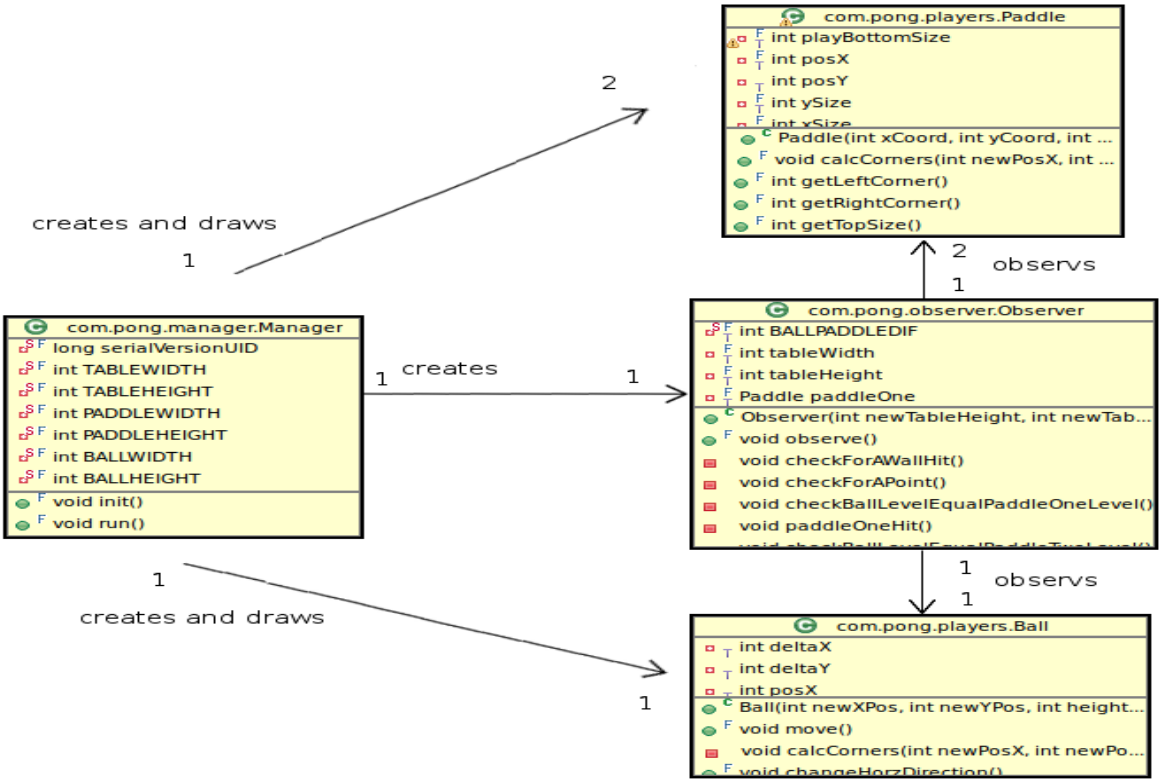
2. Requirements

Based on our project proposal and after the discussion with the project supervisor Dr. Joe Kiniry, we decide to focus our work on the following features:

- Correctness of the code.
- Graphical verification of the game.
- Model verification.

3. System Architecture

Before going deep into implementation details, we need to know something more about our system architecture. Based on the analysis of the problem domain the particular implementation of the system class diagram contains the following classes and relationships:



3.1 Manager

The Manager is responsible for the initialization of the main set of objects (the Observer, the two Paddles, and the Ball). Our decision to implement this class as a Java applet is based on the

inheritance which comes from java.awt.Component. By inheriting from java.awt.Component the Manager applet has all that we need in a way to draw our main objects on the screen and also register users key inputs (see appendix A for the class implementation).

3.2 Observer

The Observer has the purpose to regulate the game. Initialized by the Manager the Observer object controls the players (the Ball and the two Paddles) and applies the game rules. For instance, if the Ball hit one of the Paddles, the Observer changes the state of the Ball by changing the direction of the Ball. Another rule is if the Ball is out of the playground, the Observer tells the Ball to restart its coordinates. We consider this class the most critical in our architecture and most of our verification efforts are aimed to ensure the correctness of this class.

3.3 Ball

Essentially this class creates a small data structure which represents the Ball object. It contains methods for assessing some of the internal instance fields of the object and also methods for changing the state of the object.

3.4 Paddle

As with the Ball, the Paddle class essentially creates a data structure which represents a Paddle object in the game. Again the class contains assessor and mutator methods in order to provide the necessary access points for control.

4.Tools

Based on our experience from the Advanced Models and Programs course in the PONG project we use the following set of tools:

- **FindBugs** – we use FindBugs as a lightweight static analysis tool. The analysis engine of the tool analyses the bytecode (the compiled class files). Behind the scene**,** the tool uses syntactical match source code for known bad programming practices.
- **Metrics** – even though it is not a verification tool, the Metrics plugin for Eclipse (our IDE) gives us a good overview for some of the properties of our code.
- **CheckStyle** – in this project we focus on correctness and we try to follow only good programming practices. We consider the coding conventions and the coding style as one of the important aspects which significantly improves the quality of the product. To guarantee that we use CheckStyle which checks our code against the Sun's coding conventions.
- **PMD** – we use that static analysis tool in combination with the rest of the tools. What we find particularly useful with PMD is the ability to check the complexity of the code. The measurement of the Cyclomatic complexity comes really useful when we use ESC/Java (the Extensive Static Checking for Java) and implement the JML specifications. In order to satisfy the preconditions of the underlying methods, some methods need to handle a lot of conditional statements. A particular example will be explained and discuss in the verification section of the report.
- **Jalopy** – this tool does not check our code for bugs, flows or mistakes, but the tool is useful when it comes for applying coding conventions. The tool formats the code automatically when the developer hits save or apply the changes on demand.

– **Auto-Grader** it gives us a good overview of our system by capturing the output from all of the mentioned tools.

Metrics	BONc	Checkstyle	ESC/Java2	FindBugs	PMD	Total
TLOC:	Errors:	Errors per KLOC:	Errors per KLOC:	Errors per KLOC:	Errors per KLOC:	
1,152	0	0 (Total: 0.0)	0 (Total: 0.0)	0 (Total: 0.0)	0 (Total: 0.0)	
Grade: N/A	Grade: A	Grade: A	Grade: A	Grade: A	Grade: A	
Average method LOC:	Warnings:	Warnings per KLOC:	Warnings per KLOC:	Warnings per KLOC:	Warnings per KLOC:	
7.21	0	0 (Total: 0.0)	0 (Total: 0.0)	0 (Total: 0.0)	6.94 (Total: 8.0)	
Grade: B	Grade: A	Grade: A	Grade: A	Grade: A	Grade: A	
Average method CC:						
1.91						
Grade: B						
Total:	Total:	Total:	Total:	Total:	Total:	**Overall:**
Grade: B	**Grade: A**	**Grade: A**	**Grade: A**	**Grade: A**	**Grade: A**	**Grade: A-**

From the Auto-Grader output we could see the nice looking green color and the overall grade A-. It is true that we have 8 PMD warnings and grade B from the Metrics but the warnings are taken into consideration, examined and consider as not harmful for our system. For instance, in the code bellow the Metrics tool issues a warning which tells us that we try to pass too many parameters in the Observer's constructor.

```
    /**
     * Creates a new Observer object.
     *
     * @param newTableHeight        the new table height
     * @param newTableWidth         the new table width
     * @param newPaddleOne          the new paddle one
     * @param newPaddleTwo          the new paddle two
     * @param newBall               the new ball
     * @param newPaddleHeight       the new paddle height
     */
    //@ requires newPaddleOne != null;
    //@ requires newPaddleTwo !=null;
    //@ requires newBall != null;
    //@ ensures this != null;
    //@ ensures this.tableWidth == newTableWidth;
    //@ ensures this.tableHeight == newTableHeight;
    //@ ensures this.paddleHeight == newPaddleHeight;
    public Observer(final int newTableHeight,
                    final int newTableWidth,
                    final Paddle newPaddleOne,
                    final Paddle newPaddleTwo,
                    final Ball newBall,
                    final int newPaddleHeight) {  - More than 5 parameters
        this.tableWidth = newTableWidth;
        this.tableHeight = newTableHeight;
        this.paddleOne = newPaddleOne;
        this.paddleTwo = newPaddleTwo;
        this.ball = newBall;
        this.paddleHeight = newPaddleHeight;
    }
```

Even though we have more than 5 parameters in the constructor, the Observer object needs all of them in order to effectively control the game. The good thing with warnings is that the tool points the potential problems and we know exactly where we need to do some refactoring.

Another good example is a warning which comes from PMD.

```
/**
 * The Paddle class.
 *
```

```
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Paddle {
    //~ Instance fields ----------------------------------------------------------

     //@ ghost int playBottomSize;
/**
     * Instantiates a new paddle.
     *
     * @param  xCoord       the initial X coordinates
     * @param  yCoord       the initial Y coordinates
     * @param  width        the width
     * @param  height       the height
     * @param  pBottomSize  the bottom size
     * @param  newSpeed     the new speed
     */
      //@ ensures this != null;
      //@ ensures getPosX() == xCoord;
      //@ ensures getPosY() == yCoord;
      //@ ensures this.xSize == width;
      //@ ensures this.ySize == height;
      //@ ensures this.playBottomSize == pBottomSize;
    public Paddle(final int xCoord,
                  final int yCoord,
                  final int width,
                  final int height,
                  final int pBottomSize,- PMD warning. Avoid unused constructor
                                          parameters such as 'pBottomSize'.
                  final int newSpeed) {
        this.posX = xCoord;
        this.posY = yCoord;
        this.xSize = width;
        this.ySize = height;
        this.speed = newSpeed;
        this.setDirection(0);
        calcCorners(this.posX, this.posY);
        //@ set playBottomSize = pBottomSize;
    }
```

In this particular case the pBottomSize parameter provides the Paddle with information how big the playground is. The parameter is used only in our verification to initialize and set the value of the JML ghost field `playBottomSize`. Because we use JML syntax to set the field, PMD cannot recognize it and warns us for a potential problem.

In a conclusion, all of the tools above does not have the intention to tell us what is wrong or what is right, but to attract our attention to places in our code that need to be improved or there is something suspicious and needs to be examined more closely. Once again all warning in our code are taken into consideration and we know why they are there.

5. Verification.

For verification of our methods in addition to the mentioned tools we use ESC/JAVA2 – the Extensive Static Checking system for Java. Our method of working is first to implement all of the methods for particular class and then to verify their correctness with ESC/JAVA2 by providing supplementary JML specifications. The next example is taken from the Ball class.

```
/**
```

```
 * The Ball class.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Ball {
    //~ Instance fields ----------------------------------------------------

    /** The current X position of the Ball. */
      /*@ spec_public @*/ private transient int posX;

    /** The left corner of the Ball. */
      /*@ spec_public @*/ private transient int leftCorner;

    /** The right corner of the ball. */
      /*@ spec_public @*/ private transient int rightCorner;

    //~ Methods ------------------------------------------------------------
.................................................

    /**
     * Changes the horizontal direction of the Ball.
     */
    //@ requires this.getPosX() + (this.getDeltaX() * (-1)) >= 0;
    //@ requires this.getPosX() + (this.getDeltaX() * (-1)) <= this.groundXSize;
    //@ ensures getDeltaX() == \old(getDeltaX()) * (-1);
    //@ ensures getPosX() == \old(getPosX()) + getDeltaX();
    public final void changeHorzDirection() {
      this.deltaX = this.getDeltaX() * (-1);
        this.posX = this.posX + this.deltaX;
  }
.....................................................................


      //@ invariant this.posX >= 0;
      //@ invariant this.posX <= this.groundXSize;
      //@ invariant this.posY >= 0;
      //@ invariant this.posY <= this.groundYSize;
}
```

Even though ECS/JAVA2 is capable to verify our classes without JML specifications, the JML specifications could express much more. For instance, expressed in JML notation we capture the class invariants that restrict the Ball object from leaving the playground. Mostly because of the structure of the Ball class and the simplicity of the data structure our JML specs are lightweight. For instance, let's take the `changeHorzDirection()` method which changes the horizontal direction of the Ball. The first two lines of code marked with the keyword `requires` aim to satisfy the class invariants - the Ball X coordinate combined with the diagonal position of the Ball must be not negative and be less or equal than the groundXSize (the maximum X value of the playground). The next two lines from our JML specs ensure that if the preconditions are satisfied the appropriate result will be accomplished (changing the sign of the deltaX value, thus changing the moving direction of the Ball) .

```
    /**
     * Changes the horizontal direction of the Ball.
     */
    //@ requires this.getPosX() + (this.getDeltaX() * (-1)) >= 0;
```

```
//@ requires this.getPosX() + (this.getDeltaX() * (-1)) <= this.groundXSize;
//@ ensures getDeltaX() == \old(getDeltaX()) * (-1);
//@ ensures getPosX() == \old(getPosX()) + getDeltaX();
public final void changeHorzDirection() {
  this.deltaX = this.getDeltaX() * (-1);
  this.posX = this.posX + this.deltaX;
}
```

Something important which needs to be mentioned here is the close correlation between the method body and the JML specifications. The similarity is a result from our way of work which first implements the method body and after that provides the JML specification. The potential problem with this approach is that the specifications cloud becomes too rich and thus hard to be understand, implemented and changed.

The next example shows a piece of code from the Observer class which calls the `changeHorzDirection()` method of the Ball class.

```
/*@ private normal_behavior
  @ requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
  @ requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
  @                                          <=this.ball.getGroundYSize();
  @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
  @                                          > BALLPADDLEDIF;
  @ assignable \not_specified;
  @ ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
  @
  @ also
  @
  @ private normal_behavior
  @ requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1)) >= 0;
  @ requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1))
  @                                          <= this.ball.getGroundXSize();
  @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
  @                                                          > 0;
  @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
  @                                          < BALLPADDLEDIF;
  @ assignable \not_specified;
  @ ensures this.ball.getDeltaX() == \old(this.ball.getDeltaX()) * (-1);
  @*/
/**
 * Hit paddle one.
 */
private void hitPaddleOne() {
        final int diffX = this.paddleOne.getRightCorner();
        this.ball.getLeftCorner();

    if (diffX > 0) {
       if (diffX > BALLPADDLEDIF) {
            this.ball.changeVertDirection();
       } else {
           this.ball.changeHorzDirection();
        }
     }
}
```

For this method we use heavyweight specification to capture all possible code brunches and thus be able to ensure that the result from the method will be exactly what we want to be. Again we observe very long and specific JML specifications. The situation becomes even more complicated

when we need to handle the requirements that come from below, from the Ball class. As you can see the `changeHorzDirection()` requirements now pop up in the caller - the `hitPaddleOne()` method. Eventually this process could accumulate requirements from many underlying methods and end in the main method of the system where all of the requirements need to be handled in some way. At some point in our project we made the same mistake and the result was a Cyclomatic complexity equal to 40, and time for verification half of an hour. By using refactoring and introducing some additional methods we successfully resolve the complexity issue, cut the verification time to couple of seconds and still keep the positive verification from ESC/JAVA2. For the ESC/JAVA2 output with enabled -Stats switch, please see Appendix A.

6. Tests

In addition to the verification tools a substantial part of our project work is focused on unit tests. The task is done by using the following set of tools:

1. JUnit 4 – Unit testing
2. EclEmma – Code coverage
3. FEST framework – Java applet testing.
4. PrivateAccessor – private visibility accessor.

The JUnit framework as a standard testing tool for Eclipse gives us most of the functionalities to test our system. However, we enhance JUnit with couple of add-ons and the first of them is EclEmma. The tool gives us the ability to see and to measure the percentage of code covered by unit tests. In our project, where the verification needs to be considered as priority one our aim is to have at least 96% overall coverage. Unfortunately, some of parts of the code have private visibility and thus restricts us to directly touch and test the code. Of course, we can use the technique where we examine the state of the objects involved in this private area, but still it is hard and time consuming process. To remove this limitation we use a JUnit add-on with author Vladimir R. Bossicard. The PrivateAccessor add-on gives us the ability to directly invoke, get or set private methods and fields. The next example shows how we use it to invoke a private method from the Observer class which otherwise is directly unaccessible.

```
/**
 * The Class Observer.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Observer {
      …..........................................

    /*@ private normal_behavior
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
      @                                         <= this.ball.getGroundYSize();
      @     requires this.ball.getTopSize() <= 0;
      @     assignable \not_specified;
      @     ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
      @
      @ also
      @
      @ private normal_behavior
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
      @                                      <= this.ball.getGroundYSize();
```

```
  @     requires this.ball.getBottomSize() > this.tableHeight;
  @     assignable \not_specified;
  @     ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
  @ */
/**
 * Hit walls.
 */
private void hitWalls() {
  if (this.ball.getTopSize() <= 0) {
              this.ball.changeVertDirection();
  }

    if (this.ball.getBottomSize() > this.tableHeight) {
            this.ball.changeVertDirection();
    }
}
}
  …..........................................



/**
 * Test make a point - change vertical direction.
 *
 * @throws  Throwable  the throwable
 */
@Test
public final void testHitTopWallVerical() throws Throwable {
    final int ballXCoord = 20;
    final int ballYCoord = -4;

    this.ball.jumpTo(ballXCoord, ballYCoord);
    this.ball.move();

    final int oldDeltaY = 3;

    assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());
    final int newDeltaY = -3;

    PrivateAccessor.invoke(this.observer, "hitWalls", null, null);

    assertEquals(DELETAYMESSAGE, newDeltaY, this.ball.getDeltaY());
}
```

We find it really interesting to know how exactly the private visibility restriction is removed, but for now we do not find detailed information how actually this is done. We speculate that the tool uses some sort of reflection mechanism where the information is extracted from the bytecode.

As we said earlier our aim is to reach at least 96 percent code coverage, however by using the PrivateAccessor add-on the overall code coverage is close to 98 percent. This fact gives us the confidence to say that our system behaves as it should be and the requirements are satisfied.

Problems | Javadoc | Declaration | Console | Coverage | TestNG | Auto-Grader | Debug | Jalopy

AllTests (May 21, 2010 12:09:36 PM)

Element	Coverage	Covered Instructi
Pong	98.2 %	2474
src	96.1 %	1032
com.pong.manager	96.1 %	320
com.pong.observer	93.7 %	428
com.pong.players	100.0 %	284
test	99.8 %	1442
com.pong.managertests	100.0 %	553
com.pong.playerstests	100.0 %	270
com.pong.testsuite	86.4 %	19
test.pong.observertests	100.0 %	600

7. Graphical Verification.

One of the interesting aspects of our project is to do graphical verification of our players (the Ball and the two Paddles). The verification aims to prove that objects are rendered at the right positions with the right colors. To accomplish this we use the PrivateAccessor and the JUnit test framework.


```
/**
 * Test table paint.
 *
 * @throws  Throwable  if the PrivateAccessor cannot get or invoke the
 *                     private field or method
 */
@Test
public final void testObjectsGraphics() throws Throwable {
    PrivateAccessor.setField(this.manager, "inRun", false);

    PrivateAccessor.invoke(this.manager, "tablePaint", null, null);
```

```
        final BufferedImage bufferedImage = (BufferedImage)
                  PrivateAccessor.getField(this.manager, "bufferedImage");

        final int tableWidth = (Integer)
                  PrivateAccessor.getField(this.manager, "TABLEWIDTH");

        final int tableHeight = (Integer)
                  PrivateAccessor.getField(this.manager, "TABLEHEIGHT");

        assertNotNull("Buffered Image not as expected", bufferedImage);

        for (int y = 0; y < tableHeight; y++) {
            for (int x = 0; x < tableWidth; x++) {
                tPaddleOne(bufferedImage, x, y);
                tPaddleTwo(bufferedImage, x, y);
                tBall(bufferedImage, x, y);
            }
        }
    }
```

In the first part of the graphical verification we use the PrivateAccessor to penetrate the private visibility of the Manager class, stop the running animation thread, invoke the method which draws the Ball and Paddles objects on the screen and retrieve references. For rendering our objects, we use the technique of the double buffering where first we draw to a BufferedImage and after that draw that image at once. When this is done we have the bufferedImage object which holds the graphical representation of our Ball and Paddles objects. The next step is to use the height and the width of the playground to enter into a double loop where we use the bufferedImage to retrieve the color of each pixel and check it against the coordinates of the players – Ball or Paddle. You can see that we do our graphical verification in isolation – each object is isolated from the rest of the players, but we believe that with the rest of the tests we prove that our objects have the correct coordinates and we do not need to test each pixel from the image.

```
    /**
     * Test paddle one.
     *
     * @param   bufferedImage  bufferedImage the bufferedImage
     * @param   xCoordinate    the x coordinate
     * @param   yCoordinate    the y coordinate
     *
     * @throws  NoSuchFieldException  if the PrivateAccessor cannot get private
     *                                field paddleOne
     */
    private void tPaddleOne(final BufferedImage bufferedImage,
                            final int xCoordinate,
                            final int yCoordinate) throws NoSuchFieldException {
        final Paddle paddleOne = (Paddle) PrivateAccessor.getField(this.manager,
                                                                   PADDLEONE);

        if (xCoordinate >= paddleOne.getLeftCorner()
                && xCoordinate < paddleOne.getRightCorner()
                && yCoordinate >= paddleOne.getTopSize()
                && yCoordinate < paddleOne.getBottomSize()) {
            final Color color = new Color(bufferedImage.getRGB(xCoordinate,
                                                               yCoordinate));
            assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                         + yCoordinate,
                         WHITECOLOR,
```

```
                    color.getBlue());
            assertEquals("Coordinates: X:" + xCoordinate + YCOORDMESSAGE
                    + yCoordinate,
                    WHITECOLOR,
                    color.getRed());
            assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                    + yCoordinate,
                    WHITECOLOR,
                    color.getGreen());
        }
    }
```

8. Model verification

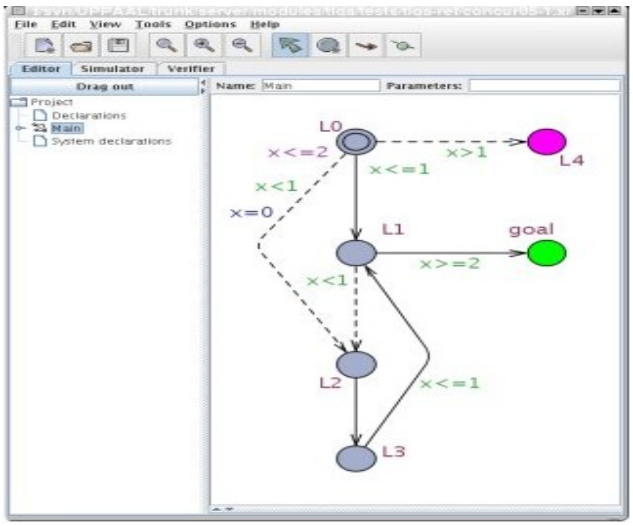
For the last part of the project we do a model verification research. Even though not all of the tools are run against the project we believe that by exploring different possibilities we will extend our knowledge regarding the verification of programs and models.

We looked at three model checking tools which listed below:

- UPPAAL TIGA
- GEAR
- JAVA PATH FINDER

The following sections will give you a brief description of each tool.

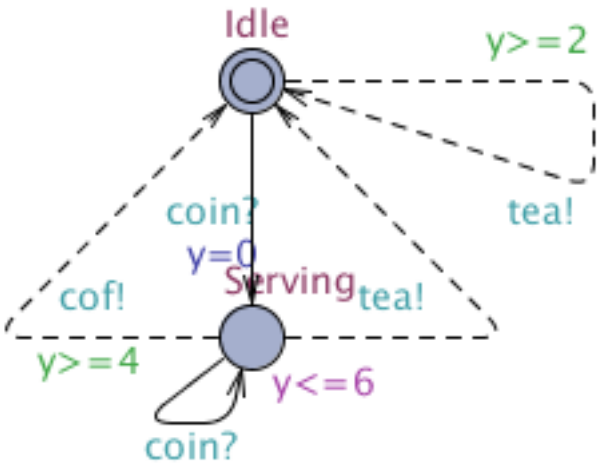
UPPAAL TIGA



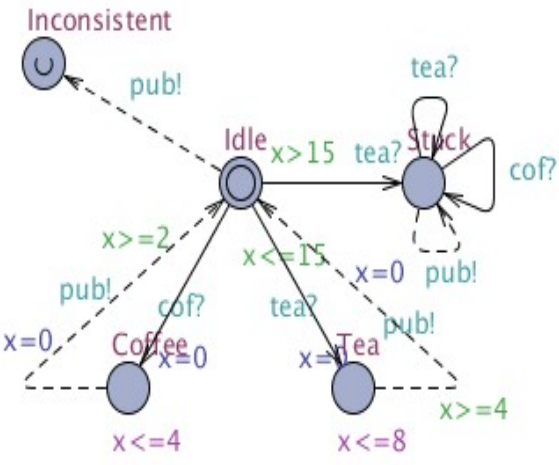
UPPAAL TIGA is an extension of the UPPAAL (http://www.uppaal.com) Model Checker open source framework. According to UPPAALTIGA's website," it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties. ". It is based on the on-the-fly algorithm invented by Liu and Smolka for linear-time model-checking of finite-state systems.

The developers of TIGA describe that they have implemented "various optimizations of the basic symbolic algorithm, as well as methods for obtaining time-optimal winning strategies (for reachability games)"

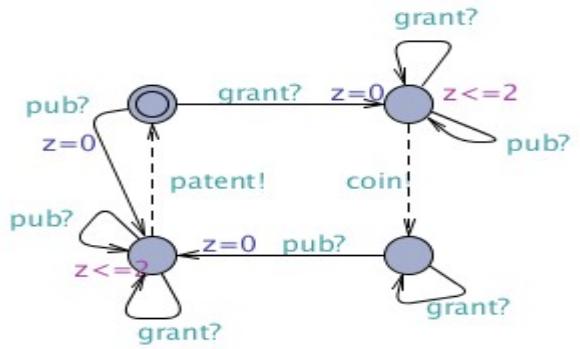
Figures bellow illustrates the notion of automaton for different objects in a university model:



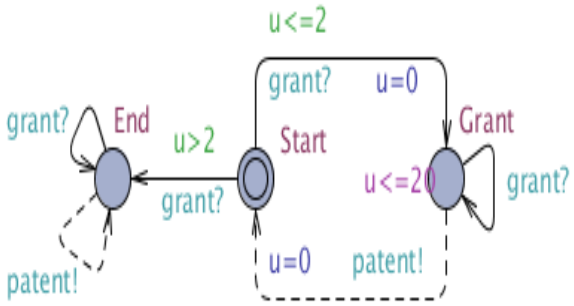
Machine automaton.



Researcher automaton.



University automaton.

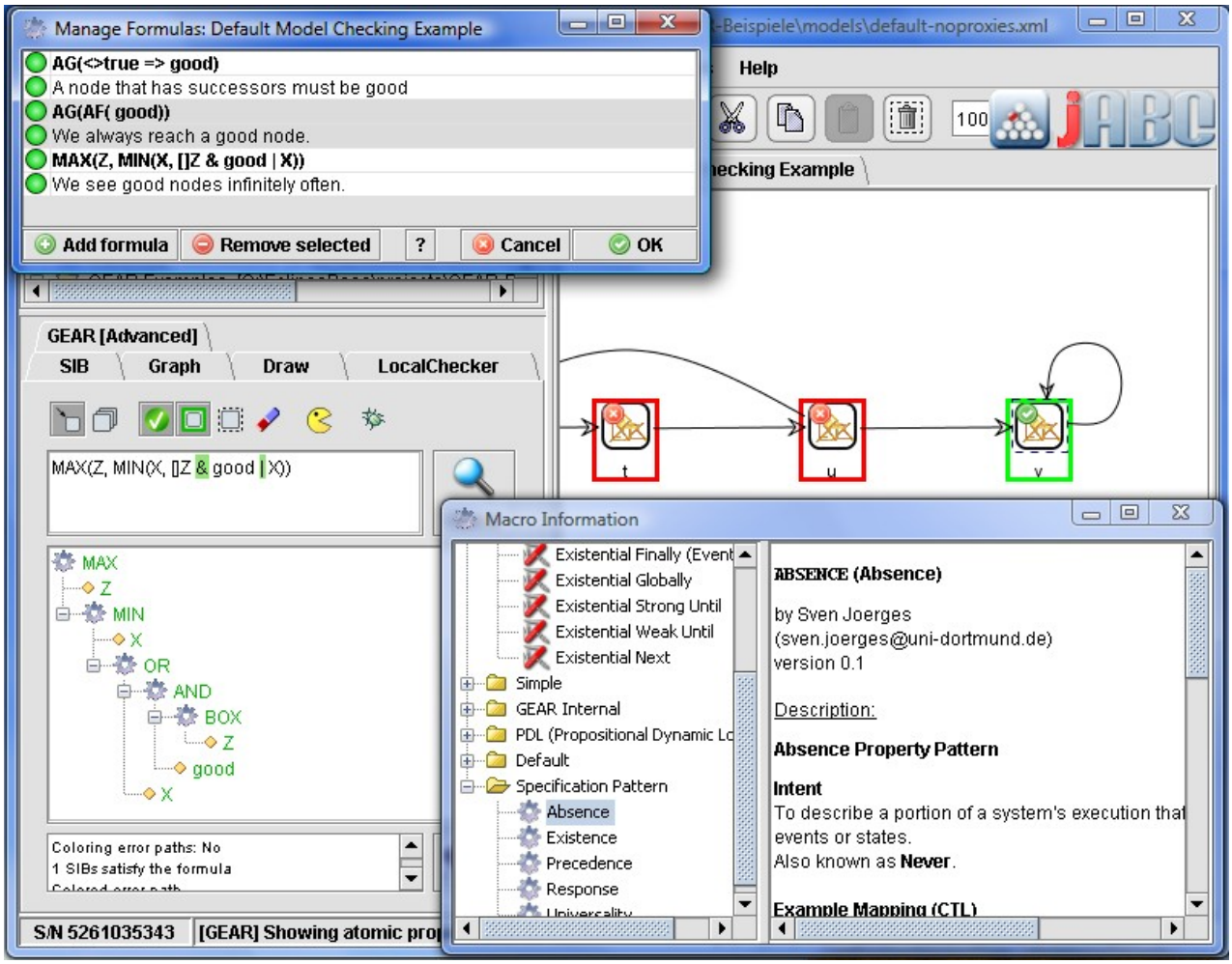


Spec automaton.

Grammar example in TIGA

Property := 'consistency:' System | 'refinement:' System '<=' System | 'specification:' System | 'implementation:' System
System := ID | '(' System Expr ')' | '(' Composition Expr ')' | '(' Conjunction Expr ')' | '(' Quotient Expr ')'
Composition := System '||' System | Composition '||' System
Conjunction := System '&&' System | Conjunction '&&' System
Quotient := System '\' System
Expr := /* nothing */ | ':' tctl

GEAR



GEAR is a verification tool for game based model verification. It provides an easy modeling syntax and an intuitive user interface shown in the picture above. GEAR is an extension of the jABC (http://jabc.cs.tu-dortmund.de) framework. Therefore it is suitable for being used in the model-driven development process.

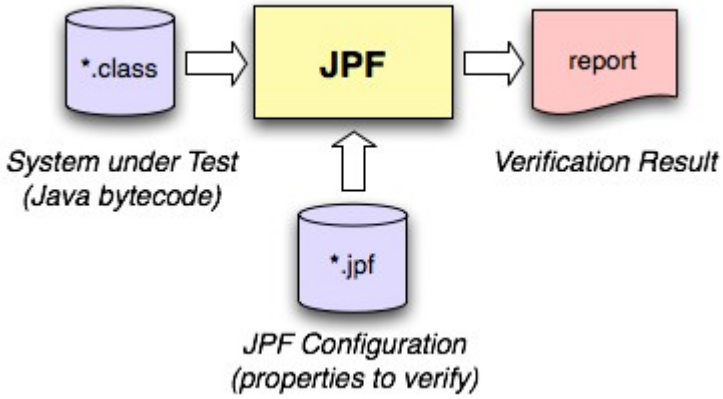
Grammar example in GEAR.

```
f ::= AP
    | f & f & ...
    | f | f | ...
    | !f
    | f => f
    | []f | [CONSTRAINT] f
    | BOXB(f) | BOXB(CONSTRAINT, f)
    | <>f | <CONSTRAINT> f
    | DIAB(f) | DIAB(CONSTRAINT, f)
    | MIN(FIX, f)
    | MAX(FIX, f)
    | TRUE | FALSE
    | MACRO(args)
```

In a conclusion, both TIGA and GEAR are suitable for model verification, however in this project we aim verification of the Java source code and neither TIGA or GEAR can be used with this purpose. They have their own domain specific languages and if we decide to use them we have to translate our model to their domain specific languages. For this reason we decide to focus our further efforts on Java Path Finder (JPF).

Java Path Finder

JPF is an open source virtual machine framework for Java bytecode verification. It can be extended and used in many kinds of verification tools.



The main use cases for JPF are:

- Software model checking (deadlock & race detection)
  - Data acquisition (random, user input)
  - Concurrency(deadlock, races)
- Deep inspection (numeric access)
  - Property annotations (Const, PbC)
  - Numeric verification (overflow, cancellation)
- Model verification
  - UML state-charts
  - Test case generation
  - Verification of distributed applications

For the project we use JPF as software model checker. When we use it as such one of the features of JPF is to cover all possible states of the objects and also all possible paths. For instance, as we describe earlier we spent a lot of time on unit testing, but with JPF we can be sure that we explore all the possible values and not only the provided test assertions.

Model checking vs. Testing (http://babelfish.arc.nasa.gov/trac/jpf/wiki)

JPF has its own JVM which enables it to cover every state of a model. It continues to investigate the model until it has checked all data combinations or has found an error.

One way of using the Java Path Finder is to create *.jpf file(s) for the model *.class file(s). These files are called application properties. JPF needs *.jpf files in order to know which main class has to execute. In addition, it is possible to define any number of arguments that specify the JPF properties you want to check. There are very different kinds of properties such as listeners, bytecode factories, search policy etc. The minimum requirement for a *jpf is the “target” element that indicates the main class.

```
#Target class
target=com.pong.manager.StartGame
```

JPF is also able to verify the model according to its default settings. In other words one can verify a model only by indicating the main class. Model checking is JPF's standard mode of operation. In other words JPF keeps track of concrete values of local variables, stack-frames, heap objects and thread states. JPF’s JVM runs on top of the standard java JVM and is of course slower than standard JVM. It is because it performs many additional tasks as checks and verifications compared with standard JVM.

The result of JPF verification is a rich report indicating all errors, traces and statistics information. The next example shows a JPF standard output run on the PONG project.

```
Executing command: java -jar C:\Users\birani\workspace\Pong\jpf-
core\build\RunJPF.jar +shell.port=4242
C:\Users\birani\workspace\Pong\src\com\pong\manager\StartGame.jpf
JavaPathfinder v5.0 - (C) 1999-2007 RIACS/NASA Ames Research Center


============================================================ system under test
application: com\pong\manager\StartGame.java

============================================================ search started:
5/23/10 1:54 PM
All objects have been created. You can play now!
============================================================ results
no errors detected

============================================================ statistics
elapsed time:       0:00:00
states:             new=1, visited=0, backtracked=0, end=1
search:             maxDepth=0, constraints=0
choice generators:  thread=1, data=0
heap:               gc=1, new=271, free=11
instructions:       3264
max memory:         15MB
loaded code:        classes=68, methods=866

============================================================ search finished:
5/23/10 1:54 PM
```

We decided to use JPF to verify the Observer class which is the core class. The result section tells us that JPF did not find any error in the model.

To prove the correctness of the verification, we introduce some intentional errors. In the next example we intentionally remove the Ball object and wait for JPF to discover the error. From the output below we can see that JPF finds the error and reports it in the verification output.

```
Executing command: java -jar C:\Users\birani\workspace\Pong\jpf-
core\build\RunJPF.jar +shell.port=4242
C:\Users\birani\workspace\Pong\src\com\pong\observer\ObserverJPF.jpf
JavaPathfinder v5.0 - (C) 1999-2007 RIACS/NASA Ames Research Center


====================================================== system under test
application: com\pong\observer\ObserverJPF.java

====================================================== search started:
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.NullPointerException: calling 'move()V' on null object
      at com.pong.observer.Observer.observe(Observer.java:82)
      at com.pong.observer.ObserverJPF.main(ObserverJPF.java:129)


====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [2860 insn w/o sources]

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
"java.lang.NullPointerException: calling 'move()V' ..."

====================================================== statistics
elapsed time:       0:00:00
states:             new=0, visited=1, backtracked=0, end=0
search:             maxDepth=0, constraints=0
choice generators:  thread=1, data=0
heap:               gc=0, new=274, free=0
instructions:       2860
max memory:         15MB
loaded code:        classes=71, methods=878

====================================================== search finished:
5/23/10 5:03 PM
```

As we mentioned above there are many kinds of properties one can use to customize the verification with JPF. Here we examine the property “enumerate_random”.

```
#Target class
target=com.pong.observer.ObserverJPF
cg.enumerate_random = true
report.console.property_violation=error,trace
```

By using this property we ask JPF to consider all possible values for expressions in the model. It means when the model allows unpredictable input, for example random values, JPF is able to examine all possible values and report defects. Running this property on our model do not return a defect.

The next example shows another use of JPF. The assert keyword is one the JPF native annotations. This example shows how JPF detects a property violation.

For the code below we intentionally set the value of the X position to - 1 and we verify with JPF.

```java
/**
  * Gets the X position.
  *
  * @return  the X position
  */

 public final int getPosX() {
     assert(this.posX > 0 );
     return this.posX;
 }
```

```
Executing command: java -jar C:\Users\birani\workspace\Pong\jpf-
core\build\RunJPF.jar +shell.port=4242
C:\Users\birani\workspace\Pong\src\com\pong\observer\ObserverJPF.jpf
JavaPathfinder v5.0 - (C) 1999-2007 RIACS/NASA Ames Research Center
====================================================== system under test
application: com\pong\observer\ObserverJPF.java

====================================================== search started
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.AssertionError
      at com.pong.players.Ball.setPosX(Ball.java:171)
      at com.pong.observer.ObserverJPF.main(ObserverJPF.java:134)
====================================================== trace #1
------------------------------------------------------ transition #0
thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [3431 insn w/o sources]

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
"java.lang.AssertionError  at com.pong.players.Ball..."

====================================================== statistics
elapsed time:       0:00:00
states:             new=0, visited=1, backtracked=0, end=0
search:             maxDepth=0, constraints=0
choice generators:  thread=1, data=0
heap:               gc=0, new=304, free=0
instructions:       3431
max memory:         15MB
loaded code:        classes=74, methods=1052

====================================================== search finished:
5/24/10 8:33 AM
```

Conclusion:

JPF is an advanced model checker with a huge set of abilities. There are very different ways of using JPF and it needs a full focus to explorer it in depth. This paper is not to describe JPF. The goal is to use JPF as a complimentary verification of the PONG model.

Appendix

The Manager class

```
/**
 * The Manager class.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Manager extends Applet implements Runnable {
    //~ Static fields/initializers ---------------------------------------------

      /** The Constant serialVersionUID. */
    private static final long serialVersionUID = 1L;

    /** The Constant TABLEWIDTH. */
    private static final int TABLEWIDTH = 500;

    /** The Constant TABLEHEIGTH. */
    private static final int TABLEHEIGHT = 500;

    /** The Constant PADDLEWIDTH. */
    private static final int PADDLEWIDTH = 10;

    /** The Constant PADDLEHEIGHT. */
    private static final int PADDLEHEIGHT = 30;

    /** The Constant BALLWIDTH. */
    private static final int BALLWIDTH = 10;

    /** The Constant BALLHEIGHT. */
    private static final int BALLHEIGHT = 10;

    /** The Constant PLAYERONEKEYDOWN. */
    private static final int PLAYERONEKEYDOWN = 97;

    /** The Constant PLAYERONEKEYUP. */
    private static final int PLAYERONEKEYUP = 113;

    /** The Constant THREADSLEEPTIME. */
    private static final int THREADSLEEPTIME = 6;

    /** The Constant INITIALX. */
    private static final int INITIALX = 10;

    /** The Constant INITIALY. */
    private static final int INITIALY = 250;

    /** The Constant BALLSIZE. */
    private static final int BALLSIZE = 10;

    /** The Constant BALLSPEED. */
    private static final int BALLSPEED = 2;

    /** The Constant PADDLESPEED. */
    private static final int PADDLESPEED = 4;
```

```
/** The Constant CIRCLEPERIOD. */
private static final int CIRCLEPERIOD = 50;

//~ Instance fields ------------------------------------------------------

/** The paddle one. */
private transient Paddle paddleOne;

/** The paddle two. */
private transient Paddle paddleTwo;

/** The observer. */
private transient Observer observer;

/** The ball. */
private transient Ball ball;

/** The count. */
private transient int count;

/** The graphics. */
private transient Graphics graphics;

/** The buffered image. */
private transient BufferedImage bufferedImage;

/** The the background image. */
private transient Image image;

/** The canvas graphics. */
private transient Graphics canvasGraphics;

/** The in run. */
private transient boolean inRun;

//~ Methods --------------------------------------------------------------

/**
 * Initialise.
 */
@Override
public final void init() {
    this.resize(TABLEWIDTH, TABLEHEIGHT);
    final Canvas canvas = new Canvas();
    canvas.setSize(TABLEWIDTH, TABLEHEIGHT);
    add(canvas);

    this.canvasGraphics = canvas.getGraphics();

    this.bufferedImage = new BufferedImage(TABLEWIDTH,
                                           TABLEHEIGHT,
                                           BufferedImage.TYPE_INT_RGB);

    this.graphics = this.bufferedImage.createGraphics();


    final URL url =
        getClass().getResource("/com/pong/sounds/PluckyDaisy.wav");
```

```
        final AudioClip music = newAudioClip(url);

        if (music != null) {
            music.loop();
        }

        this.paddleOne = new Paddle(INITIALX,
                                    INITIALY,
                                    PADDLEWIDTH,
                                    PADDLEHEIGHT,
                                    TABLEHEIGHT,
                                    PADDLESPEED);

        this.paddleTwo = new Paddle(TABLEWIDTH - (2 * INITIALX),
                                    INITIALY,
                                    PADDLEWIDTH,
                                    PADDLEHEIGHT,
                                    TABLEHEIGHT,
                                    PADDLESPEED);

        this.ball = new Ball(TABLEWIDTH / 2,
                             TABLEHEIGHT
                                 / 2,
                             BALLSIZE,
                             BALLSIZE,
                             BALLSPEED,
                             TABLEWIDTH,
                             TABLEHEIGHT);

        this.observer = new Observer(TABLEHEIGHT,
                                     TABLEWIDTH,
                                     this.paddleOne,
                                     this.paddleTwo,
                                     this.ball,
                                     PADDLEHEIGHT);

        final URL iurl =
                  getClass().getResource("/com/pong/pics/background.gif");

        this.image = getImage(iurl);

        this.inRun = true;

        final Thread animate = new Thread(this);
        animate.start();
    }


    /**
     * Runs the game.
     */
    public final void run() {
        while (inRun) {

            this.graphics = this.bufferedImage.createGraphics();

            tablePaint();

            if (this.observer != null) {
                this.observer.observe();
```

```
        }

        makeOldEffect();

        try {
            Thread.sleep(THREADSLEEPTIME);
        } catch (InterruptedException e) {
            // Ignore, it is not harmful for our system.
        }
    }
}


/**
 * Table_paint.
 */
private void tablePaint() {

    if (this.graphics != null) {
        this.graphics.setColor(Color.WHITE);
        this.graphics.drawImage(image, 0, 0, this);
    }

    if (paddleOne != null && this.graphics != null) {
        this.graphics.fillRect(this.paddleOne.getPosX(),
                               this.paddleOne.getPosY(),
                               PADDLEWIDTH,
                               PADDLEHEIGHT);
    }

    if (paddleTwo != null && this.graphics != null) {
        this.graphics.fillRect(this.paddleTwo.getPosX(),
                               this.paddleTwo.getPosY(),
                               PADDLEWIDTH,
                               PADDLEHEIGHT);
    }

    if (this.ball != null && this.graphics != null) {
        this.graphics.fillRect(this.ball.getPosX(),
                               this.ball.getPosY(),
                               BALLWIDTH,
                               BALLHEIGHT);
    }
}


/**
 * Make the old movie effect.
 */
private void makeOldEffect() {
    if (graphics != null) {
        count++;
        this.graphics.setColor(Color.WHITE);
        int temp = new Random().nextInt(TABLEHEIGHT);
        this.graphics.drawLine(temp, 0, temp, TABLEHEIGHT);

        if (count == CIRCLEPERIOD) {
            temp = new Random().nextInt(TABLEHEIGHT);
            this.graphics.fillOval(temp, temp, temp, 2 * temp);
            count = 0;
```

```
        }

        if (this.canvasGraphics != null && this.bufferedImage != null) {
            this.canvasGraphics.drawImage(this.bufferedImage, 0, 0, this);
        }
    }
}


/**
 * Finds if a control key is pressed.
 *
 * @param   evt  the key event
 * @param   key  the value of the key
 *
 * @return  true, if successful
 */
@Override
public final boolean keyDown(final Event evt, final int key) {
    if (this.paddleOne != null && this.paddleTwo != null) {
        if (key == PLAYERONEKEYDOWN) {
            this.paddleOne.setDirection(1);
        }

        if (key == PLAYERONEKEYUP) {
            this.paddleOne.setDirection(-1);
        }

        if (key == Event.DOWN) {
            this.paddleTwo.setDirection(1);
        }

        if (key == Event.UP) {
            this.paddleTwo.setDirection(-1);
        }
    }
    return true;
}


/**
 * Finds if a key is released.
 *
 * @param   evt  the key event
 * @param   key  the value of the key
 *
 * @return  true, if successful
 */
@Override
public final boolean keyUp(final Event evt, final int key) {
    if (evt.id == Event.KEY_ACTION_RELEASE && this.paddleOne != null
            && this.paddleTwo != null) {
        this.paddleTwo.setDirection(0);
    } else {
        if (paddleTwo != null) {
            this.paddleOne.setDirection(0);
        }
    }

    return true;
```

```
    }
}
```

The Observer class

```
/**
 * The Class Observer.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Observer {

      /** The Constant BALLPADDLEDIF. */
      private static final transient int BALLPADDLEDIF = 3;

    //~ Instance fields --------------------------------------------------------

    /** The table width. */
      /*@ spec_public @*/ private final transient int tableWidth;

    /** The table height. */
      /*@ spec_public @*/ private final transient int tableHeight;

    /*@ non_null @*/ /*@ spec_public @*/
      /** The paddle one. */
      private final transient Paddle paddleOne;

    /*@ non_null @*/ /*@ spec_public @*/
      /** The paddle two. */
      private final transient Paddle paddleTwo;

    /*@ spec_public @*/
      /** The paddle height. */
      private final transient int paddleHeight;

    /*@ non_null @*/ /*@ spec_public @*/
      /** The ball. */
      private final transient Ball ball;

    //~ Constructors -----------------------------------------------------------

    /**
     * Creates a new Observer object.
     *
     * @param  newTableHeight   the new table height
     * @param  newTableWidth    the new table width
     * @param  newPaddleOne     the new paddle one
     * @param  newPaddleTwo     the new paddle two
     * @param  newBall          the new ball
     * @param  newPaddleHeight  the new paddle height
     */
    //@ requires newPaddleOne != null;
    //@ requires newPaddleTwo !=null;
    //@ requires newBall != null;
    //@ ensures this != null;
    //@ ensures this.tableWidth == newTableWidth;
```

```
//@ ensures this.tableHeight == newTableHeight;
//@ ensures this.paddleHeight == newPaddleHeight;
public Observer(final int newTableHeight,
                final int newTableWidth,
                final Paddle newPaddleOne,
                final Paddle newPaddleTwo,
                final Ball newBall,
                final int newPaddleHeight) {
    this.tableWidth = newTableWidth;
    this.tableHeight = newTableHeight;
    this.paddleOne = newPaddleOne;
    this.paddleTwo = newPaddleTwo;
    this.ball = newBall;
    this.paddleHeight = newPaddleHeight;
}

//~ Methods ----------------------------------------------------------------

/**
 * Observe.
 */
//@ requires this.ball != null;
//@ requires this.paddleOne != null;
//@ requires this.paddleTwo != null;
//@ requires this.tableWidth > 0;
//@ requires this.tableHeight >0;
public final void observe() {

  //Move the ball;
    this.ball.move();

    // Set the Paddle One direction
    this.paddleOne.setPaddlePosY(this.paddleOne.getPosY()
                                 + this.paddleOne.getDirection());

    // If Paddle One is not at the end of the PlayGroung
    // move the paddle
    if ((this.paddleOne.getPosY() == this.tableHeight - this.paddleHeight)
                            || (this.paddleOne.getPosY() <= 0)) {
        this.paddleOne.setPaddlePosY(this.paddleOne.getPosY()
                                 - this.paddleOne.getDirection());
    }

    // Calculate Paddle One corners
    this.paddleOne.calcCorners(this.paddleOne.getPosX(),
                                 this.paddleOne.getPosY());


    // Set the Paddle Two direction
    this.paddleTwo.setPaddlePosY(this.paddleTwo.getPosY()
                            + this.paddleTwo.getDirection());


    // If Paddle Two is not at the end of the PlayGroung
    // move the paddle
    if ((this.paddleTwo.getPosY() == this.tableHeight - this.paddleHeight)
                                 || (this.paddleTwo.getPosY() <= 0)) {
        this.paddleTwo.setPaddlePosY(this.paddleTwo.getPosY()
                            - this.paddleTwo.getDirection());
    }
```

```
    // Calculate Paddle Two corners
    this.paddleTwo.calcCorners(this.paddleTwo.getPosX(),

  this.paddleTwo.getPosY());

    checkForAWallHit();
    checkForAPoint();
    checkBallLevelEqualPaddleOneLevel();
    checkBallLevelEqualPaddleTwoLevel();
}

/**
 * Checks if is a wall hit.
 */
private void checkForAWallHit() {
   // If the Ball is at the top or at the bottom of the PlayGround
    // hit the wall
    if (this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0
        && this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
                                                <= this.ball.getGroundYSize()
        && (this.ball.getTopSize() <= 0
                    || this.ball.getBottomSize() > this.tableHeight)) {
              hitWalls();
    }
}

/**
 * Checks if is there a point.
 */
private void checkForAPoint() {
  // If the Ball is out of the PlayGround make a point
    if ((this.ball.getPosX() <= 0 || this.ball.getPosX() >= this.tableWidth)
        && this.tableWidth / 2 >= 0
        && this.tableWidth / 2 <= this.ball.getGroundXSize()
        && this.tableHeight / 2 >= 0
        && this.tableHeight / 2 <= this.ball.getGroundYSize()) {
        restartBall();
    }
}

/**
 * Checks if is paddle one hit.
 */
private void checkBallLevelEqualPaddleOneLevel() {
  if (this.paddleOne.getTopSize() <= this.ball.getBottomSize()
        && this.paddleOne.getBottomSize() >= this.ball.getTopSize()) {
              paddleOneHit();
        }
}

/**
 * Checks if is paddle.
 */
private void paddleOneHit() {
  if (this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0
              && this.ball.getPosX() + (this.ball.getDeltaX() * (-1)) >= 0
              && this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
                                              <= this.ball.getGroundYSize()
            && this.ball.getPosX() + (this.ball.getDeltaX() * (-1))
```

```
                                                <= this.ball.getGroundXSize()
                && this.paddleOne.getRightCorner()
                                                - this.ball.getLeftCorner() > 0
                && (this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
                                                > BALLPADDLEDIF
                || this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
                                                < BALLPADDLEDIF)) {
            hitPaddleOne();
      }
    }


    /**
     * Checks if is paddle one hit.
     */
    private void checkBallLevelEqualPaddleTwoLevel() {
      if (this.paddleTwo.getTopSize() <= this.ball.getBottomSize()
            && this.paddleTwo.getBottomSize() >= this.ball.getTopSize()) {
                  paddleTwoHit();
            }
    }


    /**
     * Checks if is paddle two hit.
     */
    private void paddleTwoHit() {
      if (this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0
                  && this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
                                                      <=
this.ball.getGroundYSize()
                && this.ball.getPosX() + (this.ball.getDeltaX() * (-1)) >= 0
                && this.ball.getPosX() + (this.ball.getDeltaX() * (-1))
                                                      <=
this.ball.getGroundXSize()
                  && this.ball.getRightCorner()
                                                      -
this.paddleTwo.getLeftCorner() > 0
                  && (this.ball.getRightCorner() -
this.paddleTwo.getLeftCorner()
                                                      > BALLPADDLEDIF
                || this.ball.getRightCorner() - this.paddleTwo.getLeftCorner()
                                                      < BALLPADDLEDIF)) {
                        hitPaddleTwo();
            }
    }


    /*@ private normal_behavior
      @ requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
      @ requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
      @                                         <= this.ball.getGroundYSize();
      @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
      @                                         > BALLPADDLEDIF;
      @ assignable \not_specified;
      @ ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
      @
      @ also
      @
      @ private normal_behavior
```

```
      @ requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1)) >= 0;
      @ requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1))
      @                                          <= this.ball.getGroundXSize();
      @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
      @                                                            > 0;
      @ requires this.paddleOne.getRightCorner() - this.ball.getLeftCorner()
      @                                          < BALLPADDLEDIF;
      @ assignable \not_specified;
      @ ensures this.ball.getDeltaX() == \old(this.ball.getDeltaX()) * (-1);
      @*/
    /**
     * Hit paddle one.
     */
    private void hitPaddleOne() {
            final int diffX = this.paddleOne.getRightCorner()
                                                                    -
this.ball.getLeftCorner();

        if (diffX > 0) {
           if (diffX > BALLPADDLEDIF) {
                this.ball.changeVertDirection();
           } else {
               this.ball.changeHorzDirection();
            }
         }
    }


    /*@ private normal_behavior
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
      @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
      @                                     <= this.ball.getGroundYSize();
      @     requires this.ball.getRightCorner() - this.paddleTwo.getLeftCorner()
      @                                                     > BALLPADDLEDIF;
      @     assignable \not_specified;
      @     ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
      @
      @ also
      @
      @ private normal_behavior
      @     requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1)) >= 0;
      @     requires this.ball.getPosX() + (this.ball.getDeltaX() * (-1))
      @                                     <= this.ball.getGroundXSize();
      @     requires this.ball.getRightCorner() - this.paddleTwo.getLeftCorner()
      @                                                            > 0;
      @     requires this.ball.getRightCorner() - this.paddleTwo.getLeftCorner()
      @                                     < BALLPADDLEDIF;
      @     assignable \not_specified;
      @     ensures this.ball.getDeltaX() == \old(this.ball.getDeltaX()) * (-1);
      @*/
    /**
     * Hit paddle two.
     */
    private void hitPaddleTwo() {
        final int diffX = this.ball.getRightCorner()
                                                                    -
this.paddleTwo.getLeftCorner();

            if (diffX > 0) {
              if (diffX > BALLPADDLEDIF) {
```

```
            this.ball.changeVertDirection();
          } else {
            this.ball.changeHorzDirection();
          }
        }
}


/*@ private normal_behavior
  @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
  @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
  @                                   <= this.ball.getGroundYSize();
  @     requires this.ball.getTopSize() <= 0;
  @     assignable \not_specified;
  @     ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
  @
  @ also
  @
  @ private normal_behavior
  @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1)) >= 0;
  @     requires this.ball.getPosY() + (this.ball.getDeltaY() * (-1))
  @                                    <= this.ball.getGroundYSize();
  @     requires this.ball.getBottomSize() > this.tableHeight;
  @     assignable \not_specified;
  @     ensures this.ball.getDeltaY() == \old(this.ball.getDeltaY()) * (-1);
  @ */
/**
 * Hit walls.
 */
private void hitWalls() {
  if (this.ball.getTopSize() <= 0) {
             this.ball.changeVertDirection();
  }

    if (this.ball.getBottomSize() > this.tableHeight) {
            this.ball.changeVertDirection();
    }
}

/*@ private normal_behavior
  @ requires this.ball.getPosX() <= 0;
  @ requires this.tableWidth / 2 >= 0;
  @ requires this.tableWidth / 2 <= this.ball.getGroundXSize();
  @ requires this.tableHeight / 2 >= 0;
  @ requires this.tableHeight / 2 <= this.ball.getGroundYSize();
  @ assignable \not_specified;
  @ ensures this.ball.getPosX() == this.tableWidth / 2;
  @ ensures this.ball.getPosY() == this.tableHeight / 2;
  @
  @ also
  @
  @ private normal_behavior
  @ requires this.ball.getPosX() >= this.tableWidth;
  @ requires this.tableWidth / 2 >= 0;
  @ requires this.tableWidth / 2 <= this.ball.getGroundXSize();
  @ requires this.tableHeight / 2 >= 0;
  @ requires this.tableHeight / 2 <= this.ball.getGroundYSize();
  @ assignable \not_specified;
  @ ensures this.ball.getPosX() == this.tableWidth / 2;
  @ ensures this.ball.getPosY() == this.tableHeight / 2;
```

```
      @ */
    /**
     * Start volley.
     */
    private void restartBall() {
            if (this.ball.getPosX() <= 0
                                    || this.ball.getPosX() >= this.tableWidth) {
                  this.ball.jumpTo(this.tableWidth / 2, this.tableHeight / 2);
            }
    }
}
```

The Ball class

```
/**
 * The Ball class.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Ball {
    //~ Instance fields ----------------------------------------------------------

    /** The delta value of the Ball(speed and direction). */
      /*@ spec_public @*/ private transient int deltaX = 1;

      /** The delta value of the Ball(speed and direction). */
      /*@ spec_public @*/ private transient int deltaY = 1;

    /** The current X position of the Ball. */
      /*@ spec_public @*/ private transient int posX;

    /** The current Y position of the Ball. */
      /*@ spec_public @*/ private transient int posY;

    /** The left corner of the Ball. */
      /*@ spec_public @*/ private transient int leftCorner;

    /** The right corner of the ball. */
      /*@ spec_public @*/ private transient int rightCorner;

    /** The top position of the Ball. */
      /*@ spec_public @*/ private transient int topSize;

    /** The bottom position of the Ball. */
      /*@ spec_public @*/ private transient int bottomSize;

    /** The height of the Ball. */
      /*@ spec_public @*/ private final transient int ySize;

    /** The width of the Ball. */
      /*@ spec_public @*/ private final transient int xSize;

      /** The ground X size. */
      /*@ spec_public @*/ private final transient int groundXSize;

      /** The ground Y size. */
      /*@ spec_public @*/ private final transient int groundYSize;
```

```
//~ Constructors ----------------------------------------------------------

/**
 * Constructor.
 *
 * @param  newXPos      the new x position of the Ball
 * @param  newYPos      the new y position of the Ball
 * @param  height       the height of the Ball
 * @param  width        the width of the Ball
 * @param  speed        the speed of the Ball
 * @param  tableWidth   the width of the PlayGround
 * @param  tableHeight  the height of the PlayGround
 */
  //@ requires speed > 0;
  //@ requires newXPos >= 0;
  //@ requires tableWidth >= 0;
  //@ requires newXPos <= tableWidth;
  //@ requires newYPos <= tableHeight;
  //@ requires newYPos >= 0;
  //@ ensures this != null;
  //@ ensures this.groundXSize == tableWidth;
  //@ ensures this.groundYSize == tableHeight;
  //@ ensures this.posX >= 0;
  //@ ensures this.posX <= this.groundXSize;
  //@ ensures this.posY <= this.groundYSize;
  //@ ensures getPosX() == newXPos;
  //@ ensures getPosY() == newYPos;
  //@ ensures this.ySize == height;
  //@ ensures this.xSize == width;
public Ball(final int newXPos,
            final int newYPos,
            final int height,
            final int width,
            final int speed,
            final int tableWidth,
            final int tableHeight) {
    this.posX = newXPos;
    this.posY = newYPos;
    this.ySize = height;
    this.xSize = width;
    this.deltaX = 1;
    this.deltaY = 1;
    this.groundXSize = tableWidth;
    this.groundYSize = tableHeight;
    setSpeed(speed);
    calcCorners(this.posX, this.posY);
}

//~ Methods ----------------------------------------------------------------

/**
 * Moves the ball.
 */
public final void move() {
  if (this.getPosX() + this.deltaX >= 0
        && this.getPosX() + this.deltaX <= this.groundXSize
        && this.getPosY() + this.deltaY >= 0
        && this.getPosY() + this.deltaY <= this.groundYSize) {
        this.posX += this.deltaX;
```

```
        this.posY += this.deltaY;
        if (this.posX >= 0) {
              calcCorners(posX, posY);
        }
  }
}


/**
 * Calculates the Ball's corners.
 *
 * @param  newPosX  the new X position
 * @param  newPosY  the new Y position
 */
  //@ requires newPosX >= 0;
//@ requires newPosX <= this.groundXSize;
  //@ requires newPosY >= 0;
//@ ensures getLeftCorner() == newPosX;
//@ ensures getRightCorner() == newPosX + this.xSize;
//@ ensures getTopSize() == newPosY;
//@ ensures getBottomSize() == newPosY + ySize;
private void calcCorners(final int newPosX, final int newPosY) {
    this.leftCorner = newPosX;
    this.rightCorner = newPosX + this.xSize;
    this.topSize = newPosY;
    this.bottomSize = newPosY + this.ySize;
}


/**
 * Changes the horizontal direction of the Ball.
 */
//@ requires this.getPosX() + (this.getDeltaX() * (-1)) >= 0;
//@ requires this.getPosX() + (this.getDeltaX() * (-1)) <= this.groundXSize;
//@ ensures getDeltaX() == \old(getDeltaX()) * (-1);
//@ ensures getPosX() == \old(getPosX()) + getDeltaX();
public final void changeHorzDirection() {
  this.deltaX = this.getDeltaX() * (-1);
    this.posX = this.posX + this.deltaX;
}


/**
 * Changes the vertical direction of the Ball.
 */
//@ requires this.getPosY() + (this.getDeltaY() * (-1)) >= 0;
//@ requires this.getPosY() + (this.getDeltaY() * (-1)) <= this.groundYSize;
//@ ensures getDeltaY() == \old(getDeltaY()) * (-1);
//@ ensures getPosY() == \old(getPosY()) + getDeltaY();
public final void changeVertDirection() {
        this.deltaY = this.getDeltaY() * (-1);
        this.posY = this.posY + this.deltaY;
}


/**
 * Sets the Ball's speed.
 *
 * @param  ballSpeed  the new ball speed
 */
```

```
  //@ requires ballSpeed > 0;
  private void setSpeed(final int ballSpeed) {
    if (this.deltaX > 0 && this.deltaY > 0) {
          this.deltaX = (this.deltaX / this.deltaX) * ballSpeed;
          this.deltaY = (this.deltaY / this.deltaY) * ballSpeed;
    }
  }


  /**
   * Jump to new position - puts the Ball on a new position.
   *
   * @param  xCoord  the new X coordinate
   * @param  yCoord  the new Y coordinate
   */
  //@ requires xCoord >= 0;
  //@ requires yCoord >= 0;
  //@ requires xCoord <= this.groundXSize;
  //@ requires yCoord <= this.groundYSize;
  //@ ensures getPosX() == xCoord;
  //@ ensures getPosY() == yCoord;
  public final void jumpTo(final int xCoord, final int yCoord) {
      this.posX = xCoord;
      this.posY = yCoord;
  }


  /**
   * Gets the X position of the Ball.
   *
   * @return  the X position
   */
//@ ensures \result == this.posX;
  public final /*@ pure @*/ int getPosX() {
      return this.posX;
  }


  /**
   * Gets the Y position of the Ball.
   *
   * @return  the Y position
   */
  //@ ensures \result == this.posY;
  public final /*@ pure @*/ int getPosY() {
      return this.posY;
  }


  /**
   * Gets the left corner of the Ball.
   *
   * @return  the value of the left corner
   */
  //@ ensures \result == this.leftCorner;
  public final /*@ pure @*/ int getLeftCorner() {
      return this.leftCorner;
  }
```

```java
/**
 * Gets the right corner of the Ball.
 *
 * @return  the value  of the right corner
 */
//@ ensures \result == this.rightCorner;
public final /*@ pure @*/ int getRightCorner() {
    return this.rightCorner;
}


/**
 * Gets the top position of the Ball.
 *
 * @return  the top position of the Ball
 */
//@ ensures \result == this.topSize;
public final /*@ pure @*/ int getTopSize() {
    return this.topSize;
}


/**
 * Gets the bottom position of the ball.
 *
 * @return  the bottom position of the ball
 */
//@ ensures \result == this.bottomSize;
public final /*@ pure @*/ int getBottomSize() {
    return this.bottomSize;
}


/**
 * Gets the delta X.
 *
 * @return the delta X
 */
//@ ensures \result == this.deltaX;
public final /*@ pure @*/  int getDeltaX() {
        return deltaX;
  }

  /**
   * Gets the delta Y.
   *
   * @return the delta Y
   */
//@ ensures \result == this.deltaY;
  public final /*@ pure @*/  int getDeltaY() {
        return deltaY;
  }

  /**
   * Gets the ground X size.
   *
   * @return the ground X size
   */
  //@ ensures \result == this.groundXSize;
  public final /*@ pure @*/ int getGroundXSize() {
```

```
            return groundXSize;
      }

      /**
       * Gets the ground Y size.
       *
       * @return the ground Y size
       */
      //@ ensures \result == this.groundYSize;
      public final /*@ pure @*/ int getGroundYSize() {
            return groundYSize;
      }

      //@ invariant this.posX >= 0;
      //@ invariant this.posX <= this.groundXSize;
      //@ invariant this.posY >= 0;
      //@ invariant this.posY <= this.groundYSize;
      ////@ set this.owner = Observer.class;
      ////@ invariant \typeof(this.owner) == Observer.class;
}
```

The Paddle class

```
/**
 * The Paddle class.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class Paddle {
    //~ Instance fields --------------------------------------------------------

     //@ ghost int playBottomSize;

    /** The current X position of the paddle. */
      /*@ spec_public @*/ private final transient int posX;

    /** The current Y position of the paddle. */
      /*@ spec_public @*/ private transient int posY;

    /** The height of the paddle. */
      /*@ spec_public @*/ private final transient int ySize;

    /** The width of the paddle. */
      /*@ spec_public @*/ private final transient int xSize;

    /** The speed of the paddle. */
      /*@ spec_public @*/ private final transient int speed;

    /** The left corner of the paddle. */
      /*@ spec_public @*/ private transient int leftCorner;

    /** The right corner of the paddle. */
      /*@ spec_public @*/ private transient int rightCorner;

    /** The top size of the paddle. */
```

```
  /*@ spec_public @*/ private transient int topSize;

/** The bottom size of the paddle. */
  /*@ spec_public @*/ private transient int bottomSize;

/** The direction of the paddle. */
  /*@ spec_public @*/ private transient int direction;

//~ Constructors ----------------------------------------------------------

/**
 * Instantiates a new paddle.
 *
 * @param  xCoord       the initial X coordinates
 * @param  yCoord       the initial Y coordinates
 * @param  width        the width
 * @param  height       the height
 * @param  pBottomSize  the bottom size
 * @param  newSpeed     the new speed
 */
  //@ ensures this != null;
  //@ ensures getPosX() == xCoord;
  //@ ensures getPosY() == yCoord;
  //@ ensures this.xSize == width;
  //@ ensures this.ySize == height;
  //@ ensures this.playBottomSize == pBottomSize;
public Paddle(final int xCoord,
              final int yCoord,
              final int width,
              final int height,
              final int pBottomSize,
              final int newSpeed) {
    this.posX = xCoord;
    this.posY = yCoord;
    this.xSize = width;
    this.ySize = height;
    this.speed = newSpeed;
    this.setDirection(0);
    calcCorners(this.posX, this.posY);
    //@ set playBottomSize = pBottomSize;
}

//~ Methods ----------------------------------------------------------------


/**
 * Calculates the paddle's corners.
 *
 * @param  newPosX  the new X position
 * @param  newPosY  the new Y position
 */
//@ ensures getLeftCorner() == newPosX;
//@ ensures getRightCorner() == newPosX + this.xSize;
//@ ensures getTopSize() == newPosY;
//@ ensures getBottomSize() == newPosY + this.ySize;
public final void calcCorners(final int newPosX, final int newPosY) {
    this.leftCorner = newPosX;
    this.rightCorner = newPosX + this.xSize;
    this.topSize = newPosY;
    this.bottomSize = newPosY + ySize;
```

```
}


/**
 * Gets the left corner of the paddle.
 *
 * @return  the value of the left corner
 */
//@ ensures \result == this.leftCorner;
public final /*@ pure @*/ int getLeftCorner() {
    return this.leftCorner;
}


/**
 * Gets the right corner of the paddle.
 *
 * @return  the value of the right corner
 */
//@ ensures \result == this.rightCorner;
public final /*@ pure @*/ int getRightCorner() {
    return this.rightCorner;
}


/**
 * Gets the top position of the paddle.
 *
 * @return  the value of the top size
 */
  //@ ensures \result == this.topSize;
public final /*@ pure @*/ int getTopSize() {
    return this.topSize;
}


/**
 * Gets the bottom position of the paddle.
 *
 * @return  the value of the bottom size
 */
//@ ensures \result == this.bottomSize;
public final /*@ pure @*/ int getBottomSize() {
    return this.bottomSize;
}


/**
 * Sets the paddle's direction.
 *
 * @param  newDirection  the new direction
 */
//@ assignable this.direction;
//@ ensures this.direction == this.speed * newDirection;
public final void setDirection(final int newDirection) {
    this.direction = (this.speed * newDirection);
}


/**
```

```
     * Gets the  X position of the paddle.
     *
     * @return  the X position
     */
    //@ ensures \result == this.posX;
    public final /*@ pure @*/ int getPosX() {
        return this.posX;
    }


    /**
     * Gets the Y position of the paddle.
     *
     * @return  the Y position
     */
    //@ ensures \result == this.posY;
    public final /*@ pure @*/ int getPosY() {
        return this.posY;
    }

    /**
     * Sets the paddle's Y coordinate.
     *
     * @param newPosY the new paddle Y coordinate
     */
    //@ ensures getPosY() == newPosY;
    public final void setPaddlePosY(final int newPosY) {
      this.posY = newPosY;
    }


    /**
     * Gets the paddle's direction.
     *
     * @return the direction
     */
    //@ ensures  \result == this.direction;
    public final int getDirection() {
            return direction;
      }

    /**
     * Gets the paddle's speed.
     *
     * @return the speed
     */
  //@ ensures  \result == this.speed;
    public final int getSpeed() {
      return this.speed;
    }
}
```

ESC/Java2 Output:

ESC/Java version ESCJava-2.0.5

./players/Paddle.java: Caution: Using given file as the .java file, even though it is not the java file for com.pong.players.Paddle on the classpath

./players/Ball.java: Caution: Using given file as the .java file, even though it is not the java file for com.pong.players.Ball on the classpath

observer/Observer.java: Caution: Using given file as the .java file, even though it is not the java file for com.pong.observer.Observer on the classpath

[0.07 s 25646744 bytes]

com.pong.players.Paddle ...

Prover started:0.012 s 48816704 bytes

[0.464 s 48816704 bytes]

com.pong.players.Paddle: Paddle(int, int, int, int, int, int) ...

[prediction time: 0.019 s 49459696 bytes]

[Time: 0.224 s 30835984 bytes GC: 0.029 s 49459696 bytes DSA: 0.038 s 28260880 bytes Ejp: 0.0060 s 28260880 bytes VC: 0.0 s 28260880 bytes Proof(s): 0.151 s 30835984 bytes]

[Size: src: 202 GC: 1857 DSA: 4051 VC: 5734]

[0.229 s 30835984 bytes] passed

com.pong.players.Paddle: calcCorners(int, int) ...

[prediction time: 0.0 s 30835984 bytes]

[Time: 0.065 s 32121992 bytes GC: 0.012 s 30835984 bytes DSA: 0.0040 s 30835984 bytes Ejp: 0.0010 s 31479008 bytes VC: 0.0 s 31479008 bytes Proof(s): 0.048 s 32121992 bytes]

[Size: src: 198 GC: 2098 DSA: 1970 VC: 2447]

[0.066 s 32121992 bytes] passed

com.pong.players.Paddle: getLeftCorner() ...

[prediction time: 0.0 s 32121992 bytes]

[Time: 0.017 s 32765000 bytes GC: 0.0020 s 32121992 bytes DSA: 0.0 s 32121992 bytes Ejp: 0.0010 s 32121992 bytes VC: 0.0 s 32121992 bytes Proof(s): 0.014 s 32765000 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.017 s 32765000 bytes] passed

com.pong.players.Paddle: getRightCorner() ...

[prediction time: 0.0 s 32765000 bytes]

[Time: 0.017 s 32765000 bytes GC: 0.0010 s 32765000 bytes DSA: 0.0010 s 32765000 bytes Ejp: 0.0010 s 32765000 bytes VC: 0.0 s 32765000 bytes Proof(s): 0.014 s 32765000 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.017 s 32765000 bytes] passed

com.pong.players.Paddle: getTopSize() ...

[prediction time: 0.0010 s 32765000 bytes]

[Time: 0.023 s 33408016 bytes GC: 0.0020 s 32765000 bytes DSA: 0.0010 s 33408016 bytes Ejp: 0.0 s 33408016 bytes VC: 0.0 s 33408016 bytes Proof(s): 0.02 s 33408016 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.024 s 33408016 bytes] passed

com.pong.players.Paddle: getBottomSize() ...

[prediction time: 0.0010 s 33408016 bytes]

[Time: 0.022 s 34051248 bytes GC: 0.0030 s 33408016 bytes DSA: 0.0 s 33408016 bytes Ejp: 0.0010 s 33408016 bytes VC: 0.0 s 33408016 bytes Proof(s): 0.018 s 34051248 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.022 s 34051248 bytes] passed

com.pong.players.Paddle: setDirection(int) ...

[prediction time: 0.0 s 34051248 bytes]

[Time: 0.025 s 34694232 bytes GC: 0.0030 s 34051248 bytes DSA: 0.0020 s 34051248 bytes Ejp: 0.0 s 34051248 bytes VC: 0.0 s 34051248 bytes Proof(s): 0.02 s 34694232 bytes]

[Size: src: 95 GC: 451 DSA: 388 VC: 889]

[0.025 s 34694232 bytes] passed

com.pong.players.Paddle: getPosX() ...

[prediction time: 0.0 s 34694232 bytes]

[Time: 0.018 s 34694232 bytes GC: 0.0020 s 34694232 bytes DSA: 0.0 s 34694232 bytes Ejp: 0.0010 s 34694232 bytes VC: 0.0 s 34694232 bytes Proof(s): 0.015 s 34694232 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.018 s 34694232 bytes] passed

com.pong.players.Paddle: getPosY() ...

[prediction time: 0.0 s 34694232 bytes]

[Time: 0.024 s 35337224 bytes GC: 0.0040 s 34694232 bytes DSA: 0.0010 s 34694232 bytes Ejp: 0.0010 s 35337224 bytes VC: 0.0 s 35337224 bytes Proof(s): 0.018 s 35337224 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 849]

[0.024 s 35337224 bytes] passed

com.pong.players.Paddle: setPaddlePosY(int) ...

[prediction time: 0.0 s 35337224 bytes]

[Time: 0.019 s 35980288 bytes GC: 0.0030 s 35337224 bytes DSA: 0.0010 s 35337224 bytes Ejp: 0.0 s 35337224 bytes VC: 0.0 s 35337224 bytes Proof(s): 0.015 s 35980288 bytes]

[Size: src: 78 GC: 745 DSA: 681 VC: 1177]

[0.019 s 35980288 bytes] passed

com.pong.players.Paddle: getDirection() ...

[prediction time: 0.0 s 35980288 bytes]

[Time: 0.0080 s 36623328 bytes GC: 0.0020 s 35980288 bytes DSA: 0.0 s 35980288 bytes Ejp: 0.0 s 35980288 bytes VC: 0.0 s 35980288 bytes Proof(s): 0.0060 s 36623328 bytes]

[Size: src: 65 GC: 371 DSA: 352 VC: 849]

[0.0080 s 36623328 bytes] passed

com.pong.players.Paddle: getSpeed() ...

[prediction time: 0.0 s 36623328 bytes]

[Time: 0.0080 s 36623328 bytes GC: 0.0010 s 36623328 bytes DSA: 0.0010 s 36623328 bytes Ejp: 0.0 s 36623328 bytes VC: 0.0 s 36623328 bytes Proof(s): 0.0060 s 36623328 bytes]

[Size: src: 65 GC: 371 DSA: 352 VC: 849]

[0.0080 s 36623328 bytes] passed

[0.944 s 36623328 bytes total]

com.pong.players.Ball ...

[0.027 s 37909328 bytes]

com.pong.players.Ball: Ball(int, int, int, int, int, int, int) ...

[prediction time: 0.0070 s 37909328 bytes]

[Time: 0.182 s 42410440 bytes GC: 0.032 s 38552320 bytes DSA: 0.02 s 39195360 bytes Ejp: 0.0020 s 39838352 bytes VC: 0.0 s 39838352 bytes Proof(s): 0.128 s 42410440 bytes]

[Size: src: 1143 GC: 3657 DSA: 6390 VC: 7960]

[0.184 s 42410440 bytes] passed

com.pong.players.Ball: move() ...

[prediction time: 0.013 s 43053464 bytes]

[Time: 0.274 s 49483512 bytes GC: 0.036 s 43696456 bytes DSA: 0.0070 s 44339440 bytes Ejp: 0.0050 s 44339440 bytes VC: 0.0 s 44339440 bytes Proof(s): 0.226 s 49483512 bytes]

[Size: src: 138 GC: 1413 DSA: 5752 VC: 7823]

[0.275 s 49483512 bytes] passed

com.pong.players.Ball: calcCorners(int, int) ...

[prediction time: 0.0 s 49483512 bytes]

[Time: 0.043 s 50769544 bytes GC: 0.0070 s 49483512 bytes DSA: 0.0020 s 49483512 bytes Ejp: 0.0010 s 50126536 bytes VC: 0.0 s 50126536 bytes Proof(s): 0.033 s 50769544 bytes]

[Size: src: 444 GC: 2497 DSA: 2369 VC: 2882]

[0.043 s 50769544 bytes] passed

com.pong.players.Ball: changeHorzDirection() ...

[prediction time: 0.0010 s 50769544 bytes]

[Time: 0.076 s 52698568 bytes GC: 0.0070 s 51412536 bytes DSA: 0.0030 s 51412536 bytes Ejp: 0.0010 s 51412536 bytes VC: 0.0 s 51412536 bytes Proof(s): 0.065 s 52698568 bytes]

[Size: src: 406 GC: 3174 DSA: 3432 VC: 3940]

[0.077 s 52698568 bytes] passed

com.pong.players.Ball: changeVertDirection() ...

[prediction time: 0.0010 s 52698568 bytes]

[Time: 0.071 s 54627624 bytes GC: 0.0060 s 53341608 bytes DSA: 0.0030 s 53341608 bytes Ejp: 0.0010 s 53341608 bytes VC: 0.0 s 53341608 bytes Proof(s): 0.061 s 54627624 bytes]

[Size: src: 406 GC: 3174 DSA: 3432 VC: 3940]

[0.072 s 54627624 bytes] passed

com.pong.players.Ball: setSpeed(int) ...

[prediction time: 0.0010 s 54627624 bytes]

[Time: 0.03 s 55913648 bytes GC: 0.0030 s 54627624 bytes DSA: 0.0010 s 55270640 bytes Ejp: 0.0010 s 55270640 bytes VC: 0.0 s 55270640 bytes Proof(s): 0.025 s 55913648 bytes]

[Size: src: 155 GC: 654 DSA: 640 VC: 1411]

[0.03 s 55913648 bytes] passed

com.pong.players.Ball: jumpTo(int, int) ...

[prediction time: 0.0 s 55913648 bytes]

[Time: 0.048 s 57199648 bytes GC: 0.0030 s 55913648 bytes DSA: 0.0020 s 55913648 bytes Ejp: 0.0010 s 55913648 bytes VC: 0.0 s 55913648 bytes Proof(s): 0.042 s 57199648 bytes]

[Size: src: 372 GC: 2057 DSA: 1951 VC: 2475]

[0.048 s 57199648 bytes] passed

com.pong.players.Ball: getPosX() ...

[prediction time: 0.0 s 57199648 bytes]

[Time: 0.013 s 57842664 bytes GC: 0.0020 s 57199648 bytes DSA: 0.0 s 57199648 bytes Ejp: 0.0010 s 57199648 bytes VC: 0.0 s 57199648 bytes Proof(s): 0.01 s 57842664 bytes]

[Size: src: 66 GC: 526 DSA: 507 VC: 1044]

[0.014 s 57842664 bytes] passed

com.pong.players.Ball: getPosY() ...

[prediction time: 0.0 s 57842664 bytes]

[Time: 0.012 s 57842664 bytes GC: 0.0010 s 57842664 bytes DSA: 0.0010 s 57842664 bytes Ejp: 0.0 s 57842664 bytes VC: 0.0 s 57842664 bytes Proof(s): 0.01 s 57842664 bytes]

[Size: src: 66 GC: 526 DSA: 507 VC: 1044]

[0.012 s 57842664 bytes] passed

com.pong.players.Ball: getLeftCorner() ...

[prediction time: 0.0010 s 57842664 bytes]

[Time: 0.024 s 58485688 bytes GC: 0.0010 s 58485688 bytes DSA: 0.0 s 58485688 bytes Ejp: 0.0 s 58485688 bytes VC: 0.0 s 58485688 bytes Proof(s): 0.022 s 58485688 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 891]

[0.024 s 58485688 bytes] passed

com.pong.players.Ball: getRightCorner() ...

[prediction time: 0.0 s 58485688 bytes]

[Time: 0.011 s 59128696 bytes GC: 0.0010 s 58485688 bytes DSA: 0.0010 s 58485688 bytes Ejp: 0.0 s 58485688 bytes VC: 0.0 s 58485688 bytes Proof(s): 0.0090 s 59128696 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 891]

[0.011 s 59128696 bytes] passed

com.pong.players.Ball: getTopSize() ...

[prediction time: 0.0010 s 59128696 bytes]

[Time: 0.011 s 59349304 bytes GC: 0.0010 s 59128696 bytes DSA: 0.0 s 59128696 bytes Ejp: 0.0 s 59128696 bytes VC: 0.0 s 59128696 bytes Proof(s): 0.0090 s 59349304 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 891]

[0.011 s 59349304 bytes] passed

com.pong.players.Ball: getBottomSize() ...

[prediction time: 0.0 s 59349304 bytes]

[Time: 0.021 s 29174736 bytes GC: 0.0010 s 59349304 bytes DSA: 0.0 s 59349304 bytes Ejp:

0.011 s 29174736 bytes VC: 0.0 s 29174736 bytes Proof(s): 0.0090 s 29174736 bytes]

[Size: src: 66 GC: 371 DSA: 352 VC: 891]

[0.021 s 29174736 bytes] passed

com.pong.players.Ball: getDeltaX() ...

[prediction time: 0.0 s 29174736 bytes]

[Time: 0.023 s 29174736 bytes GC: 0.0010 s 29174736 bytes DSA: 0.0010 s 29174736 bytes Ejp: 0.0 s 29174736 bytes VC: 0.0 s 29174736 bytes Proof(s): 0.021 s 29174736 bytes]

[Size: src: 66 GC: 373 DSA: 354 VC: 893]

[0.023 s 29174736 bytes] passed

com.pong.players.Ball: getDeltaY() ...

[prediction time: 0.0 s 29174736 bytes]

[Time: 0.01 s 30469696 bytes GC: 0.0010 s 29174736 bytes DSA: 0.0010 s 29174736 bytes Ejp: 0.0 s 29174736 bytes VC: 0.0 s 29174736 bytes Proof(s): 0.0080 s 30469696 bytes]

[Size: src: 66 GC: 373 DSA: 354 VC: 893]

[0.011 s 30469696 bytes] passed

com.pong.players.Ball: getGroundXSize() ...

[prediction time: 0.0 s 30469696 bytes]

[Time: 0.011 s 30469696 bytes GC: 0.0010 s 30469696 bytes DSA: 0.0 s 30469696 bytes Ejp: 0.0010 s 30469696 bytes VC: 0.0 s 30469696 bytes Proof(s): 0.0090 s 30469696 bytes]

[Size: src: 66 GC: 526 DSA: 507 VC: 1044]

[0.011 s 30469696 bytes] passed

com.pong.players.Ball: getGroundYSize() ...

[prediction time: 0.0 s 30469696 bytes]

[Time: 0.011 s 30469696 bytes GC: 0.0010 s 30469696 bytes DSA: 0.0 s 30469696 bytes Ejp: 0.0010 s 30469696 bytes VC: 0.0 s 30469696 bytes Proof(s): 0.0090 s 30469696 bytes]

[Size: src: 66 GC: 526 DSA: 507 VC: 1044]

[0.012 s 30469696 bytes] passed

[0.906 s 30469696 bytes total]

com.pong.observer.Observer ...

[0.026 s 31764864 bytes]

com.pong.observer.Observer: Observer(int, int, com.pong.players.Paddle, com.pong.players.Paddle, com.pong.players.Ball, int) ...

[prediction time: 0.0010 s 31764864 bytes]

[Time: 0.047 s 34354888 bytes GC: 0.0040 s 31764864 bytes DSA: 0.0020 s 33059920 bytes Ejp: 0.0010 s 33059920 bytes VC: 0.0 s 33059920 bytes Proof(s): 0.039 s 34354888 bytes]

[Size: src: 444 GC: 1511 DSA: 1467 VC: 3407]

[0.047 s 34354888 bytes] passed

com.pong.observer.Observer: observe() ...

[prediction time: 0.016 s 34354888 bytes]

[Time: 0.603 s 53779464 bytes GC: 0.035 s 35649888 bytes DSA: 0.017 s 38239872 bytes Ejp: 0.0050 s 39534816 bytes VC: 0.0 s 39534816 bytes Proof(s): 0.546 s 53779464 bytes]

[Size: src: 495 GC: 1524 DSA: 13634 VC: 26594]

[0.608 s 53779464 bytes] passed

com.pong.observer.Observer: checkForAWallHit() ...

[prediction time: 0.013 s 55074416 bytes]

[Time: 0.19 s 64139152 bytes GC: 0.02 s 56369352 bytes DSA: 0.0070 s 56369352 bytes Ejp: 0.0020 s 57664296 bytes VC: 0.0 s 57664296 bytes Proof(s): 0.161 s 64139152 bytes]

[Size: src: 115 GC: 849 DSA: 10601 VC: 16961]

[0.191 s 64139152 bytes] passed

com.pong.observer.Observer: checkForAPoint() ...

[prediction time: 0.0090 s 64139152 bytes]

[Time: 0.242 s 71908888 bytes GC: 0.02 s 64139152 bytes DSA: 0.0050 s 65434104 bytes Ejp: 0.0010 s 65434104 bytes VC: 0.0 s 65434104 bytes Proof(s): 0.216 s 71908888 bytes]

[Size: src: 115 GC: 1087 DSA: 8734 VC: 12938]

[0.242 s 71908888 bytes] passed

com.pong.observer.Observer: checkBallLevelEqualPaddleOneLevel() ...

[prediction time: 0.0010 s 71908888 bytes]

[Time: 0.053 s 74498864 bytes GC: 0.0040 s 71908888 bytes DSA: 0.0020 s 73203896 bytes Ejp: 0.0010 s 73203896 bytes VC: 0.0 s 73203896 bytes Proof(s): 0.046 s 74498864 bytes]

[Size: src: 63 GC: 461 DSA: 1917 VC: 4126]

[0.053 s 74498864 bytes] passed

com.pong.observer.Observer: paddleOneHit() ...

[prediction time: 0.01 s 75793800 bytes]

[Time: 0.844 s 31998464 bytes GC: 0.026 s 77088760 bytes DSA: 0.011 s 79678704 bytes Ejp: 0.0030 s 79678704 bytes VC: 0.0 s 79678704 bytes Proof(s): 0.803 s 31998464 bytes]

[Size: src: 217 GC: 1503 DSA: 17749 VC: 30175]

[0.847 s 31998464 bytes] passed

com.pong.observer.Observer: checkBallLevelEqualPaddleTwoLevel() ...

[prediction time: 0.0040 s 31998464 bytes]

[Time: 0.07 s 34606496 bytes GC: 0.0080 s 31998464 bytes DSA: 0.0020 s 31998464 bytes Ejp: 0.0 s 31998464 bytes VC: 0.0 s 31998464 bytes Proof(s): 0.06 s 34606496 bytes]

[Size: src: 63 GC: 461 DSA: 1917 VC: 4126]

[0.07 s 34606496 bytes] passed

com.pong.observer.Observer: paddleTwoHit() ...

[prediction time: 0.012 s 35910536 bytes]

[Time: 3.401 s 54167832 bytes GC: 0.024 s 37214576 bytes DSA: 0.0070 s 38518616 bytes Ejp: 0.0030 s 39823632 bytes VC: 0.0 s 39823632 bytes Proof(s): 3.367 s 54167832 bytes]

[Size: src: 217 GC: 1503 DSA: 17749 VC: 30175]

[3.404 s 54167832 bytes] passed

com.pong.observer.Observer: hitPaddleOne() ...

[prediction time: 0.0070 s 54167832 bytes]

[Time: 0.263 s 63296200 bytes GC: 0.016 s 55471912 bytes DSA: 0.0040 s 56775944 bytes Ejp: 0.0020 s 56775944 bytes VC: 0.0 s 56775944 bytes Proof(s): 0.241 s 63296200 bytes]

[Size: src: 1205 GC: 10448 DSA: 17281 VC: 20364]

[0.264 s 63296200 bytes] passed

com.pong.observer.Observer: hitPaddleTwo() ...

[prediction time: 0.0030 s 63296200 bytes]

[Time: 0.231 s 72424512 bytes GC: 0.013 s 64600216 bytes DSA: 0.0040 s 65904232 bytes Ejp: 0.0020 s 65904232 bytes VC: 0.0 s 65904232 bytes Proof(s): 0.212 s 72424512 bytes]

[Size: src: 1205 GC: 10448 DSA: 17281 VC: 20364]

[0.233 s 72424512 bytes] passed

com.pong.observer.Observer: hitWalls() ...

[prediction time: 0.0020 s 72424512 bytes]

[Time: 0.184 s 80248912 bytes GC: 0.013 s 73728744 bytes DSA: 0.0030 s 75032816 bytes Ejp:

0.0020 s 75032816 bytes VC: 0.0 s 75032816 bytes Proof(s): 0.166 s 80248912 bytes]

[Size: src: 944 GC: 6894 DSA: 13875 VC: 20055]

[0.186 s 80248912 bytes] passed

com.pong.observer.Observer: restartBall() ...

[prediction time: 0.0030 s 81552952 bytes]

[Time: 0.151 s 86769064 bytes GC: 0.013 s 81552952 bytes DSA: 0.0030 s 81552952 bytes Ejp: 0.0010 s 82856960 bytes VC: 0.0 s 82856960 bytes Proof(s): 0.134 s 86769064 bytes]

[Size: src: 1140 GC: 5887 DSA: 8046 VC: 10122]

[0.152 s 86769064 bytes] passed

[6.326 s 86769064 bytes total]

3 cautions

Tests

BallTest class

```
/**
 * The Class BallTest.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class BallTest extends TestCase {
    //~ Static fields/initializers -----------------------------------------------

    /** The Constant INITIALX. */
    private static final int INITIALX = 10;

    /** The Constant INITIALY. */
    private static final int INITIALY = 0;

    /** The Constant BALLSIZE. */
    private static final int BALLSIZE = 10;

    /** The Constant BALLSPEED. */
    private static final int BALLSPEED = 3;

    /** The Constant TABLEWIDTH. */
    private static final int TABLEWIDTH = 500;

    /** The Constant TABLEHEIGTH. */
    private static final int TABLEHEIGHT = 500;

    /** The Constant DELTA. */
    private static final int DELTA = -3;

    /** The Constant MOVECORECTION. */
    private static final int MOVECORECTION = 3;
```

```
//~ Instance fields ----------------------------------------------------------

/** The ball under test. */
private transient Ball ball;

//~ Methods ------------------------------------------------------------------

/**
 * The setUp - execute before each method.
 */
@Before
@Override
protected final void setUp() {
    this.ball = new Ball(INITIALX,
                         INITIALY,
                         BALLSIZE,
                         BALLSIZE,
                         BALLSPEED,
                         TABLEWIDTH,
                         TABLEHEIGHT);
}


/**
 * Test change horizontal direction.
 */
@Test
public final void testChangeHorzDirection() {
    this.ball.changeHorzDirection();
    assertEquals("DeltaX not as expected", DELTA, ball.getDeltaX());
}


/**
 * Test change vertical direction.
 */
@Test
public final void testChangeVertDirection() {
    this.ball.changeVertDirection();
    assertEquals("DeltaY not as expected", DELTA, ball.getDeltaY());
}


/**
 * Test ball's bottom position.
 */
@Test
public final void testGetBottomSize() {
    assertEquals("Bottom size not as expected",
                 BALLSIZE,
                 ball.getBottomSize());
}


/**
 * Test ball's left corner.
 */
@Test
public final void testLeftCorner() {
```

```
    assertEquals("Left corner not as expected",
                 BALLSIZE,
                 ball.getLeftCorner());
}


/**
 * Test ball's right corner.
 */
@Test
public final void testRightCorner() {
    assertEquals("Rigth corner not as expected",
                 2
                     * BALLSIZE,
                 ball.getRightCorner());
}


/**
 * Test ball's top position.
 */
@Test
public final void testTopSize() {
    assertEquals("Top position not as expected", 0, ball.getTopSize());
}


/**
 * Test ball X coordinate.
 */
@Test
public final void testPosX() {
    assertEquals("Position X not as expected", BALLSIZE, ball.getPosX());
}


/**
 * Test ball's Y coordinate.
 */
@Test
public final void testPosY() {
    assertEquals("Position Y not as expected", 0, ball.getPosY());
}


/**
 * Test ball move.
 */
@Test
public final void testMove() {
    this.ball.move();
    assertEquals("Position X not as expected",
                 BALLSIZE
                 + MOVECORECTION,
                 this.ball.getPosX());
    assertEquals("Position Y not as expected",
                 MOVECORECTION,
                 this.ball.getPosY());
    assertEquals("Left corner not as expected",
                 BALLSIZE
```

```
                    + MOVECORECTION,
                    this.ball.getLeftCorner());
        assertEquals("Right corner not as expected",
                    2 * BALLSIZE
                    + MOVECORECTION,
                    this.ball.getRightCorner());
        assertEquals("Top position not as expected",
                    MOVECORECTION,
                    this.ball.getTopSize());
        assertEquals("Bottom position not as expected",
                    BALLSIZE
                    + MOVECORECTION,
                    this.ball.getBottomSize());
    }


    /**
     * Test ball's jump to.
     */
    @Test
    public final void testJumpTo() {
        this.ball.jumpTo(TABLEWIDTH / 2, TABLEHEIGHT / 2);
        assertEquals("Position Y not as expected",
                    TABLEHEIGHT
                        / 2,
                    ball.getPosY());
        assertEquals("Position X not as expected",
                    TABLEWIDTH
                        / 2,
                    ball.getPosX());
    }


    /**
     * Test get ground X size.
     */
    @Test
    public final void testGetGroundXSize() {
        assertEquals("Ground X size not as expected",
                    TABLEWIDTH,
                    ball.getGroundXSize());
    }


    /**
     * Test get ground Y size.
     */
    @Test
    public final void testGetGroundYSize() {
        assertEquals("Ground X size not as expected",
                    TABLEHEIGHT,
                    ball.getGroundYSize());
    }
}
```

PaddleTest class

```
    //~ Static fields/initializers -----------------------------------------
```

```java
/** The Constant PADDLEWIDTH. */
private static final int PADDLEWIDTH = 10;

/** The Constant PADDLEHEIGHT. */
private static final int PADDLEHEIGHT = 30;

/** The Constant INITIALX. */
private static final int INITIALX = 10;

/** The Constant INITIALY. */
private static final int INITIALY = 0;

/** The Constant TABLEHEIGTH. */
private static final int TABLEHEIGHT = 500;

/** The Constant PADDLESPEED. */
private static final int PADDLESPEED = 2;

/** The Constant PRIGTHCORNER. */
private static final int PRIGTHCORNER = 20;

/** The Constant PLEFTCORNER. */
private static final int PLEFTCORNER = 10;

/** The Constant EXPBOTTOMSIZE. */
private static final int EXPBOTTOMSIZE = 30;

/** The Constant PSPEED. */
private static final int PSPEED = 2;

/** The Constant DIRECTION. */
private static final int DIRECTION = 1;

/** The Constant PPOSY. */
private static final int PPOSY = 250;

/** The Constant PPOSX. */
private static final int PPOSX = 250;

//~ Instance fields ----------------------------------------------------------

/** The paddle under test. */
private transient Paddle paddle;

//~ Methods ------------------------------------------------------------------

/**
 * Set up.
 */
@Before
@Override
public final void setUp() {
    this.paddle = new Paddle(INITIALX,
                             INITIALY,
                             PADDLEWIDTH,
                             PADDLEHEIGHT,
                             TABLEHEIGHT,
                             PADDLESPEED);
}
```

```java
/**
 * Test get bottom position.
 */
@Test
public final void testGetBottomSize() {
    assertEquals("Paddle bottom size not as expected",
                 EXPBOTTOMSIZE,
                 this.paddle.getBottomSize());
}


/**
 * Test get speed.
 */
@Test
public final void testGetSpeed() {
    assertEquals("Paddle speed not as expected",
                 PSPEED,
                 this.paddle.getSpeed());
}


/**
 * Test get direction.
 */
@Test
public final void testDirection() {
    this.paddle.setDirection(DIRECTION);
    assertEquals("Paddle direction not as expected",
                 2
                     * DIRECTION,
                 this.paddle.getDirection());
}


/**
 * Test get left corner.
 */
@Test
public final void testGetLeftCorner() {
    assertEquals("Paddle left corner not as expected",
                 PLEFTCORNER,
                 this.paddle.getLeftCorner());
}


/**
 * Test get right corner.
 */
@Test
public final void testGetRightCorner() {
    assertEquals("Paddle right corner not as expected",
                 PRIGTHCORNER,
                 this.paddle.getRightCorner());
}


/**
 * Test get X coordinate.
```

```
     */
    @Test
    public final void testGetPosX() {
        assertEquals("Paddle position X not as expected",
                     INITIALX,
                     this.paddle.getPosX());
    }


    /**
     * Test get Y coordinate.
     */
    @Test
    public final void testGetPosY() {
        assertEquals("Paddle position Y not as expected",
                     INITIALY,
                     this.paddle.getPosY());
    }


    /**
     * Test set Y coordinate.
     */
    @Test
    public final void testSetPosY() {
        this.paddle.setPaddlePosY(PPOSY);
        assertEquals("Paddle position Y not as expected",
                     PPOSY,
                     this.paddle.getPosY());
    }


    /**
     * Test move.
     */
    @Test
    public final void testCalculateCorners() {
        this.paddle.setPaddlePosY(PPOSY);
        this.paddle.calcCorners(PPOSX, PPOSY);


        assertEquals("Paddle calcCorners - left corner not as expected",
                     PPOSX,
                     this.paddle.getLeftCorner());

        assertEquals("Paddle calcCorners - right corner not as expected",
                     PPOSX
                     + PADDLEWIDTH,
                     this.paddle.getRightCorner());


        assertEquals("Paddle calcCorners - top position not as expected",
                     PPOSX,
                     this.paddle.getTopSize());

        assertEquals("Paddle calcCorners - bottom position not as expected",
                     PPOSX
                     + PADDLEHEIGHT,
                     this.paddle.getBottomSize());
    }
```

```
    /**
     * Test get paddle top position.
     */
    @Test
    public final void testGetPaddleTopSize() {
        assertEquals("Paddle position Y not as expected",
                     INITIALY,
                     this.paddle.getTopSize());
    }
}
```

ObserverTest class

```
/**
 * The Class ObserverTest.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class ObserverTest extends TestCase {
    //~ Static fields/initializers ---------------------------------------------

    /** The Constant TABLEWIDTH. */
    private static final int TABLEWIDTH = 500;

    /** The Constant TABLEHEIGTH. */
    private static final int TABLEHEIGHT = 500;

    /** The Constant PADDLEWIDTH. */
    private static final int PADDLEWIDTH = 10;

    /** The Constant PADDLEHEIGHT. */
    private static final int PADDLEHEIGHT = 30;

    /** The Constant INITIALX. */
    private static final int INITIALX = 10;

    /** The Constant INITIALY. */
    private static final int INITIALY = 0;

    /** The Constant BALLSIZE. */
    private static final int BALLSIZE = 10;

    /** The Constant BALLSPEED. */
    private static final int BALLSPEED = 3;

    /** The Constant PADDLESPEED. */
    private static final int PADDLESPEED = 2;

    /** The Constant DELETAYMESSAGE. */
    private static final String DELETAYMESSAGE = "DeltaY not as expected";

    /** The Constant DELETAXMESSAGE. */
    private static final String DELETAXMESSAGE = "DeltaX not as expected";

    /** The Constant MOVECORRECTION. */
    private static final int MOVECORRECTION = 3;
```

```
//~ Instance fields ----------------------------------------------------------

/** The observer under test. */
private transient Observer observer;

/** The ball. */
private transient Ball ball;

/** The paddle one. */
private transient Paddle paddleOne;

/** The paddle two. */
private transient Paddle paddleTwo;

//~ Methods ------------------------------------------------------------------

/**
 * Sets the up.
 *
 */
@Before
@Override
public final void setUp()  {
    this.paddleOne = new Paddle(INITIALX,
                                INITIALY,
                                PADDLEWIDTH,
                                PADDLEHEIGHT,
                                TABLEHEIGHT,
                                PADDLESPEED);

    this.paddleTwo = new Paddle(TABLEWIDTH - (2 * INITIALX),
                                INITIALY,
                                PADDLEWIDTH,
                                PADDLEHEIGHT,
                                TABLEHEIGHT,
                                PADDLESPEED);

    this.ball = new Ball(INITIALX,
                         INITIALY,
                         BALLSIZE,
                         BALLSIZE,
                         BALLSPEED,
                         TABLEWIDTH,
                         TABLEHEIGHT);

    this.observer = new Observer(TABLEHEIGHT,
                                 TABLEWIDTH,
                                 paddleOne,
                                 paddleTwo,
                                 ball,
                                 PADDLEHEIGHT);
}


/**
 * Test start play.
 *
 * @throws  Throwable  if the PrivateAccessor cannot find the restartBall
 *                     method
```

```
     */
    @Test
    public final void testStartPlay() throws Throwable {
        PrivateAccessor.invoke(this.observer, "restartBall", null, null);

        assertEquals("Position X not as expected",
                     INITIALX, this.ball.getPosX());

        assertEquals("Position Y not as expected",
                     INITIALY, this.ball.getPosY());
    }


    /**
     * Test make a point.
     *
     */
    @Test
    public final void testMakeAPoint() {
        final int ballXCoord = -6;
        this.ball.jumpTo(ballXCoord, BALLSIZE);

        this.observer.observe();

        assertEquals("Position X not as expected",
                     TABLEWIDTH / 2, this.ball.getPosX());

        assertEquals("Position Y not as expected",
                     TABLEHEIGHT / 2 + MOVECORRECTION, this.ball.getPosY());
    }


    /**
     * Test make a point.
     *
     */
    @Test
    public final void testMakeAPointOne() {
        this.ball.jumpTo(TABLEWIDTH + MOVECORRECTION, BALLSIZE);

        this.observer.observe();

        assertEquals("Position X not as expected",
                     TABLEWIDTH / 2, this.ball.getPosX());

        assertEquals("Position Y not as expected",
                     TABLEHEIGHT / 2 + MOVECORRECTION, this.ball.getPosY());
    }


    /**
     * Test make a point - change vertical direction.
     *
     * @throws  Throwable  if the PrivateAccessor cannot find the hitWalls
     *                     method
     */
    @Test
    public final void testHitTopWallVerical() throws Throwable {
        final int ballXCoord = 20;
        final int ballYCoord = -4;
```

```
        this.ball.jumpTo(ballXCoord, ballYCoord);
        this.ball.move();

        final int oldDeltaY = 3;

        assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());
        final int newDeltaY = -3;

        PrivateAccessor.invoke(this.observer, "hitWalls", null, null);

        assertEquals(DELETAYMESSAGE, newDeltaY, this.ball.getDeltaY());
    }


    /**
     * Test hit a wall - change vertical direction of the ball.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the hitWalls
     *                     method
     */
    @Test
    public final void testHitBottomWallVerical() throws Throwable {
        final int ballXCoord = 487;
        final int ballYCoord = 497;

        this.ball.jumpTo(ballXCoord, ballYCoord);
        this.ball.move();

        final int oldDeltaY = 3;

        assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());
        final int newDeltaY = -3;

        PrivateAccessor.invoke(this.observer, "hitWalls", null, null);

        assertEquals("DeltaY not as expected",
                     newDeltaY, this.ball.getDeltaY());
    }


    /**
     * Test make a point - change horizontal direction.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the hitWalls
     *                     method
     */
    @Test
    public final void testHitWallHorizontal() throws Throwable {
        final int ballXCoord = 30;
        final int ballYCoord = 503;

        this.ball.jumpTo(ballXCoord, ballYCoord);
        this.ball.move();

        final int oldDeltaY = 3;

        assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());

        final int newDeltaY = -3;
```

```
        PrivateAccessor.invoke(this.observer, "hitWalls", null, null);

        assertEquals(DELETAYMESSAGE, newDeltaY, this.ball.getDeltaY());
    }


    /**
     * Test hit paddle one - change vertical direction.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the
     *                     hitPaddleOne method
     */
    @Test
    public final void testHitPaddleOneVerticalChange() throws Throwable {
        final int paddleX = 10;
        final int paddleY = 30;

        this.paddleOne.setPaddlePosY(paddleY);
        this.paddleOne.calcCorners(paddleX, paddleY);

        final int ballPosX = 10;
        final int ballPosY = 50;

        this.ball.jumpTo(ballPosX, ballPosY);
        this.ball.move();

        final int oldDeltaY = 3;

        assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());

        PrivateAccessor.invoke(this.observer, "hitPaddleOne", null, null);

        final int newDeltaY = -3;

        assertEquals(DELETAYMESSAGE, newDeltaY, this.ball.getDeltaY());
    }


    /**
     * Test hit paddle one - change horizontal direction.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the
     *                     hitPaddleOne method
     */
    @Test
    public final void testHitPaddleOneHorizontalChange() throws Throwable {
        final int paddleX = 10;
        final int paddleY = 30;

        this.paddleOne.setPaddlePosY(paddleY);
        this.paddleOne.calcCorners(paddleX, paddleY);

        final int ballPosX = 15;
        final int ballPosY = 30;

        this.ball.jumpTo(ballPosX, ballPosY);
        this.ball.move();

        final int oldDeltaX = 3;
```

```
        assertEquals(DELETAXMESSAGE, oldDeltaX, this.ball.getDeltaX());

        PrivateAccessor.invoke(this.observer, "hitPaddleOne", null, null);

        final int newDeltaX = -3;

        assertEquals(DELETAXMESSAGE, newDeltaX, this.ball.getDeltaX());
    }


    /**
     * Test hit paddle two - change vertical direction.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the
     *                     hitPaddleTwo method
     */
    @Test
    public final void testHitPaddleTwoVerticalChange() throws Throwable {
        final int paddleX = 490;
        final int paddleY = 30;

        this.paddleTwo.setPaddlePosY(paddleY);
        this.paddleTwo.calcCorners(paddleX, paddleY);

        final int ballPosX = 490;
        final int ballPosY = 50;

        this.ball.jumpTo(ballPosX, ballPosY);
        this.ball.move();

        final int oldDeltaY = 3;

        assertEquals(DELETAYMESSAGE, oldDeltaY, this.ball.getDeltaY());

        PrivateAccessor.invoke(this.observer, "hitPaddleTwo", null, null);

        final int newDeltaY = -3;

        assertEquals(DELETAYMESSAGE, newDeltaY, this.ball.getDeltaY());
    }


    /**
     * Test hit paddle two - change horizontal direction.
     *
     * @throws  Throwable  the if the PrivateAccessor cannot find the
     *                     hitPaddleTwo method
     */
    @Test
    public final void testHitPaddleTwoHorizontalChange() throws Throwable {
        final int paddleX = 490;
        final int paddleY = 30;

        this.paddleTwo.setPaddlePosY(paddleY);
        this.paddleTwo.calcCorners(paddleX, paddleY);

        final int ballPosX = 479;
        final int ballPosY = 30;
```

```
        this.ball.jumpTo(ballPosX, ballPosY);
        this.ball.move();

        final int oldDeltaX = 3;

        assertEquals(DELETAXMESSAGE, oldDeltaX, this.ball.getDeltaX());

        PrivateAccessor.invoke(this.observer, "hitPaddleTwo", null, null);

        final int newDeltaX = -3;

        assertEquals(DELETAXMESSAGE, newDeltaX, this.ball.getDeltaX());
    }


    /**
     * Test observe.
     */
    @Test
    public final void testObserveBall() {
        this.observer.observe();

        final int newDeltaX = 3;

        assertEquals("DeltaX not as expected",
                     newDeltaX,
                     this.ball.getDeltaX());

        final int ballPosX = 13;

        final int ballPosY = 0;

        final int ballLeftCorner = 13;

        final int ballRightCorner = 23;

        final int ballTopSize = 3;

        final int ballBottomSize = 13;

        assertEquals("Ball X position not as expected",
                     ballPosX,
                     this.ball.getPosX());
        assertEquals("Ball Y position not as expected",
                     ballPosY,
                     this.ball.getPosY());
        assertEquals("Ball left corner position not as expected",
                     ballLeftCorner,
                     this.ball.getLeftCorner());
        assertEquals("Ball right corner position not as expected",
                     ballRightCorner,
                     this.ball.getRightCorner());
        assertEquals("Ball top size position not as expected",
                     ballTopSize,
                     this.ball.getTopSize());
        assertEquals("Ball bottom size position not as expected",
                     ballBottomSize,
                     this.ball.getBottomSize());
    }
```

```java
/**
 * Test observe.
 */
@Test
public final void testObservePaddleOne() {
    this.observer.observe();

    final int pOneDirection = 0;

    final int pOnePosX = 10;

    final int paddleOnePosY = 0;

    final int pOneLeftCorner = 10;

    final int pOneRightCorner = 20;

    final int pOneTopSize = 0;

    final int pOneBottomSize = 30;

    assertEquals("PaddleOne direction not as expected",
                 pOneDirection,
                 this.paddleOne.getDirection());

    assertEquals("PaddleOne position X not as expected",
                 pOnePosX,
                 this.paddleOne.getPosX());

    assertEquals("PaddleOne position Y not as expected",
                 paddleOnePosY,
                 this.paddleOne.getPosY());

    assertEquals("PaddleOne left corner not as expected",
                 pOneLeftCorner,
                 this.paddleOne.getLeftCorner());

    assertEquals("PaddleOne right corner not as expected",
                 pOneRightCorner,
                 this.paddleOne.getRightCorner());

    assertEquals("PaddleOne top size not as expected",
                 pOneTopSize,
                 this.paddleOne.getTopSize());

    assertEquals("PaddleOne bottom size not as expected",
                 pOneBottomSize,
                 this.paddleOne.getBottomSize());
}


/**
 * Test observe.
 */
@Test
public final void testObservePaddleTwo() {
    this.observer.observe();

    final int pTwoDirection = 0;
```

```
        final int pTwoPosX = 480;

        final int pTwoPosY = 0;

        final int pTwoLeftCorner = 480;

        final int pTwoRightCorner = 490;

        final int pTwoTopSize = 0;

        final int pTwoBottomSize = 30;

        assertEquals("PaddleTwo direction not as expected",
                     pTwoDirection,
                     this.paddleTwo.getDirection());

        assertEquals("PaddleTwo position X not as expected",
                     pTwoPosX,
                     this.paddleTwo.getPosX());

        assertEquals("PaddleTwo position Y not as expected",
                     pTwoPosY,
                     this.paddleTwo.getPosY());

        assertEquals("PaddleTwo left corner not as expected",
                     pTwoLeftCorner,
                     this.paddleTwo.getLeftCorner());

        assertEquals("PaddleOne right corner not as expected",
                     pTwoRightCorner,
                     this.paddleTwo.getRightCorner());

        assertEquals("PaddleTwo top size not as expected",
                     pTwoTopSize,
                     this.paddleTwo.getTopSize());

        assertEquals("PaddleTwo bottom size not as expected",
                     pTwoBottomSize,
                     this.paddleTwo.getBottomSize());
    }
}
```

ManagerTest class

```
/**
 * The Class ManagerTest.
 *
 * @author   Benjamin Irani and Nedyalko Kargov
 * @version  1.0.0
 */
public class ManagerTest extends TestCase {
    //~ Static fields/initializers -----------------------------------------

    /** The Constant PADDLEONE. */
    private static final String PADDLEONE = "paddleOne";

    /** The Constant PADDLETWO. */
    private static final String PADDLETWO = "paddleTwo";
```

```
/** The Constant DIRECTIONDOWN. */
private static final int DIRECTIONDOWN = 4;

/** The Constant DIRECTIONUP. */
private static final int DIRECTIONUP = -4;

/** The Constant WRONGKEY. */
private static final int WRONGKEY = 99;

/** The Constant PADDLEONEKEYDOWN. */
private static final int PADDLEONEKEYDOWN = 97;

/** The Constant PADDLEONEKEYUP. */
private static final int PADDLEONEKEYUP = 113;

/** The Constant WHITECOLOR. */
private static final int WHITECOLOR = 255;

/** The Constant PADDLEONETESTMESSAGE. */
private static final String PONETESTMESSAGE =
    "Paddle One direction not as expected ";

/** The Constant PADDLETWOTESTMESSAGE. */
private static final String PTWOTESTMESSAGE =
    "Paddle Two direction not as expected ";

/** The Constant XCOORDINATESMESSAGE. */
private static final String XCOORDMESSAGE = "Coordinates: X:";

/** The Constant YCOORDINATEMESSAGE. */
private static final String YCOORDMESSAGE = " Y:";

//~ Instance fields ------------------------------------------------------

/** The viewer. */
private transient AppletViewer viewer;

/** The applet. */
private transient FrameFixture applet;

/** The manager under test. */
private transient Manager manager;

//~ Methods --------------------------------------------------------------

/**
 * Sets the up.
 */
@Before
@Override
public final void setUp() {
    this.manager = new Manager();
    this.viewer = AppletLauncher.applet(manager).start();
    this.applet = new FrameFixture(viewer);
}


/**
 * Test applet object inisialization.
```

```
     *
     * @throws  Throwable  if the PrivateAccessor cannot get a
     *                     private field or method
     */
    @Test
    public final void testAppletObjectInisialization() throws Throwable {
        final Paddle paddleOne = (Paddle) PrivateAccessor.getField(this.manager,
                                                                   PADDLEONE);
        assertNotNull("PaddleOne object cannot be inicializated", paddleOne);

        final Paddle paddleTwo = (Paddle) PrivateAccessor.getField(this.manager,
                                                                   PADDLETWO);

        assertNotNull("PaddleTwo object cannot be inicializated", paddleTwo);

        final Ball ball = (Ball) PrivateAccessor.getField(this.manager, "ball");

        assertNotNull("Ball object cannot be inicializated", ball);

        final Observer observer = (Observer) PrivateAccessor.getField(
                                    this.manager, "observer");

        assertNotNull("Observer object cannot be inicializated", observer);
    }


    /**
     * Test graphics.
     *
     * @throws  Throwable  if the PrivateAccessor cannot get a
     *                     private field or method
     */
    @Test
    public final void testObjectsGraphics() throws Throwable {
        PrivateAccessor.setField(this.manager, "inRun", false);

        PrivateAccessor.invoke(this.manager, "tablePaint", null, null);

        final BufferedImage bufferedImage = (BufferedImage)
                                                PrivateAccessor.getField(
                                                    this.manager,
                                                    "bufferedImage");

        final int tableWidth = (Integer) PrivateAccessor.getField(this.manager,
                                                                  "TABLEWIDTH");

        final int tableHeight = (Integer) PrivateAccessor.getField(
                                    this.manager, "TABLEHEIGHT");

        assertNotNull("Buffered Image not as expected", bufferedImage);

        for (int y = 0; y < tableHeight; y++) {
            for (int x = 0; x < tableWidth; x++) {
                tPaddleOne(bufferedImage, x, y);
                tPaddleTwo(bufferedImage, x, y);
                tBall(bufferedImage, x, y);
            }
        }
    }
```

```
/**
 * Test paddle one graphic.
 *
 * @param   bufferedImage  the buffered image
 * @param   xCoordinate    the x coordinate
 * @param   yCoordinate    the y coordinate
 *
 * @throws  NoSuchFieldException  if the PrivateAccessor cannot get the
 *                                private field paddleOne
 */
private void tPaddleOne(final BufferedImage bufferedImage,
                        final int xCoordinate,
                        final int yCoordinate) throws NoSuchFieldException {
    final Paddle paddleOne = (Paddle) PrivateAccessor.getField(this.manager,
                                                               PADDLEONE);

    if (xCoordinate >= paddleOne.getLeftCorner()
            && xCoordinate < paddleOne.getRightCorner()
            && yCoordinate >= paddleOne.getTopSize()
            && yCoordinate < paddleOne.getBottomSize()) {
        final Color color = new Color(bufferedImage.getRGB(xCoordinate,
                                                           yCoordinate));
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getBlue());
        assertEquals("Coordinates: X:" + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getRed());
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getGreen());
    }
}


/**
 * Test paddle two graphics.
 *
 * @param   bufferedImage  the buffered image
 * @param   xCoordinate    the x coordinate
 * @param   yCoordinate    the y coordinate
 *
 * @throws  NoSuchFieldException  if the PrivateAccessor cannot get the
 *                                private field paddleTwo
 */
private void tPaddleTwo(final BufferedImage bufferedImage,
                        final int xCoordinate,
                        final int yCoordinate) throws NoSuchFieldException {
    final Paddle paddleTwo = (Paddle) PrivateAccessor.getField(this.manager,
                                                               PADDLETWO);

    if (xCoordinate >= paddleTwo.getLeftCorner()
            && xCoordinate < paddleTwo.getRightCorner()
            && yCoordinate >= paddleTwo.getTopSize()
            && yCoordinate < paddleTwo.getBottomSize()) {
        final Color color = new Color(bufferedImage.getRGB(xCoordinate,
```

```
                                                                yCoordinate));
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getBlue());
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getRed());
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getGreen());
    }
}


/**
 * Test ball.
 *
 * @param   bufferedImage  the buffered image
 * @param   xCoordinate    the x coordinate
 * @param   yCoordinate    the y coordinate
 *
 * @throws  NoSuchFieldException  if the PrivateAccessor cannot get the
 *                                private field Ball
 */
private void tBall(final BufferedImage bufferedImage,
                   final int xCoordinate,
                   final int yCoordinate) throws NoSuchFieldException {
    final Ball ball = (Ball) PrivateAccessor.getField(this.manager, "ball");

    if (xCoordinate >= ball.getLeftCorner()
            && xCoordinate < ball.getRightCorner()
            && yCoordinate >= ball.getTopSize()
            && yCoordinate < ball.getBottomSize()) {
        final Color color = new Color(bufferedImage.getRGB(xCoordinate,
                                                           yCoordinate));
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getBlue());
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getRed());
        assertEquals(XCOORDMESSAGE + xCoordinate + YCOORDMESSAGE
                     + yCoordinate,
                     WHITECOLOR,
                     color.getGreen());
    }
}


/**
 * Test paddle one keys.
 *
 * @throws  AWTException          if cannot create the Robot
 * @throws  NoSuchFieldException  if the PrivateAccessor cannot get the
 *                                private field paddleOne
```

```
     */
    @Test
    public final void testPaddleOneKeys() throws AWTException,
                                                 NoSuchFieldException {
        final Paddle paddleOne = (Paddle) PrivateAccessor.getField(this.manager,
                                                                   PADDLEONE);

        assertEquals(PONETESTMESSAGE, 0, paddleOne.getDirection());

        final Robot robot = new Robot();

        this.manager.keyDown(new Event(robot, 0, robot), WRONGKEY);

        assertEquals(PONETESTMESSAGE, 0, paddleOne.getDirection());

        this.manager.keyDown(new Event(robot, 0, robot), PADDLEONEKEYDOWN);

        assertEquals(PONETESTMESSAGE, DIRECTIONDOWN, paddleOne.getDirection());

        this.manager.keyUp(new Event(robot, 0, robot),
                           Event.KEY_ACTION_RELEASE);

        assertEquals(PONETESTMESSAGE, 0, paddleOne.getDirection());

        this.manager.keyDown(new Event(robot, 0, robot), PADDLEONEKEYUP);

        assertEquals(PONETESTMESSAGE, DIRECTIONUP, paddleOne.getDirection());

        this.manager.keyDown(new Event(robot, 0, robot), WRONGKEY);

        assertEquals(PONETESTMESSAGE, DIRECTIONUP, paddleOne.getDirection());

        PrivateAccessor.setField(this.manager, "paddleOne", null);

        assertTrue("Wrong Key pressed not as expected ",
                   this.manager.keyDown(new Event(robot, 0, robot),
                                        PADDLEONEKEYUP));
    }


    /**
     * Test paddle two keys.
     *
     * @throws  AWTException          if cannot create the Robot
     * @throws  NoSuchFieldException  if the PrivateAccessor cannot get the
     *                                private field paddleTwo
     */
    @Test
    public final void testPaddleTwoKeys() throws AWTException,
                                                 NoSuchFieldException {
        final Paddle paddleTwo = (Paddle) PrivateAccessor.getField(this.manager,
                                                                   PADDLETWO);

        assertEquals("Paddle Two direction not as expected",
                     0,
                     paddleTwo.getDirection());

        final Robot robot = new Robot();

        this.manager.keyDown(new Event(robot, 0, robot), WRONGKEY);
```

```
        assertEquals(PTWOTESTMESSAGE, 0, paddleTwo.getDirection());

        this.manager.keyDown(new Event(robot, 0, robot), Event.DOWN);

        assertEquals(PTWOTESTMESSAGE, DIRECTIONDOWN, paddleTwo.getDirection());

        this.manager.keyUp(new Event(null, Event.KEY_ACTION_RELEASE, null),
                           Event.KEY_ACTION_RELEASE);

        assertEquals(PTWOTESTMESSAGE, 0, paddleTwo.getDirection());

        this.manager.keyDown(new Event(robot, 0, robot), Event.UP);

        assertEquals(PTWOTESTMESSAGE, DIRECTIONUP, paddleTwo.getDirection());
    }


    /**
     * Tear down.
     */
    @After
    @Override
    public final void tearDown() {
        this.viewer.unloadApplet();
        this.applet.cleanUp();
    }
```