Ex6 - Kaboom!

This exercise is based on a similar exercise written by David H. Hovemeyer.

Objective

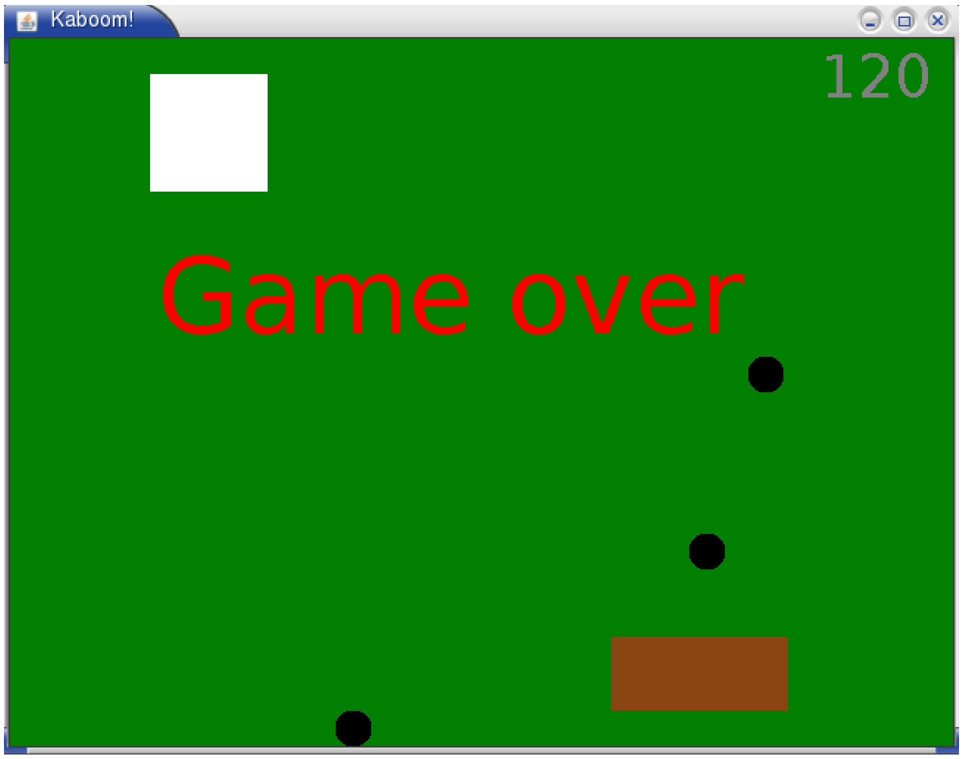
In this exercise you will practice communication between several classes. You will also get your first experience in test-driven development and in implementing a graphical user inteface (GUI).

Your Task

Your task is to implement a simple version of the classic video game Kaboom!.
The game consists of the following game play elements:

- A *bucket*, controlled by the player, is located at the bottom of the game play field. The user can move the bucket sideways by moving the mouse.
- A *bomber*, controlled by the computer, moves horizontally at the top of the game play field.
- *bombs* are dropped by the bomber and fall from the top of game play field toward the bottom. If the player catches a bomb in the bucket, the bomb is defused, and the player earns points. If a bomb reaches the bottom of the game play field without being caught, it explodes and the game is over.

Here is a screenshot (the bomber is white and the bucket is brown here):



It is recommended that you try out our implementation (see below) before starting to work on this exercise.

Part I - The Basic Elements

You are required to implement the classes Circle and Rectangle according to the supplied API. You may assume legal input to all methods and constructors in these and in all other classes in the exercise. Note that some methods in the APIs of these classes use the Point class you implemented in ex4. We recommend using the Point class also in the private implementation of the Rectangle and Circle classes. For these classes (Rectangle and Circle) you are required to write comprehensive unit-tests using the JUnit4 framework, as taught in the lecture and in the recitations. Instead of providing you with unit-tests for these classes, we provide you with several different implementations of these classes, each with a different bug (more details in the end of this document). If you write your unit-tests well, they should detect the bug in each buggy class implementation (that is, the faulty class implementations should fail one or more tests, while the correct school solution should pass all of them).
Important: We do not supply any automatic tests whatsoever for Circle and Rectangle. This means that if you have a bug that your testers do not catch, it will either cause weird bugs in the Game/Mediator classes later in this exercise (and they will be harder to debug in that context), or worse, the bugs will remain unnoticed, except for by our graders (!). For this reason, you should write tests that are as comprehensive as possible. In order to get the full manual grade for the task of writing the testers, we therefore indeed expect to see meaningful tests beyond the minimum needed to just spot our provided buggy implementations.
You are asked to follow the test-driven development process discussed in the lecture: Start by writing all the unit-tests and keep improving them until they fail all the faulty classes we supply (and pass the school solution); only then should you move on to writing your own implementation of the Rectangle or Circle classes. This way you can test your implementation during its development using the unit-tests you have already written.

Part II - The Game and GameParams classes

The **Game** class should contain all of the game data as fields, and its methods should implement the game play logic. You should implement it according to its API.
Important: Keep in mind that **Game** is an *abstract representation* of the game data and logic: it has nothing to do with the GUI (e.g. it knows nothing about colors) and no rendering/drawing should be done in the **Game** class.
You will need to think about what kind of data is required to represent the game state, and appropriately add private fields to store this data.

The **Game** class can (and should) use the classes you developed in the first part of the assignment according to their API.
Note that several bombs (arbitrarily many, depending on the game play field size) may be falling at any given time. The **Game** object will need to use an array or an **ArrayList** (or any other java Collection) in order to keep track of all of the bombs.
Among others, you will need to implement the step() method, which will be called by the **Mediator** class (see the next part) once for each frame of animation displayed. This method implements (either directly, or indirectly through private methods) the bulk of the game logic. Each call to **Game**'s **`step()`** method should update the game data:

- Move the bomber right or left (in a "random" manner) - the **GameParams** objects specifies the minimum and maximum timer ticks between direction changes. You should use a java.util.Random object to choose a random number of ticks for each direction change. For instance, if the minimum number of ticks is 10 and the maximum is 20 ticks a, possible movement of the bomber might be left (it always start in this direction) for 15 ticks, then right for 12 ticks, then left for 20 ticks and so on. (These should be randomized and therefore they will be different each time you run the game.) **A tip**: If you want, just for debugging purposes, to have the same "random" numbers returned over and over again each time you restart your program, you can use the constructor Random(long seed). Without going into too much detail, it is enough to know that if you consistently provide the same long at construction time (just use your favorite number...), you will get the same "random" behavior. Just don't forget to change back to the default (no-parameter) constructor before invoking our automatics testers / playing the game / submitting the code!
- Possibly cause the bomber to drop a bomb - when a new bomb is dropped, its starting center x coordinate is in the middle of the bomber and its starting y coordinate is according to the **GameParams** object.
- Move all bombs down towards the bottom of the game play field (the step size in each tick is specified in the **GameParams** object).
- Check to see if the bucket has just caught one or more bombs - a bomb is caught if its center is inside the bucket.
- Check to see if any bomb has reached the bottom of the game play field (i.e. if its bottom has reached the bottom of the game play field), in which case the game is over.

An instance of the **GameParams** class holds all the parameters needed for the game, such as the size of the board, the sizes of the object (bomber, bucket, and bombs), the time range for bucket's direction change and so on; we wrote this class for you - grab it from here and familiarize yourself with the code. A **GameParams** object is passed to the **Game** class constructor when a new instance of **Game** is created. You should study the **GameParams** API carefully and make sure your **Game** respects all the parameters given to it (and does not make up any arbitrarily values for these).

Part III - The Mediator class

The **Mediator** class is a class whose purpose is to mediate between the **Game** class and the provided GUI (**KaboomView**). It is responsible for updating the game and for drawing it on the screen. We provide you with its API. Notice that you have some freedom in the way you choose to draw the game (for example, you may choose your own colors) and therefore this class will not be tested automatically, but rather only tested manually by the graders (they will play your game).
This class has three public methods: **`setMouseLocation`** (called by the GUI whenever the mouse moves, and again receives an instance of the Point class you implemented in ex4), **`timerTick`** (called about 60 times per second by the GUI, and calls **Game**'s **`step`**) and **`drawGame`** (draws the game on screen).
While implementing the **`drawGame`** method, you will find the java.awt.Graphics API useful, and in particular its methods **`setColor`**, **`fillRect`**, **`setFont`** and **`drawString`**. The API of java.awt.Font and java.awt.Color may also be useful.
For example if g is a Graphics object (as the one given to **`drawGame`** as an argument) and you would like to draw a black rectangle on it, whose top left corner has the coordinates (10,20), whose width is 5 pixels and whose height is 8 pixels, you should write:

```
g.setColor(Color.BLACK);
g.fillRect(10, 20, 5, 8);
```

If you'd like to write Hello in white in size 40, such that the bottom-left corner of this text (i.e. of the letter H) has coordinates (50,100), you should write:

```
g.setColor(Color.White);
g.setFont(new Font(null,Font.PLAIN,40));
g.drawString(“Hello”, 50, 100);
```

Documentation

In this exercise, you are requested, as usual, to use Javadoc when documenting your code. The Javadoc, as always, should conform to the course guidelines and to the style taught in the lecture and in the recitations. You should not submit the html files generated by Javadoc.

School solution

An executable version of the school solution can be invoked from the lab computers by typing: `~introcsp/bin/ex6/kaboom`

Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: `~introcsp/bin/testers/ex6 ex6.jar` and verifying that all the tests pass successfully. You may also download the tests JAR from here. Extract the contents of the JAR file using the shell command: `jar xvf ex6testing.jar` and follow the instructions in the file called `TESTING`.

As mentioned above, you are required to write your own tester files for Rectangle and Circle. These should be called **RectangleTester.java** and **CircleTester.java**, and should be implemented using the JUnit4 library.
We expect you to write a comprehensive set of tests that will check that your classes function as they should. In order to help you write and test your tests, we provide you with a few buggy implementations of Rectangle and of Circle. As mentioned above, in order to get a full score for this task, it isn't enough just to write tests that find the bugs in our defected classes - you should write additional meaningful tests that thoroughly test your classes, also checking for other plausible bugs.
Your test classes should not contain a **`main`** method. You should either run them using eclipse, or using a driver that you will write. There is no need to submit this driver. (Our Ex6TesterDriver will also run them, but it will not work before you write the Game and Mediator classes.)
The .class files of the buggy and the correct implementations of the Rectangle and Circle classes on can be found in
`~introcsp/testers/ex6/buggy.zip` or through this link.
The files under directories beginning with 'Rectangle' are buggy implementations of the Rectangle class, the files under directories beginning with 'Circle' are buggy implementations of the Circle class. The exception is the files under directories ending with 'OK', these are the school solutions, for which your tester should pass. In case you tester does not perform as expected on any of our classes, the output of our tester will inform you which implementations your tester did not perform as expected on, so that you can try running your tests on them individually in order to investigate what has gone wrong.

Bonus

You may change the GUI by either enhancing the graphics drawing in **Mediator** or even by making changes to **KaboomView** (at your own risk!). If the graders really like the result, they may decide to give you a bonus of up to 10% of the grade. If you would like to be considered for a bonus, please detail your enhancements in the **README** file. In case you decide to change **KaboomView**, you should add it to your submission file and mention that in the **README** file as well.

Additional Files

We provide you with the driver for the game (**Kaboom.java**) and with the class that is responsible of the GUI (**KaboomView.java**). You should not change these files (except for as detailed in the bonus section above, if you choose to do so).

Running your game

In order to run your game, you should place, in one directory, the implementations of the classes Point (from ex4), Rectangle, Circle, Game and Mediator (that you wrote in this exercise), and KaboomView, GameParams and Kaboom (supplied by us - grab them from here). You should compile all these files and run the command

```
java Kaboom
```

Further Guidelines

- Please remember to follow the all the course coding style guidelines.
- In particular, remember to use constants where appropriate, and refrain from duplication of code.
- Document your code – add comments before all major code blocks.
- Write complete Javadoc comments whenever appropriate.
- Don't forget to write a short description of each class in your **README** file.

Submission

Submit a JAR file called **ex6.jar** containing ***only*** the following files:

1. **Point.java**
2. **Circle.java**
3. **Rectangle.java**
4. **CircleTester.java**
5. **RectangleTester.java**
6. **Game.java**
7. **Mediator.java**
8. **KaboomView.java** (only if you modified it)
9. **README**

Use the "Upload File" link on the course home page, under the ex6 link.

Good luck!