SquashIT

Documentation & Lab Report

26th September 2018

Contents

1 Documentation

1.1 Description

SquashIT is a simple keyboard-controlled game written in Java using the Processing framework. To launch it, import the project zip file into your Eclipse workspace and launch *Main.java* in the game.starter package. The game consists of a ball that moves across a game field and a paddle that's situated right above the game field's only pit. The goal is to prevent the ball from flying into the pit by blocking it with the paddle for as long as possible.

This initially easy task will become much more difficult as the ball gradually gains momentum, causing it to fly even faster over the game field. A timer in the top left corner counts the seconds the ball has survived without falling into the pit. A ball depot provides the player with exactly three balls which they can use to try for a best personal time. Once all of the balls are used up, the game is over and the best time out of the three attempts is displayed on the game over screen.

Figure 1: The game and its components in action

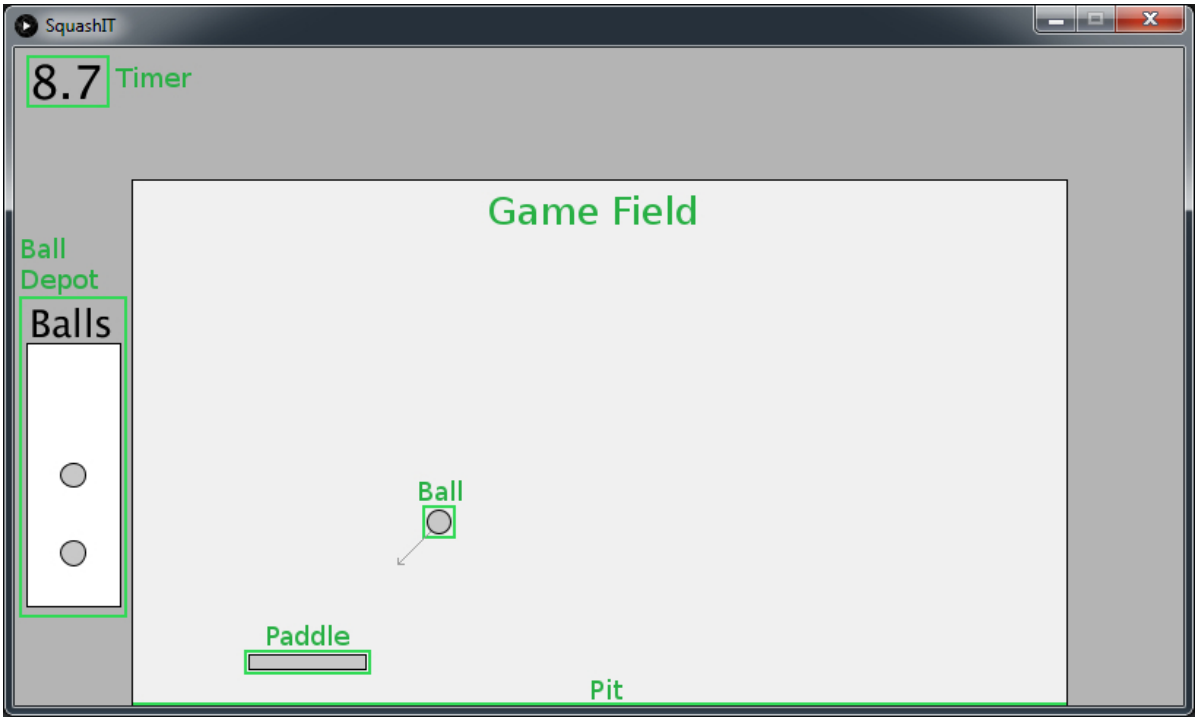
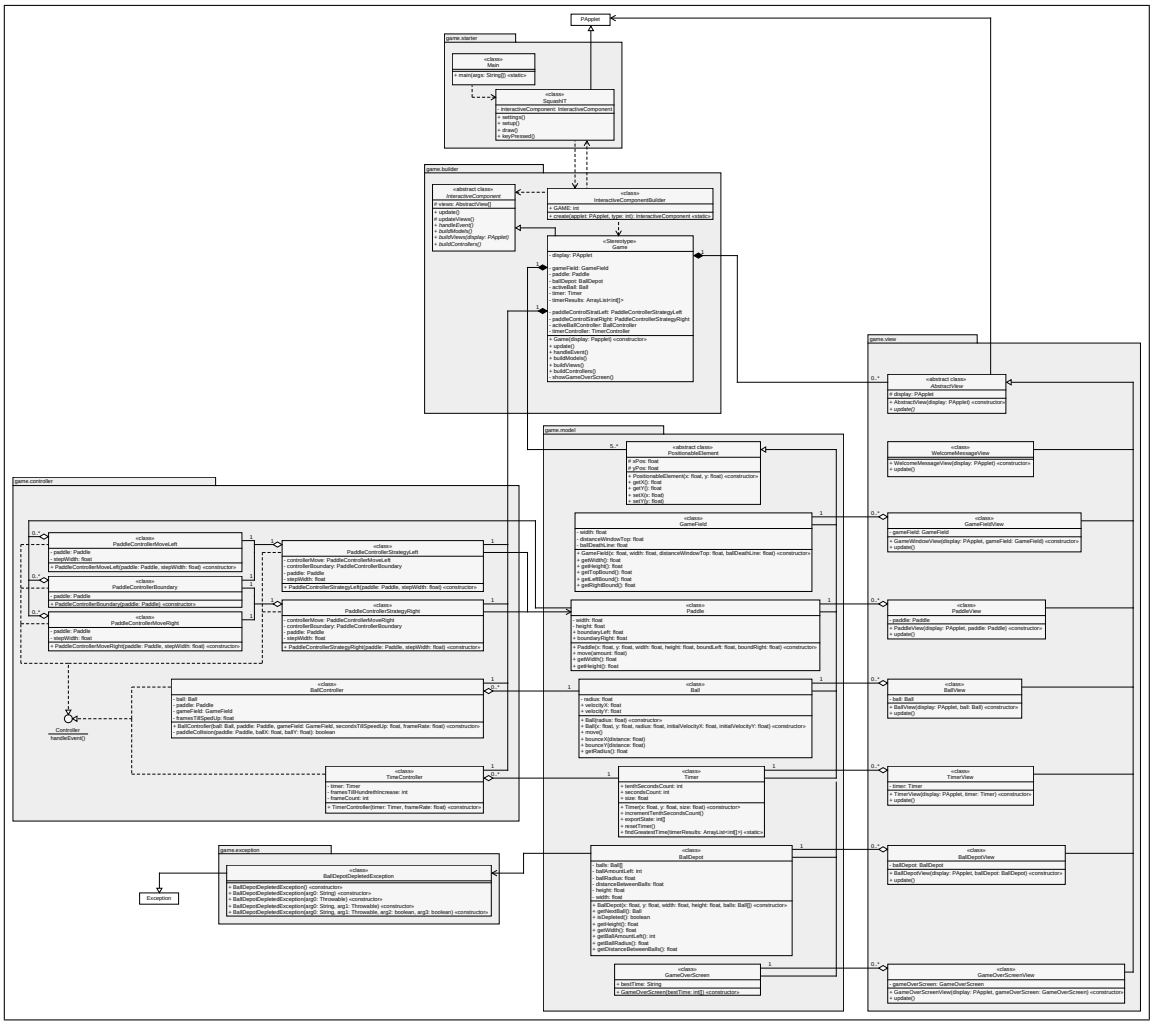


Figure 2: The game over screen

Figure 3: UML diagram of the final structure of the game



1.2 Controls

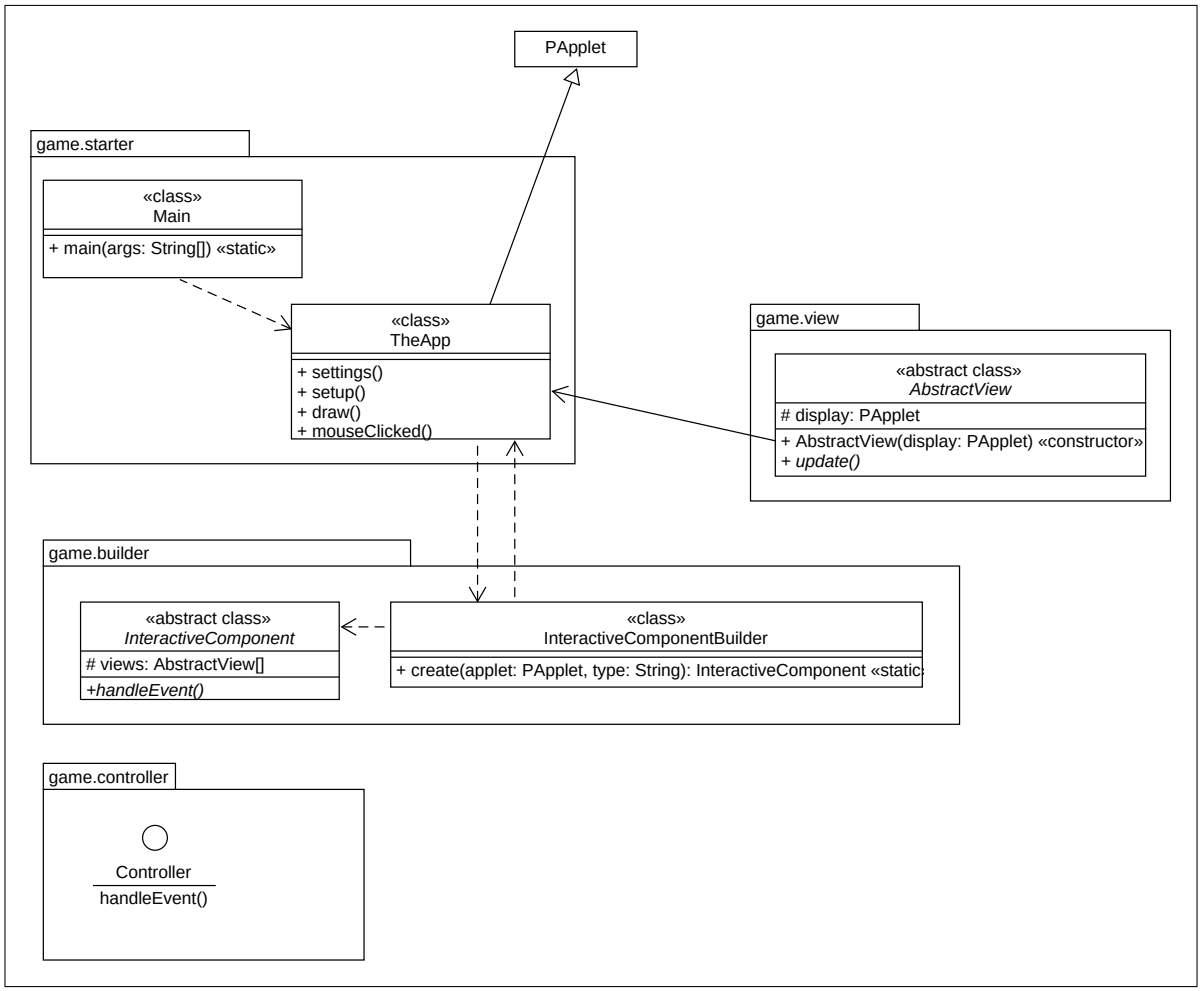
The controls of this game are fairly simple: all you need is a keyboard with a space bar and the left and right arrow keys. Whenever there is no ball in the game, you can press the SPACE bar to insert a new one, causing it to fly off towards the pit from the middle of the screen. The paddle is controlled using the LEFT and RIGHT arrow keys, allowing it to move within the bounds of the game field. A step moves it exactly half its length at a time.

Hint: Hold the LEFT or RIGHT arrow key for a while to make the paddle move much faster!

1.3 Final Project Structure

Figure 3 shows the final structure of the project.

Figure 4: UML diagram of the provided project template



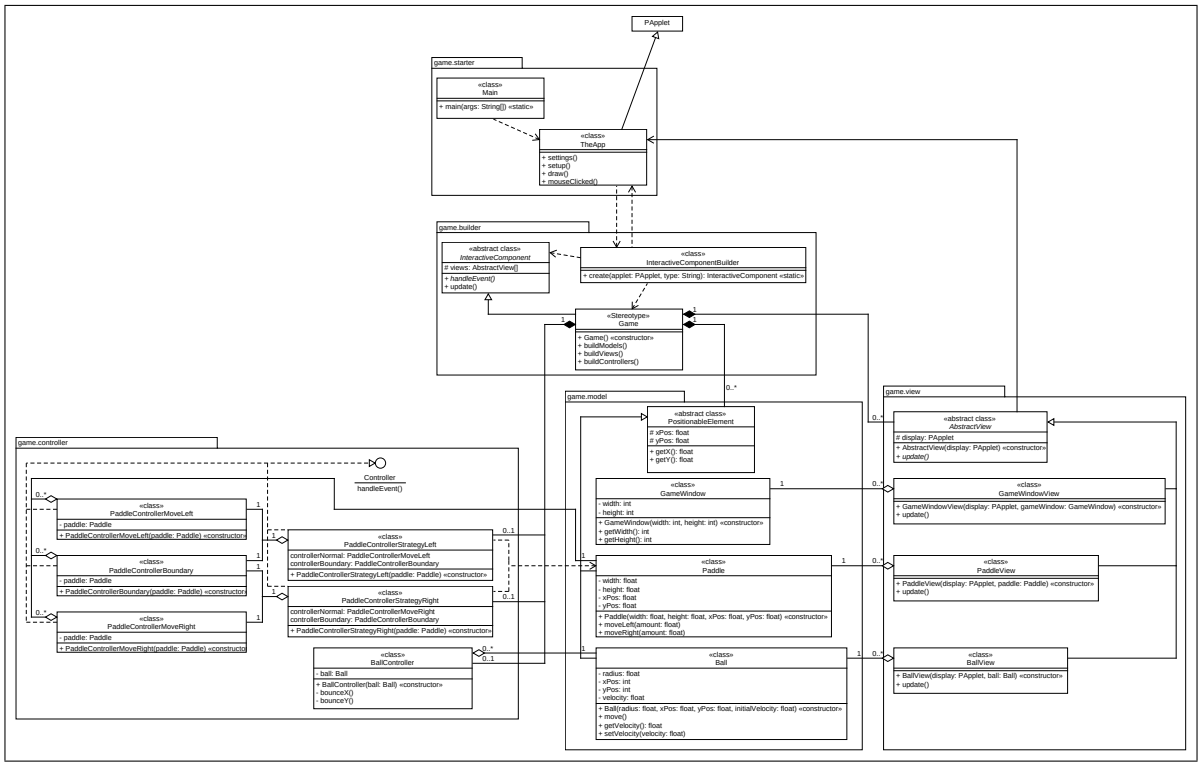
2 Lab Report—The development process

2.1 Planning the project

I began working on the project on September 16th, 2018. First of all, I pulled the given template repository "TheProject" from GitHub into my local Eclipse workspace. I did this using EGit, the built-in Eclipse git managing tool. The first real step in structuring my project was the creation of a UML diagram of what the template already gave me to work with. For this, I used the free software UMLet. The results of this are visible in figure 4.

Now that the framework the course instructors had given us was clear, it was time to plan the project. In the beginning, I wanted to keep things "simple", since I could still extend the UML diagram later on if needed. Therefore, I planned out a simple version of Squash. I had a bit of difficulty at understanding the purpose of the builder pattern,

Figure 5: UML diagram of my first project concept



but after studying the *capstone1-counter5* solution from the worksheets I knew what to do. Even though I didn't even consider a *BallDepot* or a *GameOverScreen* yet, the diagram already turned out to be quite complicated, as can be seen in figure 5.

I've added a few notable elements in comparison to the basic template UML diagram in figure 4. For instance, in the package *game.builder*, there is a new class *"Game"* which inherits from *InteractiveComponent* and can be built by the *InteractiveComponentBuilder*. It contains all the elements of the game.

In the *game.model* and *game.view* packages, I've added classes for the *Paddle*, *Ball* and a *GameWindow*. I created the latter because I didn't want the ball to collide with the actual window frame of the program. I wanted a dedicated "window", or rather, a field for the ball and paddle to move within. This field is a bit smaller than the actual window of the program and makes things look fancier in my opinion.

Finally, I also added an entirely new package *game.controller* to host the controllers of the models. While the ball only gets a single *BallController* for moving and bouncing, the Paddle is fitted with two strategies which in themselves host two controllers each—one for the movement, and one for the paddle to snap back to the closest game field boundary.

2.2 Implementing classes and making models visible

Now, I just had to implement the classes I conceptualized. I decided to work my way from the top to the bottom, so I began with the *TheApp* class. I changed the window dimensions to a 16:9 dimension. Additionally, instead of checking for the *mouseClicked* event, I checked for the *keyPressed* event, since I want keyboard controls for this game.

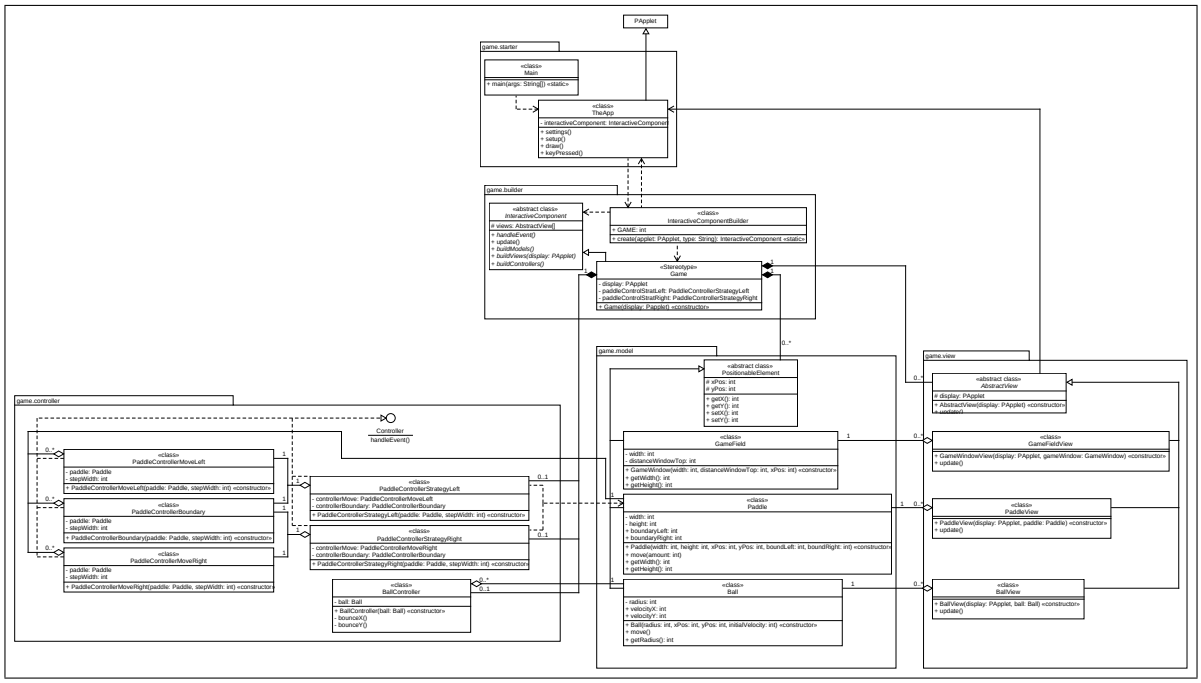
In the *InteractiveComponentBuilder*, I proceeded to define a constant *"GAME"* as an identifier to be passed to choose the "type" of component to build (of course, in this case, there is only one component we could build. But I decided to keep it like this for extensibility purposes). The given framework expected a string in the *create* method for that, so I changed the expected parameter *"type"* from a string to an integer. This seems like a better practice to me.

All the *InteractiveComponentBuilder* does is calling the build methods of the *Game* class in the right order. A problem I encountered was that in the switch-case-statement I used, the Game object I built inside it was actually a local variable that would be forgotten as soon as I exited the statement. Therefore, I couldn't return the object! But if I made a generic object of type *InteractiveComponent* in the global scope of the method (since the type of component to build isn't known before the switch-case-statement), the compiler would complain: an object of type *InteractiveComponent* doesn't have any build methods defined, and apparently, if I save a *Game* object in an *InteractiveComponent* variable, this is a problem. To circumvent this issue, I decided to define the three builder methods *buildModels*, *buildViews* and *buildControllers* in the *InteractiveComponent* class.

In order to continue with implementing the *Game* class, many other classes from the model, view, and controller packages were required, so I proceeded by creating a draft of all of the planned classes. At this point, I thought that integers were going to be precise enough for the coordinate system and lengths, so I made the decision to make every of those variables an integer instead of a float. While implementing these class drafts, I also made a few changes to the concept. For instance, the *Paddle Controllers* got an additional *stepWidth* attribute which decided how far the *Paddle* would actually move per key press. Additionally, I found that *"GameWindow"* wasn't a good name for that class. I decided to rename it to *"GameField"* instead. I also changed the attributes that defined the class. Instead of a *width* and a *height*, I decided to define it using a width, an x-position and a distance from the top. That would make drawing it much easier, as the *GameField* extends all the way to the bottom of the window. In this and the next step, I've made many changes which are mostly reflected in the updated UML diagram in figure 6. Since updating the UML diagram proved to be much work with little to no use in my workflow, this was the last UML update I made during the development process. For all further developments, you may refer to the final UML diagram in figure 3 on page 3.

After the classes were created in the project, I was able to implement the build methods. I used constants at the head of each method to increase readability and facilitate making changes later on. With the creation of the views, I was also ready to make the so far empty window display my game elements. At this point, I realized that the *Game*

Figure 6: Updated UML concept diagram



class constructor would need to take the display as a parameter, because it of course needs that for drawing. I implemented the update methods of each view and was left with my first visible results, as displayed in figure 7.

2.3 Implementing paddle and ball logic

Now that the elements of the game were visible, it was time to implement some logic. First of all, I added the input logic to the *Game* class in the *handleEvent* procedure. Depending on which key was pressed, the according controller is called. This key check is another reason as to why I needed to reference the display object within the *Game* class.

As for the controllers themselves: I began working on the Paddle Controllers first. Apart from the fact that I had forgotten to add the constructor in the UML diagram, I also needed to add an additional parameter, since the controllers needed to know how far the *Paddle* was allowed to move at all, and how far it was going to move per step. I decided to add a *stepWidth* parameter to the constructor which would determine the latter. For the former problem, I decided to add the boundary x-coordinates to the *Paddle* class so that the controllers could read them as needed. The *Paddle* would not be allowed to move past those coordinate boundaries.

For the *PaddleControllerBoundary*, I settled with it automatically detecting which boundary to snap back to when called, as I was using this same controller class for both Paddle Controller Strategies. Now, the *Paddle* was able to move as shown in figure 8.

Figure 7: The first working user interface

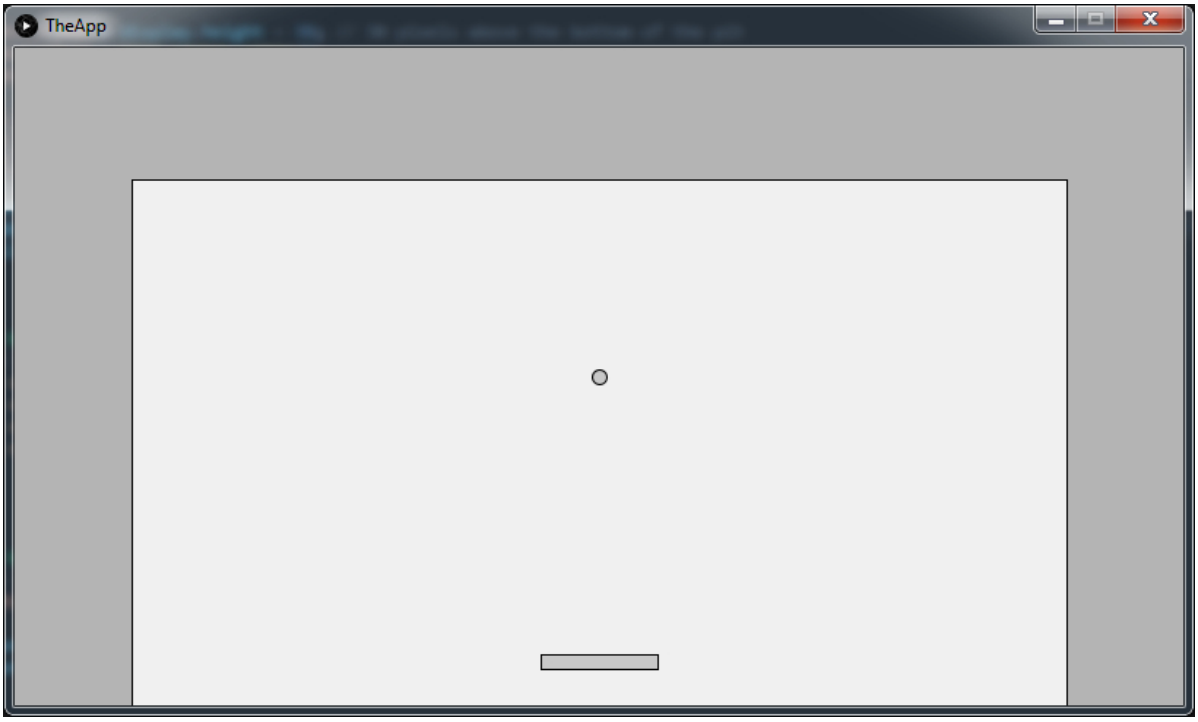
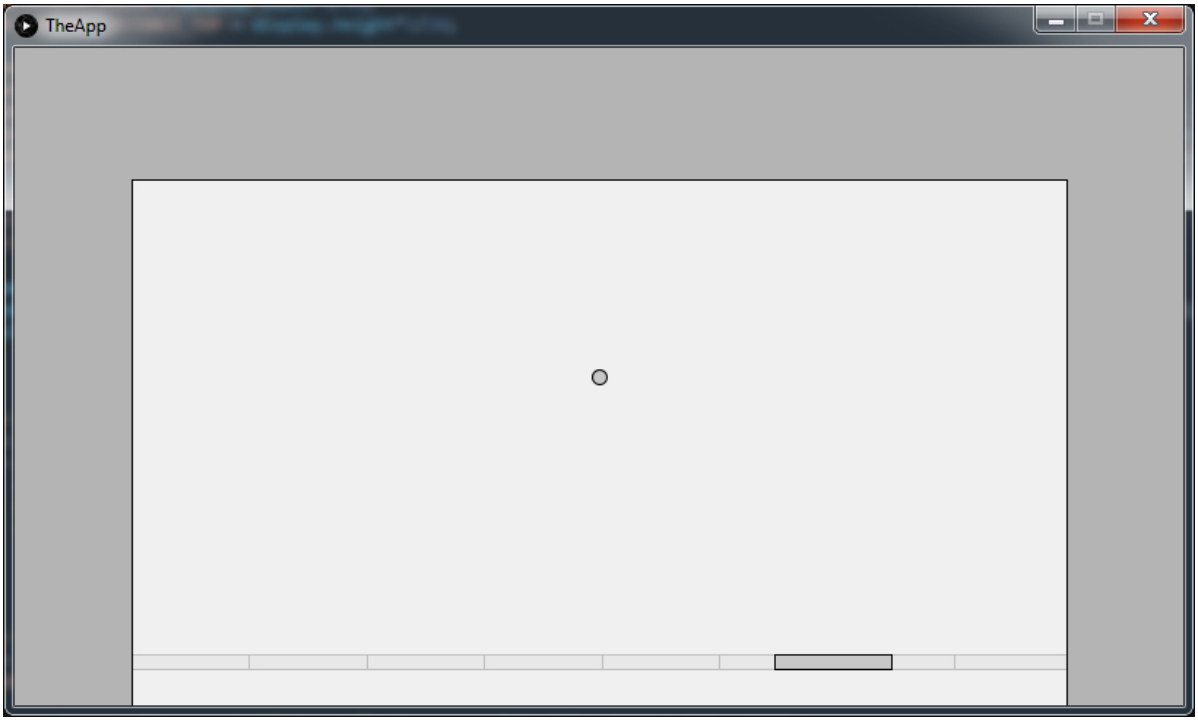


Figure 8: The possible positions of the paddle illustrated



With the paddle logic being finished, it was time to implement the ball logic as well. Here, I realized that the *BallController* needed much more information than I had thought initially. It didn't just need a reference to the *Ball*, but it also needed information about the *GameField* and the *Paddle* in order to bounce off them. In fact, every object that the *Ball* might bounce off of needs to be passed into the constructor of the *BallController*. Additionally, I had to add a *ballDeathLine* y-coordinate to the *GameField* so it's clear when the *Ball* has fallen into the pit. While I was at it, I also added convenience getters to determine the boundaries of the *GameField*.

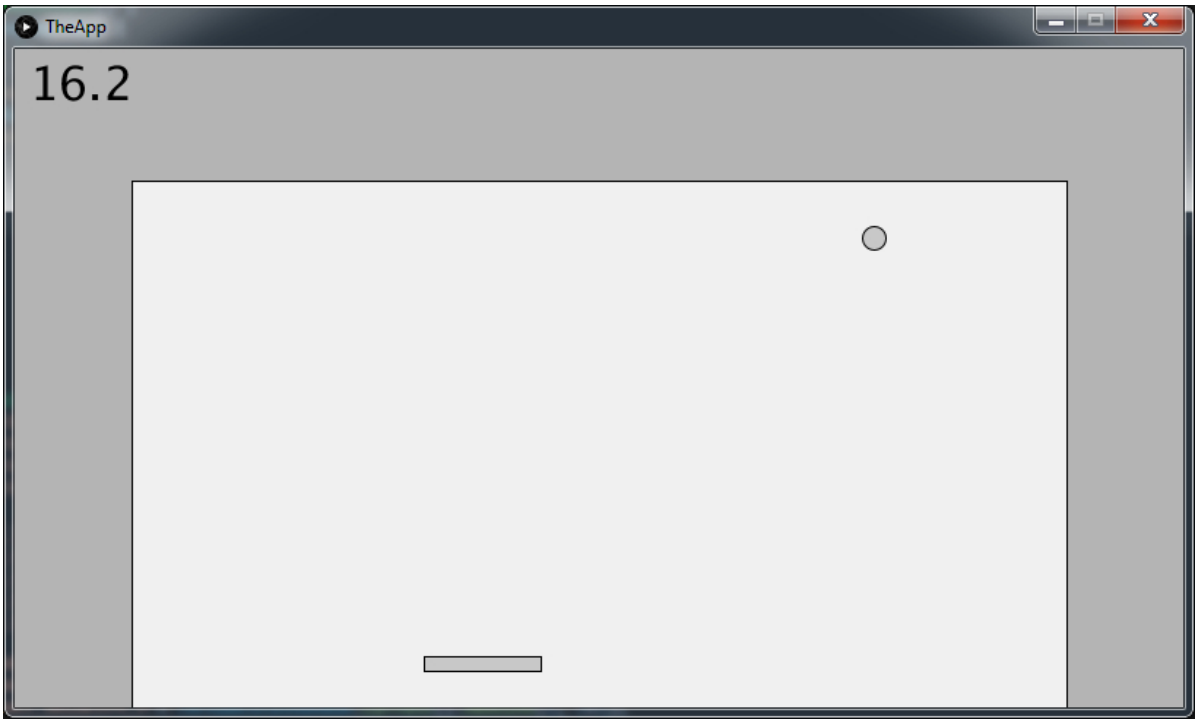
As already apparent, I've decided on a manual, coordinate-based approach on how to determine a bounce of the ball. For instance, whenever the *Ball* is within the boundaries of the *Paddle*, it is supposed to bounce and negate its y-velocity, i.e. fly back upwards. For velocities higher than one, the ball of course moves multiple units at once in a single step. I've accounted for that in the bounce methods of the *Ball* class, which don't just negate the ball's velocities, but also calculate the correct movement step. For instance, if the *Ball* is three units away from the *GameField* boundary, but it has a velocity of five, then it will move three units towards the boundary, at which point the bounce would occur, which in turn causes it to move two units away from the boundary again; and all this happens in a single step.

Now, in order to make sure the *Ball* was moved and thus updated every time the game screen was drawn, I had to update the controller in the *Game* class' *update* method. Unfortunately, that forced me to override the inherited update method from the *InteractiveComponent* abstract class, which already contained an implementation to update the views. In order to not repeat myself, I moved this functionality of this into a new inheritable procedure *"updateViews"* while letting the *Game* class define its own *update* procedure, in which *updateViews* would be called separately. Now, I could simply add a statement to update the *Ball* (i.e., to call the *BallController*) with every time the *Game* was updated. After fixing a few arithmetic mistakes I've made in the *BallController* and *Ball* bounce methods, the ball was able to bounce off the walls of the *GameField* and the *Paddle*.

Finally, I wanted to make the ball one unit faster every second. In order for the *BallController* to determine what one second is, it had to know the frame rate of the app, so I added a parameter for that into its constructor, along with another parameter to determine after how many seconds the ball velocity should increase by one. I added a counter to the *BallController* which counts how many frames have passed. After a certain number of frames had passed, the *Ball*'s velocity would be increased by one.

In theory, this worked perfectly. Unfortunately, it turned out to be a fairly ugly solution, as it was clearly visible when the ball was suddenly accelerated by one unit. So instead, I began working on increasing the *Ball*'s velocity gradually, a tiny bit every frame. Yet for this to work, I had to change all the variables which I defined as integers to floats again, or else this gradual increase wouldn't have worked. Since at this point it was clear that the additional accuracy of floats was useful to me, I went on to change every coordinate or length to a float. And finally, despite the *Ball* correctly moving faster the longer it was in the game, the increase of the *Ball*'s velocity was no longer noticeable.

Figure 9: The user interface after the timer was implemented



2.4 Implementing a timer and a ball depot

Now, I wanted to implement a timer to the player knows how long they've managed to keep the ball in the game. This proved to be a fairly easy task using the model-view-controller pattern. Once again, it was necessary to call the *TimerController* with every time the *Game* was updated. And to accurately count the seconds, it was also required to be aware of the app's frame rate again. I've decided that precision up to a tenth of a second would be the best for this game. The user interface with the timer implemented can be seen in figure 9.

At this point, I was a little lost, as I was unsure on how to go on. I had no idea how to implement and display the game over screen at the time. In order to procrastinate, I implemented a *BallDepot* first, since I intended to create one at some point in time, anyway. While for this, I used a model and a view, I decided not to implement a controller. It seemed unnecessary to me, as the controller had basically no logic to perform in this case, and a *handleEvent* method felt more restrictive than ever to me here. Instead, I let the *Game* class handle the *BallDepot* directly.

The *BallDepot* is a bit dynamic in the way it displays the *Balls* it contains. Depending on how many balls it's initialized with, the radius of the balls inside it is calculated. Unfortunately, it is not advanced enough to create multiple rows of balls if there is a very great amount of them. Instead, more balls than four tend to be displayed at a very tiny size.

Of course, the *BallDepot* has no real functionality yet. First, I let the currently active *Ball* in the *Game* die if it's below the *GameField*'s *ballDeathLine*. I did so by adding a check for it directly in the *update* method of the *Game* class. If the *Ball* had indeed passed the line, the reference to it as well as the reference to its controllers would be set to zero. Now, to prevent a *NullPointerException* due to e.g. the BallController being gone, I added a boolean *gameActive*, which basically determines whether there currently is a *Ball* moving in the game. If the game is inactive, the *update* method won't try to update the *Ball* or the *timer* (effectively causing the latter to freeze). In turn, the *Game* only reacts to the space bar (i.e. an attempt to use a new ball) if the game is inactive, which prevents the player from wasting a *Ball* for no reason. If a space bar press is handled while the game is inactive, the next *Ball* is pulled from the *BallDepot* using the *getNextBall* function and a new controller and view are created for it.

While implementing this behavior, I ran into a problem regarding the *BallView*. First of all, I always needed to know where the *BallView* was located within the *views* array of the *Game* class so I was able to destroy it once the *Ball* had died. To solve this issue, I just made sure the *BallView* is always the last element in this array. But of course, I couldn't just set it *null*, because that would have lead to a *NullPointerException* when the *Game* tried to update its views. So instead, I had to implement a dummy. Whenever there was no active *Ball* in the game, the active BallView would draw a dummy ball somewhere far outside of the window view. This might not be the cleanest solution, but at least it works.

2.5 Implementing a game over screen and a welcome message

Finally, it was time to implement the game over screen. Now that the ball depot logic had been implemented, it didn't seem so difficult anymore: the game only had to recognize whether the *BallDepot* is empty once the last *Ball* had died. So I added a function *isDepleted* to the *BallDepot* class which would return a boolean of whether it still contains a ball (false) or not (true). If the depot is depleted, a new *GameOverScreen* is created and shown.

I also wanted the *GameOverScreen* to display the best time the player had with his three balls. Therefore, the game had to remember the state of the timer before a ball had died. I added a method *exportState* that would export the *Timer*'s current second and tenth of a second into an integer array. Whenever a ball died, this state would be saved into an ArrayList *timerResults* in the *Game* class. A new static method *findGreatestTime* in the *Timer* class would find the best time of all, which can then be passed into the *GameOverScreen* constructor.

When the game ends, the *views* array of the *Game* class is replaced with a new array containing only one single element: the *GameOverScreenView*. That way, the *GameOverScreen* is the only thing that is drawn. While actually the controllers are still working in the background, the game ends up seemingly locked.

There was still room for one little extra: I decided to implement a welcome message which would tell the player how to insert their first ball. I figured that the arrow key controls would be straightforward enough to not require any comment. The welcome

message actually consists of only a view and not a model. It is created as the last element of the *views* array in the *buildViews* method of the *Game* class, basically replacing the dummy *BallView* that would have been inserted otherwise. Since a new *BallView* is created at this spot when a *Ball* is inserted, the welcome message disappears as soon as the game starts.

As the final polish, I decided to change the title of the game. Instead of "TheApp" being displayed at the top of the window frame, I wanted it to be called: "SquashIT". I achieved this by simply refactoring the name of the class *"TheApp"* to *"SquashIT"*. Afterwards, I only did a little bit of refactoring to increase the code's readability. With that, the development of the game had come to a finish.