**Breakout - Marek Hummel, Alexander Pluska**

Initially we split our program in the three packages “model”, ”controller” and “view”. This was the basis to work with. We decided to have the entry point of the program in the controller, meaning that the one controller class we have, “BreakoutController.java”, is a subclass of GraphicsProgram. The idea behind that was to have the controller as the hub of the program, so that it manages and administrates both the model and the view.

Starting with the model, we put everything inside there which calculates stuff. Most importantly there is the “GameModel.java”, a class which represents the entire game. Within that package we have another package called objects. It contains an interface “GameObject.java” which is implemented by all objects, another interface “Updateable.java” for the moving objects (paddle and ball) and one abstract class “RectangluarObject.java” to outsource some common method implementations. The single game elements are then “Ball.java”, “Brick.java”, “Bounds.java”, “Paddle.java”. Additionally, we needed some helpers, first a simple 2D-Vector class “Vector.java”, which we used as both vectors and points, and some general methods which we put in an (static) class “Helper.java”. And finally, we added an “LevelGenerator.java”, to have the option to add as many levels as wanted.

An important note is, that the model itself works on a frame which has dimensions of 0 to 1 on the x axis and 0 to a on the y axis, where a is a set aspect ratio.

The view contains out of two parts, the views and the renderers. The renderers take care of putting the graphics onto a screen, therefore there is a “DefaultRenderer.java”, which just paints the window of the application, and a “LighthouseRenderer.java”, which sends the data to the lighthouse. Both implement the interface “Renderer.java”, so that the Controller can handle any kind of renderer.

The views on the other hand are literally different views the player can have in the game, thus there is a “StartView.java” for the starting sequence, a “GameView.java” for the game itself and an “EndView.java” for an end screen. All of these implement the interface “View.java”, to group them like we did with the renderers.

The main thought behind this is to have an arbitrary renderer in the controller, giving it any view and that’s all that’s important, no further information needed.

Finally, the controller, which explains how our game works as well: The controller is the entry point, which means it initializes the window and goes after the “init-method” in the “run-method”. In these methods we have further initializations, one for the game model itself, one for the current view, and as many renderers as wished. Note that the renderers get the current view passed in their constructor.

What the controller now basically does is starting in endless loop, and each iteration it calls its update method and the render method of the renderers. To avoid having inconsistent iteration times or too fast iteration times, there’s a time control which pauses the current thread to achieve constant fps rate (frames per second) and especially an upper bound for the fps.

Updating means mainly, that the “update-method” of the game model gets invoked, so every member of the game updates itself. Furthermore, the current view may change, which means the StartView gets replaced after 3 seconds with the GameView, and the GameView gets replaced with the EndView as soon as the game is lost / won.

Rendering means, that the renderer calls the must-have “toGObjects()-method” of the view, and either places them on the canvas or converts its properties into the data-array for the lighthouse.

We are aware, that we could’ve limited the views to only one, and extracting the start and end sequence to the game model and let the model decide, when which part of the game is active, which would make sense. But this way we implemented the possibility to add multiple views, which might’ve been several menus, scoreboard etc. This would make the game model class explode in size or result in multiple main model classes.

In addition, we are also aware that the controller as an entry point is debatable, but for the expected complexity of this project and our possibilities with java and the acm library, we affirm our solution as an justifiable implementation.