Scale Game project document

Tommi Honkanen
1012602
Computer Science (1st year)

April 26, 2022

Contents

1 General description

Scale game is a game in which players compete against each other by stacking weights on top of scales and other weights. The game is turn-based and every players gets to place one weight each turn. Weights can be placed on any scale and players can choose the exact tile on which they want to place the weight.

Each player has a predetermined amount of weights which is chosen before the game starts. Using these weights the players have to collect as many points as possible before they run out of weights. The player with the most points at the end of the game wins. The game ends when all players run out of weights.

Each player is differentiated with a color. The possible colors are green, red, orange and blue. Players' weights are also colored using that color to separate them.

Points are determined based on how far from the scale's center the particular weight is. For example, a weight that is on a distance of 3 tiles from the center, grants the owner of that weight 3 points. Weights can also be stacked. If a player stacks a weight on top of other weights, that player is given the ownership of the weights that are below that weight and therefore also the points that those weights yield.

If a weight is on the right side of a scale on a distance of 3 from the scale's center, that weight also weighs down the right side by 3. The allowed imbalance before a scale tips over is equal to the scale's radius. So for a scale with a radius of 5, the maximum allowed imbalance is 5. If a player tries to place a weight somewhere that would cause a scale to tip over, the scale is not lost, but the player loses the weight and the turn is passed to the next person in line.

Scales can also be stacked on top of each other. At the end of each round there is a chance that a new scale will be placed on top of an existing scale. That chance is chosen by the players before the start of the game. Points given by weights that are on top other scales are multiplied by the distance of that scale relative to the lower scales center. So if a scale is on a distance of 2 from another scale's center. All points on the upper scale are multiplied by 2. If a tile has a scale on it, weights can't be placed on that tile.

I have implemented the project on intermediate difficulty. My original plan was to implement it on easy difficulty, but after realizing that intermediate would probably be easier, I decided to move to intermediate.

2 User interface

The game is started by running the GUI-object. After the program has been run, the start menu will open and the players are first prompted to choose the amount of players, followed by the new scale probability and the weight amount. These are all chosen using sliders. After that the players can press the button that says "Start Game" which causes the start menu to disappear and the actual game window to open.

The players can then choose who wants to start and begin playing. Each turn the player chooses the scale on which that player wants to place the weight, followed by the side of that scale and the distance from the center on which the weight will be placed. These are chosen using the menus at the top the interface. After choosing them, the player can press the "Play turn" button which places the weight and passes the turn to the next player in line. Green always starts the game and the turns go in this order: Green $\rightarrow$ Red $\rightarrow$ Orange $\rightarrow$ Blue. After this it is again Green's turn.

The game continues like this until every player has run out of weights. After that the winner will be declared (or a tie) and no more inputs can be given and the game can be closed.

3 Program structure

3.1 General descriptions of the classes

The program consists of 6 main classes. These are:

- Scale
- Tile
- Weight
- Player
- Game
- GUI (object)

Scale-objects represent individual scales around which the game revolves. Each scale has a unique symbol, a radius and arrays for tiles on its left side and right side. The lengths of these arrays are equal to the scales radius.

Each scale consists of tile-objects. Weights and scales can be placed on top of these. Each tile has a distance which tells how far it is from the scale's center, a buffer for the weights that are on top of it and an option that will hold the scale that might be placed on top of it.

Weights are weight-objects. These objects know the player that currently owns them.

Players-objects represent the players of the game. Each player has a unique color and information about how many weights the player has left.

Each game-object represents a single game. Here the individual parts are brought together to form a whole. The game class has information about its players, its scales, the new scale probability and whether the game has ended or not.

The GUI-object constructs the game's graphical user interface. It takes inputs from the players and it keeps track of the game's game-object, whose turn it is and some other miscellaneous properties to ensure that the program works as intended.

Of course many other kinds of class structures could've been used aswell, but for me this structure made the most sense. For example, the weight-class could've been omitted completely, but in my opinion making it a separate class made the program clearer.

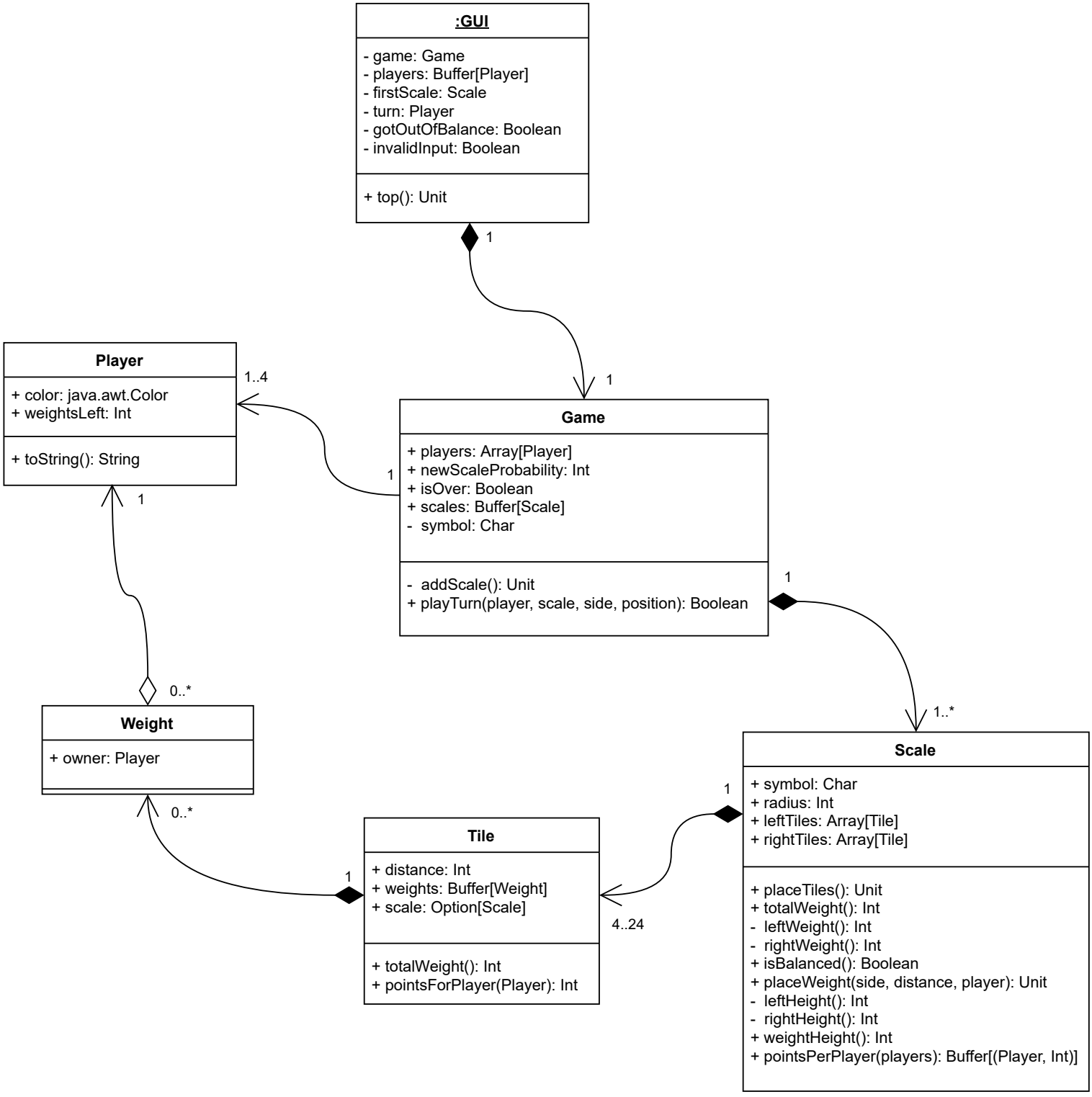

Figure 1: UML diagram

In theory, associations should have also been drawn between GUI and Player and GUI and Scale due to the players and firstScale variables in GUI, but for the most part GUI accesses those classes through the game-object so I decided not to include those associations for clarity.

3.2 The key methods of the classes

- Scale
  - totalWeight() calculates the total weight of the scale.
  - isBalanced() determines whether the scale is balanced i.e. whether the imbalance is greater than the radius of the scale or not.
  - placeWeight() is used to place a weight on some tile on the scale.
  - pointsPerPlayer() is one of the most important methods of the game. It calculates the points each player has on this scale.
- Tile
  - totalWeight() calculates how much this tile weighs.
  - pointsForPlayer() takes a player as a parameter and calculates how many points that player gets from weights on this tile.
- Weight
  - This class has no methods.
- Player
  - This class has no methods.
- Game
  - addScale() places a scale of random width on a random free spot on a random scale.
  - playTurn() plays the turn for a single player i.e. places a weight and determines whether a new scale should be added to the game.
- GUI (object)
  - top() builds the interface of the game and it holds several other methods inside of it, most notably paintComponent() which in turn contains paintScale() which paints the actual scales on the interface. paintComponent() also draws everything else on the interface with the exception of the buttons and the selection sliders and lists.

More thorough descriptions and descriptions of the less important methods can be found in the code comments.

4 Algorithms

The game utilizes the same kind of recursive algorithm in several methods. Arguably the most important instance of said algorithm is the pointsPerPlayer() method of the scale-class. This method uses recursion to calculate points for each player. It begins the calculation by looping over each tile of the scale. If it encounters a scale with weights on it, it calculates the points these weights yield using the tile's pointsForPlayer() method and adds these points to the sum variable. If instead it encounters a tile with a scale on it, it calls pointsPerPlayer() on that scale and multiplies the points that that scale yields by this tile's distance variable and adds them to the sum variable. Essentially the algorithm ascends to the top of the scale stack and then descends back to the bottom each time multiplying the points of the higher scale by the value of the distance variable of the lower scale's tile. When it comes back to the bottom, the sum variable contains the total points. Due to how the algorithm works, one call of pointsPerPlayer() on the bottommost scale is enough to calculate all points.

The algorithm in the totalWeight() method in the scale-class works very similarly. It also loops over every tile and if it encounters a tile with weights on it, it adds the sum of those weights to the weight variable. If a tile contains a scale, it calls totalWeight() on that scale and adds that weight to the weight variable. Thus it works pretty identically to the pointsPerPlayer() method. The main difference is that it doesn't use the distance variable to multiple the weight of the upper scales.

The paintScale() method in the GUI-object also works very similarly to these two. It draws the scale and while it loops over the tiles of the scale, if it encounters a tile with weights on it, it draws also those weights. If it encounters a tile with a scale on it, It calls paintScale() on it and passes it the location information of this tile. This way the whole scale stack can be drawn with just one call of paintScale() on the bottommost scale.

5 Data structures

These data structures were used to implement the program:

- Array

  Arrays were used in situations where it is known that the collection will remain the same size for the entirety of the game. These include, for example, the players array of the game-object and the tile arrays of scale-objects. The player amount is assumed to remain constant throughout the game so an immutable data structure is ideal for holding the players. Also the size of the tile arrays is always going to be equal to the radius of the corresponding scale-object so for the same reason arrays are adequate for holding the tiles.

- Buffer (Mutable)

  Mutable buffers were also used in several places throughout the program. They were used in places where the size of the collection could change during the game. They were used, for example, to hold the scales in the game-object. New scales can be added to the game while it is running so a mutable data structure was necessary. Buffers are also used to hold the weights on tile-objects. New weights are constantly added to these collections throughout the game so here a mutable data structure was also crucial. A buffer is also used to initially hold all the possible (4) players in the GUI-object and after the game has started this buffer is used to determine whose turn it is. This collection changes size countless times during the game

so a mutable data structure was an obvious choice. Originally, I had the pointsPerPlayer() method return a Map, but I ended up changing that to a Buffer also because I wanted to preserve the original order of the players and this seemed like the easiest option. Perhaps the original implementation was a bit more concise, but I feel like preserving the order was of greater importance.

6 Files and Internet access

This program doesn't have internet access or use any text files or any other external files.

7 Testing

The testing process went pretty much according to plan. At the beginning after any major changes to the program I always tested in REPL that everything was working as intended. This made it easy to spot any mistakes as quickly as possible.

After the main classes were mostly implemented I constructed an artificial game scenario and wrote unit tests for totalWeight(), pointsPerPlayer() and isBalanced() methods of the scale-class. These methods are crucial to the game, but small calculation mistakes in implementation may easily go unnoticed while playing the game so I deemed it best to write unit tests for these particular methods to ensure that they work properly. The unit tests came in handy since after any major changes to the program, I always ran the unit tests to ensure that I hadn't broken anything. I didn't feel the need to write unit tests for other methods such as addScale(), playTurn() or placeWeight() since mistakes in these would have been noticed when playing the game.

After the GUI was at a good enough state, I started to test the program by playing the actual game. This made it easy to spot any mistakes and bugs and make adjustments to the program. After playing a myriad of test games, it is unlikely that any bugs that I have yet to find still remain in the program.

8 Known bugs and missing features

8.1 Collisions

Easily the most troublesome bug that still lingers in the game is scales colliding with each other. By collisions I mean a situation where two (or more) scales have a tile in the same spot which causes the tiles of one of the scales to be hidden behind the other scale which looks rather disorienting.

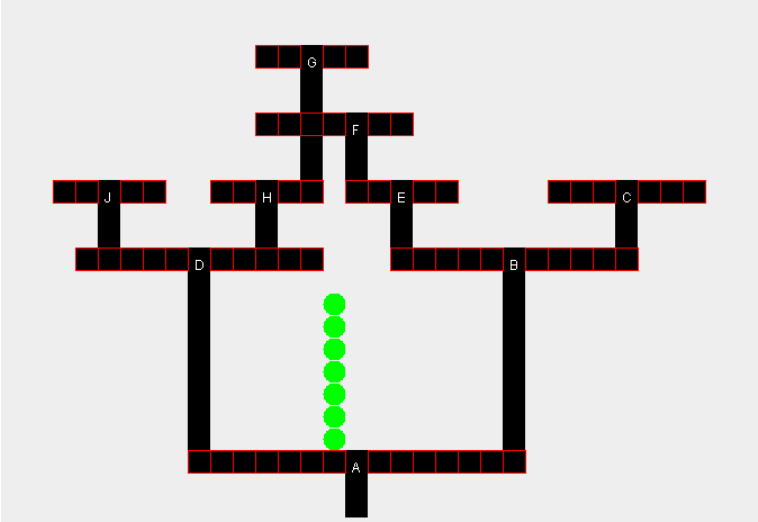

Figure 2: A Collision of 2 scales

In Figure 2 the F-scale has collided with another scale causing some of the tiles of that scale to be hidden. Luckily, this doesn't happen in every game and the probability of it happening greatly depends on the chosen weight amount and new scale probability. A game with a maximum weight amount and new scale probability has the greatest chance of experiencing collisions. I was able to greatly reduce collisions of weights and scales by making the height of the scale's leg increase as the weight stack on the scale increases (as seen in Figure 2) though it can still happen in rare situations but not nearly as often as collisions of scales.

I've tried to figure out as many ways as possible to mitigate collisions of scales and here is a list of everything I've done to achieve minimal collisions:

1. Limit the amount of scales a scale can have on top of it to 2 (one on each side).
2. Limit the amount of scales a scale with a radius of 2 can have on top of it to 1.
3. Prevent new scales from spawning on tiles with a distance of 1 (except for scales with a radius of 2, for which this hasn't been denied).
4. The radius of a new scale that spawns can be at maximum one less than the value of the tile's distance on which it spawns (but at minimum it can be 2 since scales with a radius of 1 don't exist).
5. Limit the maximum value of new scale probability to 50%.

These limitations somewhat limit the diversity of the game, but also greatly reduce the chance of scales colliding. Completely getting rid of collisions would probably require much greater limitations to the game and that would most likely be an even less pleasant choice. Another choice could be a rather complex algorithm that calculates a position and width for a scale that wouldn't cause collisions but an algorithm like that is beyond the scope of this project.

8.2 Scale tower's height exceeds the height limit

I haven't witnessed this myself but I suspect that it is possible that the tower of scales gets so high that the GUI runs out of space to display the whole tower. This would most likely only happen in games with more than 2 players and a high weight amount and new scale probability and the main cause would probably be the fact that the height of the scales' legs increase as the heights of weight stacks increase. So in a game with 4 players and a weight amount of 20, a total of 80 weights can be placed which could mean that some of the scales could get very tall. I tried to fix this by adding a scroll bar to the GUI but this didn't fix the issue. A simple fix would be limiting the maximum weight amount to something like 10, but I decided not to go through with that since this is presumably an issue that only affects larger games so having a weight amount of 20 shouldn't cause any issues in smaller games.

8.3 Towers of scales with a radius of 2

This isn't really a bug, but a quite interesting phenomenon nevertheless. As the game progresses, the radius of new scales decreases and at some point only scales with a radius of 2-scales will spawn as seen in Figure 3.

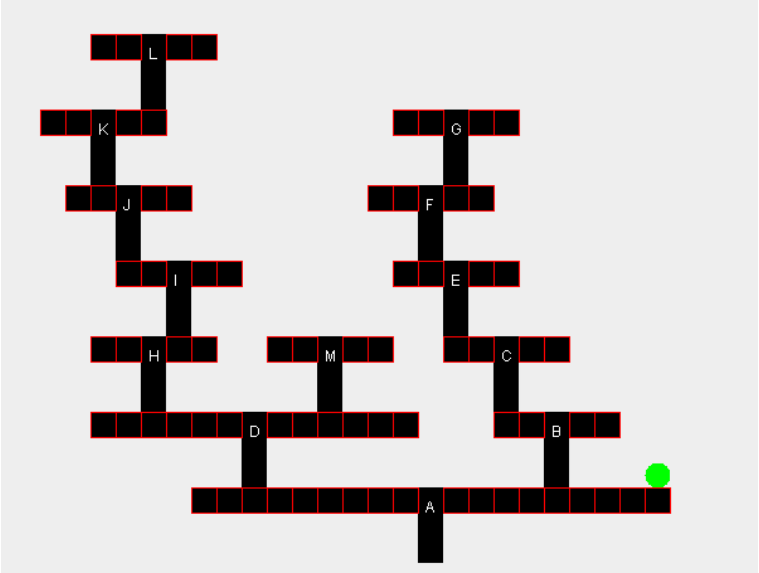

Figure 3: A tower of 2-radius scales

This happens due to the limitations mentioned in the Collisions part. Sometimes this happens earlier if the first scales that spawn are already quite narrow and sometimes this doesn't happen at all. The probability of this happening greatly depends on the weight amount and the new scale probability because if the game goes on for longer then the probability is obviously much higher. This phenomenon is the main reason why I decided to remove 1-radius scales from the game because this used to also happen with them and a tower of 1-radius scales is much more uninteresting compared to a tower of 2-radius scales because it is not even possible to get point

multipliers from 1-radius scales. Removing them increased the chance of collisions a little, but I had to make a decision and opted for removing them since I figured that it was the lesser of two evils. Getting rid of this phenomenon completely would require removing the limitations mentioned in the Collisions part, but that would naturally cause a lot more problems than it would fix.

8.4 The worst case scenario

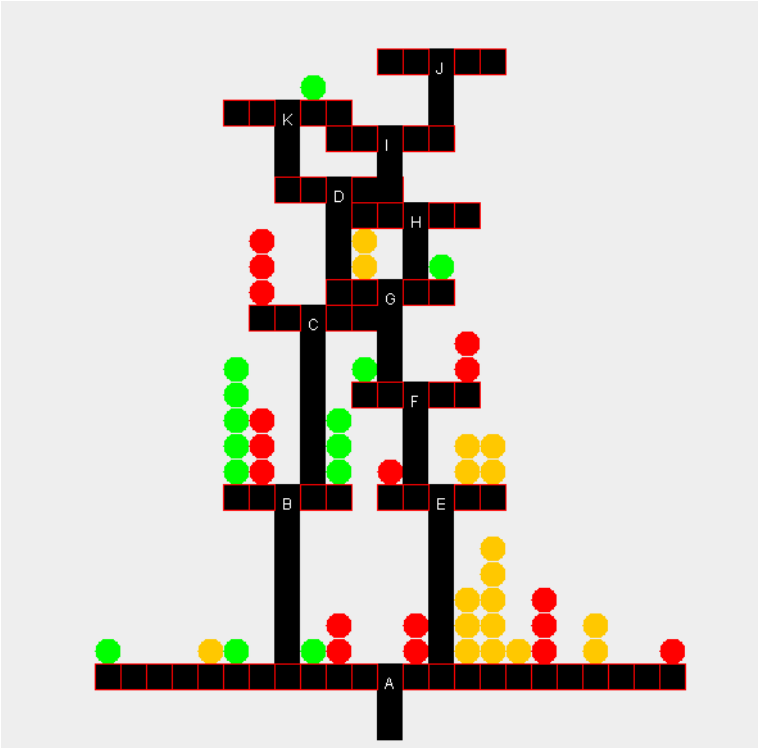

Figure 4: The worst case scenario

Figure 4 displays what can happen in the worst case scenario. It shows the problems 8.1 and 8.3 combined in the same game. Essentially everything that could go wrong, went wrong in this particular game which caused this to happen. Both the scales that spawned on the bottom scale were 2-radius scales, which will cause only 2-radius scales to spawn for the rest of the game. The scales also spawned very close to each other which makes the game prone to collisions. This was also a game with the new scale probability and weight amount set to the maximum values, which is the worst combination if you want to avoid these from happening. Luckily something of this magnitude only happens very rarely and even in this situation the game was still playable.

9 3 strengths and 3 weaknesses

9.1 Strengths

1. Collision evasion

   Although collisions still exist, I think I've done a fairly good job at reducing them as much as possibly while not limiting playability excessively. Although this wasn't a requirement in the project instructions, I feel if I had not done anything to reduce collisions it would've made collisions way too prevalent therefore reducing playability colossally.

2. Easy to play

   I think I've made the game reasonably easy to operate. The start menu is pretty self-explanatory and the game restricts the players from giving misinputs, for example, if a player tries to place a weight on a tile that has a scale on it. Although the GUI doesn't look the most visually appealing, it is still easy to make sense of and gameplay for the most part is flawless excluding the occasional collisions.

3. Class structure

   In my opinion, the class structure makes sense. All the clearly separate entities have been made into their own classes, but at the same time there aren't too many different classes which would have reduced the code needed per class, but would have also made the relations between classes more difficult to comprehend and just made it unnecessarily complicated.

9.2 Weaknesses

1. Efficiency of some of the algorithms

   Some of the algorithms could be made more efficient. For example the point counting algorithm in pointsPerPlayer() loops over each scale separately for every player although it could just do it once and count the points for all players at the same time. This would most likely be an easy fix but it doesn't seem to affect the performance of the game and I didn't want to try fixing something that already works.

2. Complexity in some parts of the code

   The code in certain parts is quite repetitive and there are a lot of methods chained one after another which makes those parts somewhat hard to read and interpret. I have to mention at least the addScale() method of the game-class. This could be improved at least a little bit by dividing the method into a couple of separate methods. However, at the end of the day this isn't going to do much since the same code would still exist in the program and the big method chains can't be partitioned very well anyway.

3. Limitations to the scales

   Although justified, I still want to mention the limitations mentioned in the Collisions part here. Even though in my opinion the limitations are necessary for the playability of the game, in an ideal world they wouldn't have to exist. It is possible that an easy fix exists that just hasn't crossed my mind, but to my current knowledge I think I've done the best I could.

10 Deviations from the plan, realized process and schedule

There weren't any major deviations from the plan, except that the project ended up taking quite a lot more time than I had anticipated. This was mainly due to the fact that I ended up making a GUI instead of a text-based interface. A more accurate breakdown of all the 2 week periods is provided below. Although testing isn't mentioned there, of course testing was also done in every period. Implementation order, for the most part, didn't deviate from the plan.

28.2 - 14.3.2022

During this time period, I implemented working versions of all the methods than handle the functionality of the game (Scale, Game, Weight, Tile, Player). I had originally reserved 4 weeks for this, but 2 weeks turned out to be enough. At the end of these 2 weeks, I also wrote the unit tests which I hadn't planned on writing this early.

Hours spent: **20h**

14.3 - 28.3.2022

Here I was 2 weeks ahead of schedule, but I ended up wasting the first week by working on the text-based UI that I ended up ditching. The latter week I spent researching information about swing before I started implementing the GUI during the following weeks. It was good that I had time to spare here, so I didn't fall behind of schedule.

Hours spent: **20h**

28.3 - 11.4.2022

During these weeks I managed to implement a working version of the GUI. I also did some modifications to the other classes but the bulk of the time was spent working on the GUI. Here I was pretty much in sync with the plan since this was the period I had reserved for implementing the GUI (although originally it was supposed to be a text-based interface).

Hours spent: **25h**

11.4 - 25.4.2022

Here the work was pretty much done, but during this time I still ended up making quite a lot of changes to improve the gameplay experience based on my testing. This included, for example, removing 1-radius scales from the game. Here I was in sync with the plan aswell since I had reserved this time for testing the game and making minor improvements.

Hours spent: **7h**

11 Final evaluation

Everything I wanted to say has already been explained in other parts of this document, thus I will refrain from going into too much detail here, but overall in my opinion the project went better than I had expected. The only thing that still bothers me is that I wasn't able to fix the collisions, but at least I can get some peace of mind knowing that I reduced it significantly. Of course there is always something that could be done better, but I'm happy with how the program turned out. It can for the most part be played without difficulties and there aren't any mistakes (that I have found) that completely break the game, for example, by crashing it. I also haven't encountered any problems with the performance of the game. It seems to perform fast even when the scale tower becomes tall. I wonder if there could be a performance issue on less powerful processors, but I doubt it.

If I had a lot of time on my hands, I would mainly focus on trying to fix the collisions since that is the main shortcoming. Other things that could be improved are the graphics and making the GUI responsive by making it adapt to the player's monitor size and generally making the window size adjustable.

I am happy with the data structure choices and the class structure. The chosen data structures achieve the desired goals well and there aren't any problems with their efficiency, therefore I don't feel the need to change them. Of course many different class structures could be used, but as I have stated before, this makes the most sense in my opinion. Perhaps some of the methods could be broken up into several smaller methods in order to prevent a single method from containing too much code, but personally I like the current method layout and I'm not certain that it would've made much of a difference.

I consider the structure of the program well suitable for making changes or extensions. Everything is divided well into separate classes and there isn't anything restricting changes. I don't see why extensions couldn't be rather easily added to the program and I've found making changes to the program to always be a seamless process.

If I started the program again from the beginning, the one thing I would change would be that I wouldn't waste time trying to implement the text-based UI. I spent a lot of time trying to figure that out just to come to the conclusion that a GUI would be much easier. Otherwise, I'm happy with how I implemented the program and I don't see the need to alter the process. I didn't encounter any obstacles regarding the implementation order and I managed my time fairly well.

12 References

Like I had planned, I ended up using the Scala API Description for Basic Class Libraries to look up collection methods. I also used the Scala Swing API to help me with the GUI. An invaluable resource ended up being a playlist called Introduction to the Art of Programming Using Scala (Part I) by Mark Lewis. This was hands down the most useful resource I relied on when implementing the GUI.

13 Appendices

13.1 Screenshots of different parts of the game

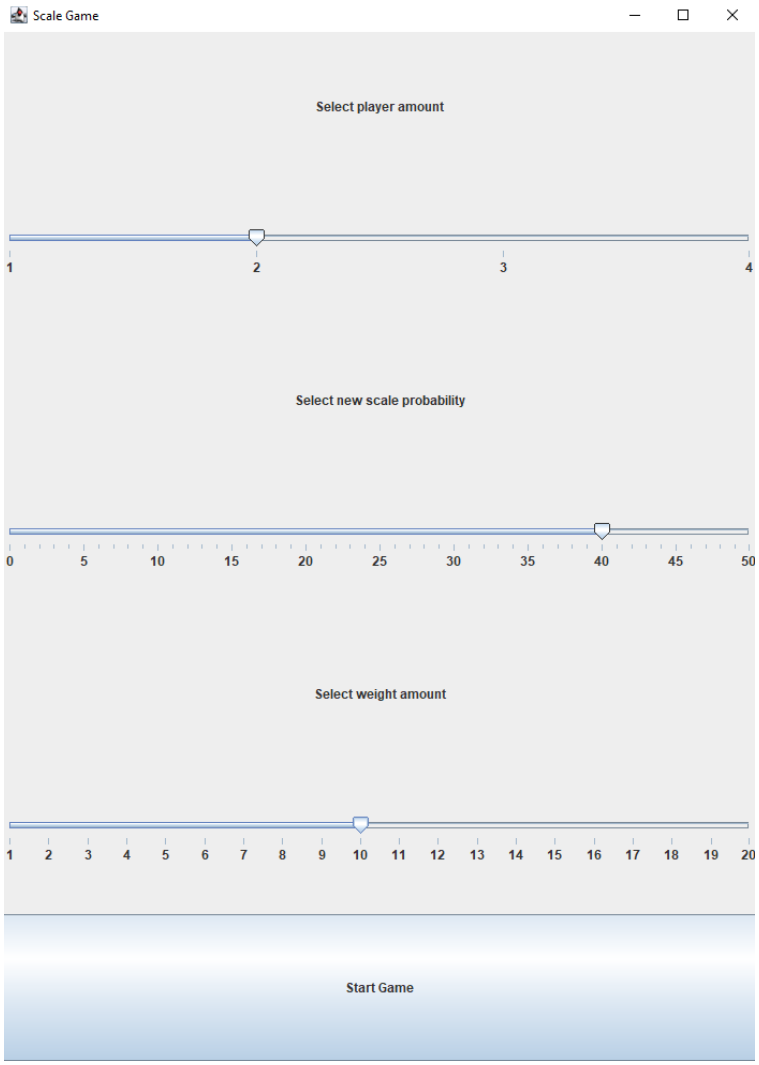

Figure 5: Start menu

Figure 5 displays what the start menu looks like. This menu appears immediately after the program has been started. Here players can choose the player amount, the new scale probability and the weight amount. After they are done, they can begin the actual game by pressing the "Start Game" button.

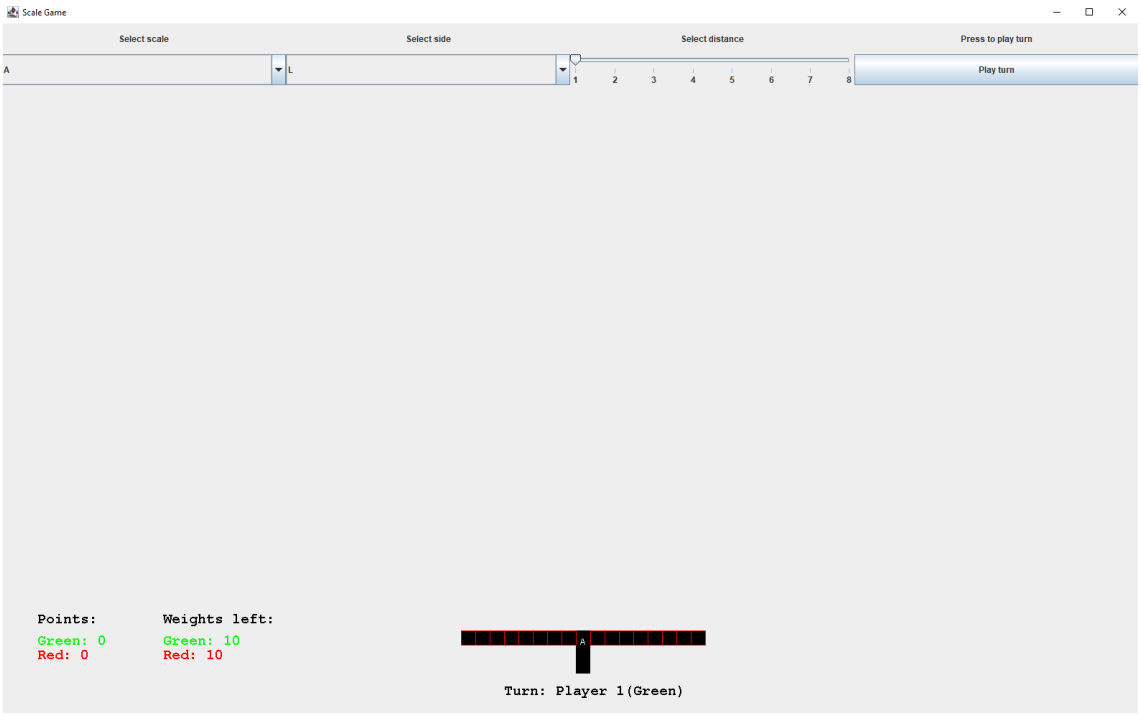

Figure 6: The start of a game

Figure 6 shows what the GUI displays immediately after the "Start Game" has been pressed with the settings shown in Figure 5. The current points and weights left are displayed in the bottom left corner, the controls are at the top of the GUI and the text below the first scale displays whose turn it is.

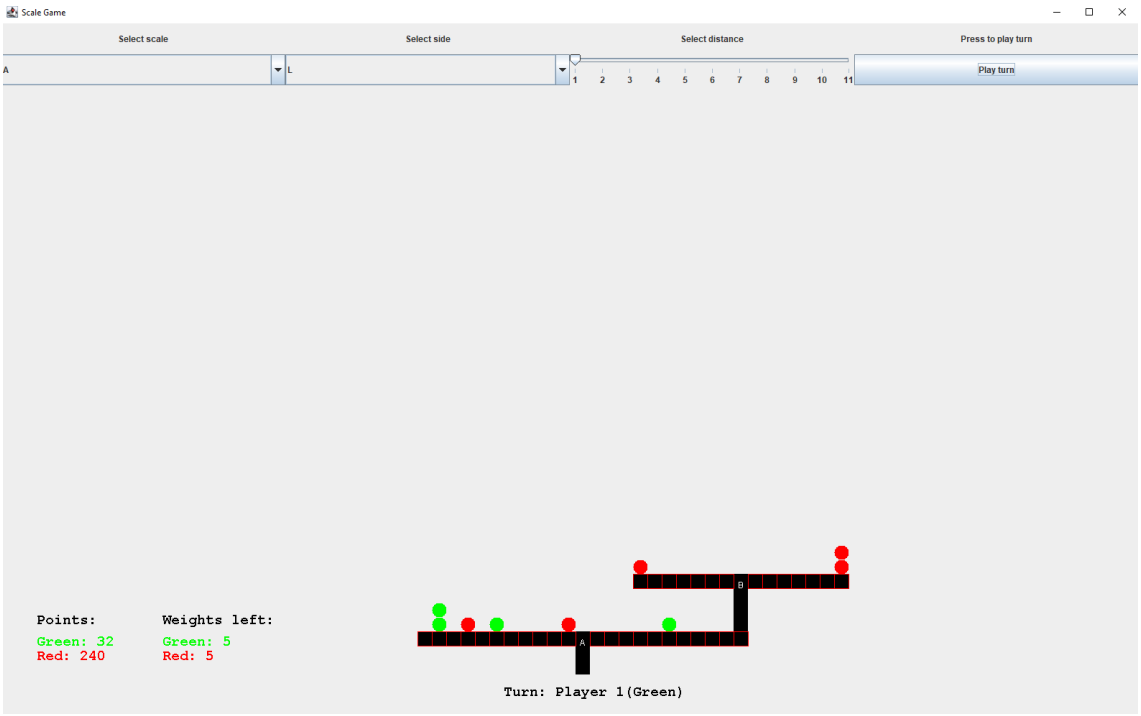

Figure 7: The middle of a game

Figure 7 displays what the middle of a game could look like (different game than in Figure 6). Both players have placed 5 weights (1 was lost because it tipped a scale) and a new scale has appeared.

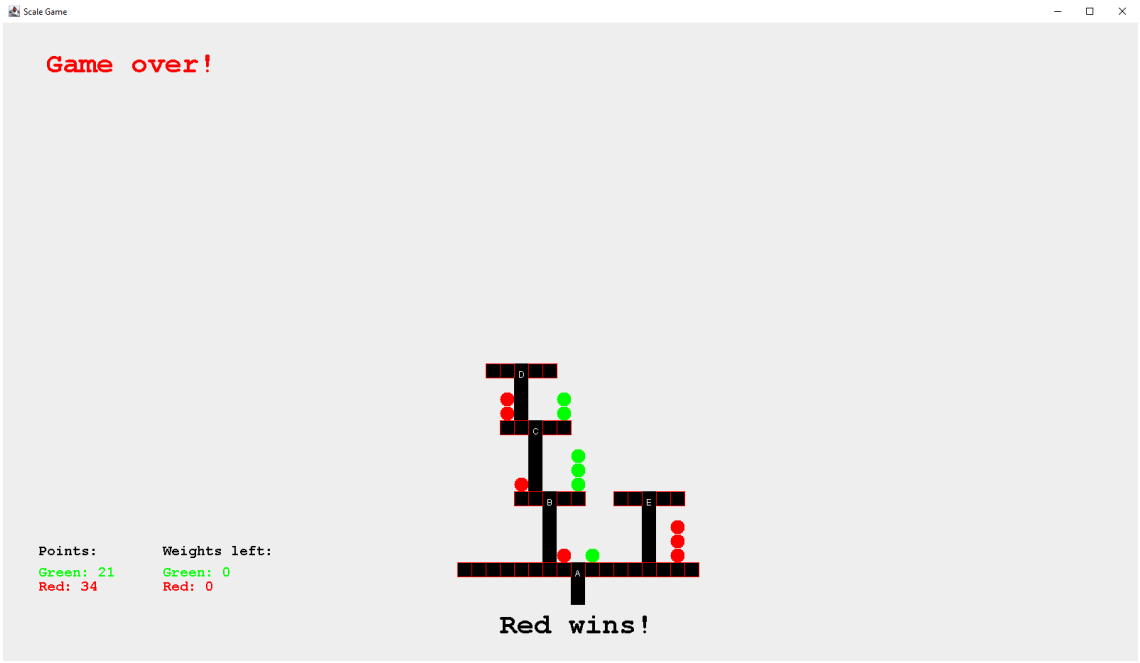

Figure 8: The end of a game

Figure 8 shows and example of what the end a game can look like (different game than in Figure 7). The controls are removed and the winner is declared (or a tie). No more inputs can be given after the game has ended.