# Documentation

Balancer

Final Project for CS-C2120

**Hau Phan**
886690

Bachelor Degree in Science
Data Science Program
Years of study: 2020-2023

Department of Computer Science
Aalto University
Finland, April 24, 2021

# Contents

# 1 General Description

A strategy game where players tries to place weights on a set of scales while keeping the scales balanced. The "riskier" the weight, the more score it yields. Player with the most score win the round. The game spans over multiple rounds and the winner is determined by the number of rounds won.

## 1.1 Concepts

### 1.1.1 Weight

A simple weight that have some mass. All the weights in the game have the same mass of 1. The farther the weight from the scale center, the more score it yields. Weights can be stacked.

### 1.1.2 Scale

A scale means a supported long "board" on which you can place other scales and weights. Here the term "board" is used flexibly to mean the two arm of the scale, the left arm and the right arm:

```
                A
            A   A       A     A
           <1=A=1>   <3=2=1=?=1=2=3>
              b             a
              *           A *
         <5=4=3=2=1=?=1=2=3=4=5>
                    a
                    *
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The farther the weight from the center point, the more torque it applies to the scale arm. The imbalance metric is the difference between the torques. Note that a small imbalance is permissible. The maximum allowed magnitude of the imbalance is the same as if 1 weight was placed at the end of the scale before the other weights were set. That is, for a balanced scale, the total imbalance $\geq$ the “radius” of the scale. (See examples below). Once a scale is tipped/flipped, it will be removed along with all its weights.

**Example 1**

```
               A
     A         A
    <3=2=1=A=1=2=3>
           a
           *
XXXXXXXXXXXXXXXXXXXXXXXX
```

If a weight is at a distance of 3 from the center of the arm, it pushes that side down with the force of three weights. If the other side has two weights at a distance of two, they push that side down with a force of four weights. (2 * 2 = 4). The imbalance is 3 - 4 = -1. The absolute value of the imbalance is less than the radius of the scale, 3, so the scale is balanced.

**Example 2**

If the left arm of the previous example did not have a weight, the scale would have been unbalanced (flipped) since the imbalance would have been 4, which exceeds the radius of the scale. In this situation, the scale is flipped and all the weights is lost.

You can also place other scales on the scale. The scale placed on the second scale applies a force equal to the sum of the weights and scales on it.

**Example 3**

```
         A
     A   A       A     A
    <1=A=1>   <3=2=1=?=1=2=3>
       b             b
       *           A *
  <5=4=3=2=1=?=1=2=3=4=5>
             a
             *
             *
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Scale *b* is balanced, the right-side weighs 2 * 1 and the left one 1 * 1 which does not exceed the radius of the scale. The scale B itself weighs three weights.

Scale *c* is balanced since the left side weighs 1 * 2 = 2 and the right side 1 * 1 = 1. The difference of these is 1, which does not exceed the radius of the balance. Scale *c* weighs 2 weights!

Scale *a* is also balanced. On the left side there is a scale *b* at a distance of 3, 3 * 3 = 9. On the right there is scale *c* at a distance of 4, 2 * 4 = 8, and a weight at a distance 3. All together there is a weight of 11 weights on the right side. The difference is 11 - 9 = 2, which is smaller than the radius of scale *a*, so the scale is balanced.

### 1.1.3 Players

There are two type of players: **human** and **bot**. Multiple players (recommended 2-3) can play the game at any one moment.

### 1.1.4 Capture and ownership

Both the **weights** and **scales** during each round is either **player-owned** or **wild** i.e owned by no one. The wild weights and scales are placed randomly at the beginning of each round and can be captured by players. Players can also captures each other weights and scales. Captured scales give addition "buffs" for the owner's weights while capture weight give points. Initially, the weight is owned by the player making the move.

Weights can be **stacked**. One can capture all the weights underneath by placing his/her weights on top. To capture a scale, one must have **at least** $r$ more weights than the player with the 2nd most weights, where $r$ is the radius of the scale. (See examples below)

Owner of a scale will have the following benefits:

- His/her weights on the scale can not be captured, even when other players place the weight on top. The moment the scales' owner change, all weight that should be captured will be updated according to the rules.
- Each of his/her weight will have its score multiplied by 2.

**Example:**

```
             A
      A B    A C
     <2=1=A=1=2>
          a
          *
          *
 XXXXXXXXXXXXXXXXXXXXX
```

Here the scale is captured by player A, since there are 3 A weights and only 1 B and C weight. The difference is $3 - 1 = 2 \geq 2$ the radius of the scale. The letter A at the middle indicates the owner of the scale

## 1.2 Rules and Gameplay

### 1.2.1 Gameplay

- There is a predefined number of round in one game. (Usually 5 for 2-3 players)
- Each round will have a fixed number of weights that all players will draw from and place onto the scales. (Usually 10-15 weights per round)

- At the beginning of each round, one scale and 5 wild weights will be randomly placed by the computer with some predefined probability.
- The round starts and the players place the weights one at a time on the scales.
- The round is over when there is no weight left. The scores are then calculated and the player with the highest score win the round.
- The next round then starts and **continues** from the last round's state.
- The player who win the most round wins the game.

### 1.2.2 Scoring

**Example:**

```
         A
   A B   A
  <2=1=A=1=2>
       b
       *         C B
       *   ?     C B
      <3=2=1=?=1=2=3>
             a
             *
             *
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Status: equalibrium

a: scale b: (2 * 1 + 1 * 2) * 2 = 8 points
   scale a: 3 * 8 = 24 points
   -------------------------------
   total: 24 points

b: scale b: 1 * 1 = 1 points
   scale a: 3 * 1 + 3 * 2 = 9 points
   -------------------------------
   total: 9 points

c: scale a: 2 * 2 = 4 points
   -------------------------------
   total: 4 points
```

Here '?' represents wild weights and uncaptured scales.

The scale with index 2 is capture by the player "a" so its points is multiplied by 2

### 1.3 Difficulty level

Completed level of difficulty: Intermediate/Difficult

The project succeed in meeting all the requirement of Easy and Intermediate criterions. Both a console-based and graphical user interface was implemented. To meet the Difficult requirements, an intelligent computer opponents was implemented along with additional game mechanics such as capturing scales, weights and random computer-generated scales. The project also aim at a creating a fully functional game: splash art, animations and tile textures is also added to improve the visual of the game.

# 2 User interface

## 2.1 Graphical

### 2.1.1 Game loop

When the user open the game, a splash screen is shown with animated logo and background. The user can click on the top left of the menu bar to select a new game, load a saved game or quit.

If the user choose to start a game, the game then switch to the game screen where player can interact with the game.

The game screen is shown on the left side of the window. Player can click directly on the scale to place the weight, dragging to pan screen and scrolling to zoom in and out.

The game information is shown on the right side of the window. This includes the score of each player, the number of rounds won by each player, the round number, how many weights left in the pool, .... There are also undo and redo buttons to undo previous moves, and buttons to add additional scales and weights if needed.

At the end of the round, a pop-up will be shown and announce the round's winner. The game then continues to the next round, randomly placing 5 weights and a scale. If it is the final ground, a pop-up will announce the game's winner instead.

### 2.1.2 Actions

Player can choose to start a new game by clicking "File" > "New" or go back to the menu with "File" > "Back to Menu".

At any moment, the user can choose to save the game by clicking "File" > "Save", exit the game completely by "File" > "Exit" or go back to the splash screen by choosing "File" > "Back to Menu"

### 2.1.3 Settings/Options

Additional players can be added by clicking the "Edit" > "Add Human" and "Edit" > "Add Bot". Undo and Redo button can also be found in the "Edit" menu.

Figure 1: The splash screen

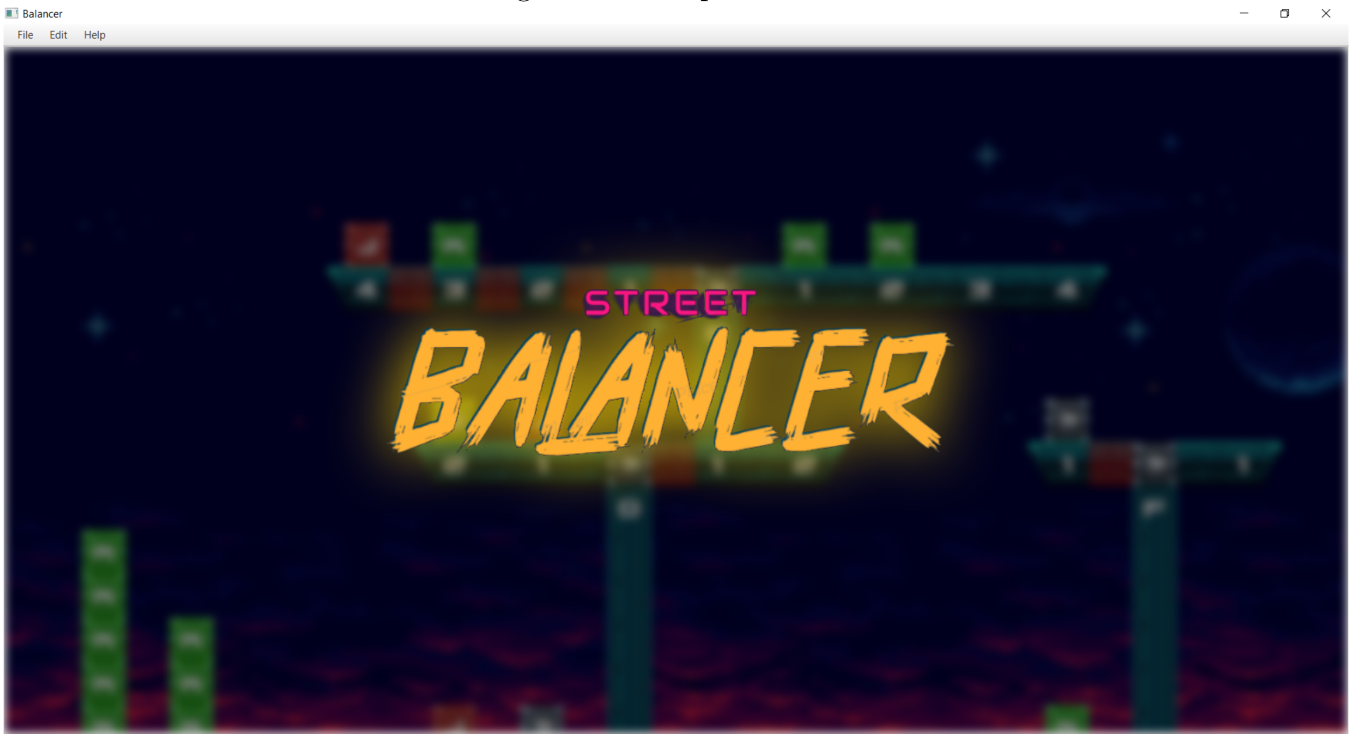
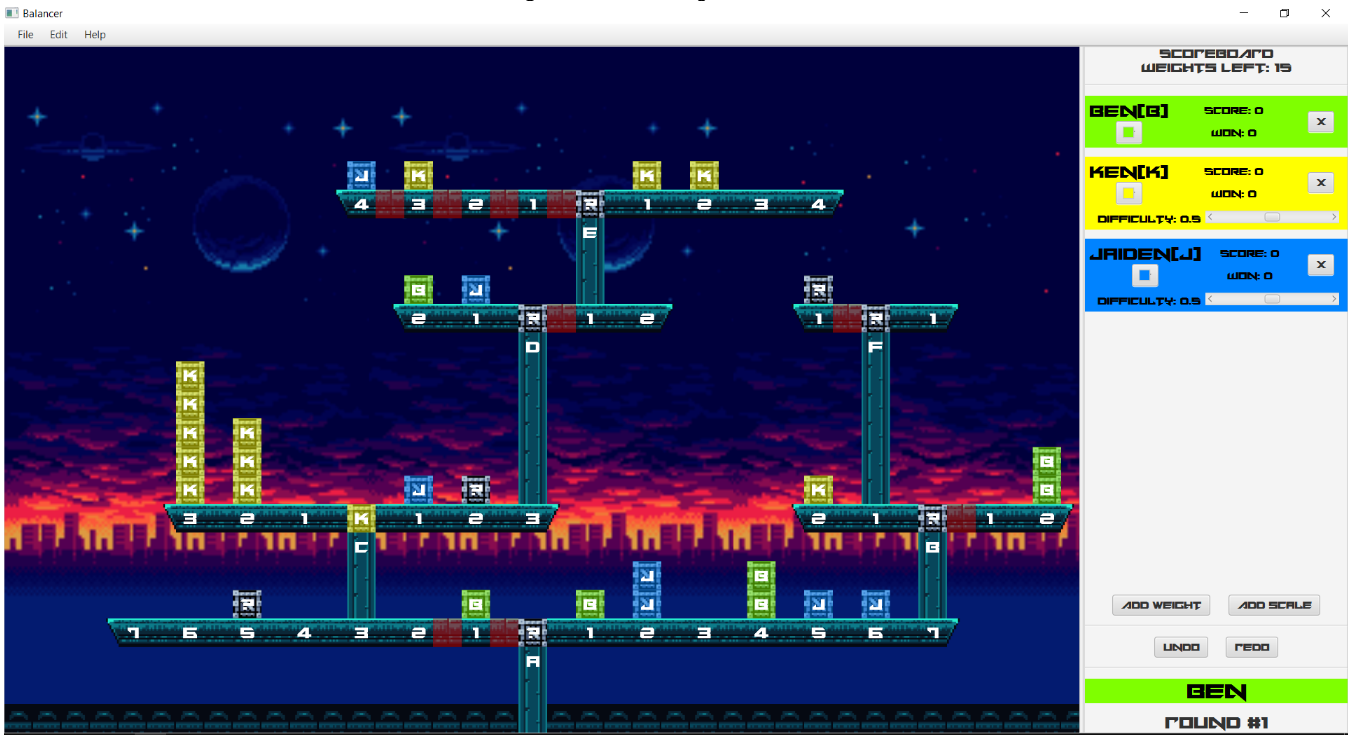


Figure 2: The game screen



The rules and additional informations on the game can be accessed via the "Help" menu in the menu bar.

The source code is also available by clicking the "Help" > "Github" and "Help" > "Gitlab", which will open the repository using the default browser.

## 2.2 Console-based

The program can be run on the command line with no arguments to start a new game or path to the save file to resume. A sample game are shown below:

```
============ ROUND  1 ============

-----------------------------------------
| Ben   (B, 0) :    56 points (human)   |
| Ken   (K, 0) :   248 points (  bot)   |
| Jaiden (J, 0) :    63 points (  bot)  |
|_______________________________________|




       <5=4=3=2=1=?=1=2=3=4=5>
                  g
                  *
            J K   *   K K
           <4=3=2=1=?=1=2=3=4>
                    e
              ?     *
              B J   *       ?
             <2=1=?=1=2>   <1=?=1>
                  d           f
      K           *           *
      K           *           *
      K K         *           *
      K K         *         ? *     B
      K K     J ? *         K *   ? B
     <3=2=1=K=1=2=3>       <2=1=?=1=2>
            c                   b
            *       ? J   B     *
    ?   ?   *   B   B J   B J J *
   <7=6=5=4=3=2=1=?=1=2=3=4=5=6=7>
                  a
                  *
XXXXXXXXXXXXXXXXXX*XXXXXXXXXXXXXXXXXXXXXX
>>>>>>>>> BEN   TURN <<<<<<<<<
Which scale ? (d,b,g,e,f,c,a): a
Position ? [-7,7]: 1
```

### 2.2.1 New Game

When a new game starts, the game will prompt the user to enter basic information about the players. The first round then starts and each player will in turn be prompted for information on where his/her weight should be placed. The current scoreboard, the state of the game will also be shown to for players to make their decision.

When a round finished, the winner of that round will be announced. At the end of the game, the final winner will be announced instead.

### 2.2.2 Load Game

When the save file is loaded successfully, the game will continue from the state when it is saved. If an error occurred, the line causing the failure will be printed.

## 3 Program Structure

The program is split into packages, categorized according to their functionalities and roles in the program. The top most package is *balancer*, containing the class **Game**, **FileManager** and **State**. **Game** stores basic default settings of the game such as the number of weights per round and the default difficulty of the bot. It also acts as a mounting point for **State**, which is different for each game and thus, change whenever the user save or load. **FileManager** handles all operations regarding saving and loading **State**. **State** models the game's state, which is any relevant information that changes during the game, and thus unique for each game. The other subpackages of *balancer* include:

- *balancer.grid:* contains the class **Grid** and **Coord**. **Grid** model a grid of character, an abstract representation of the game state. Each character can either represent a weight of a player, a part of the scale or an empty space. In other terms, grid are just a collection of tiles to display the game.The **Coord** class models a simple 2D coordinate or point, with basic arithmetic such as adding and subtracting.
- *balancer.utils* contains the objects **Constants, Helpers, Prompts** and **Exceptions**. **Constants** and **Helpers** are quite self-explanatory, they store different constants and helper functions. **Prompts** stores wrappers of the default scalafx's *Alert* class. They ranges from informing the users, asking for user inputs and open the file-chooser dialog to save the game.
- *balancer.gui* contains wrappers of scalafx's components and **MainGUI**, which is the entry point of the application. **MainPane** is a wrapper class of scalafx's SplitPane, modeling the main area of the game. **TopMenuBar** extends the scalafx's MenuBar, modeling the app's menu bar, while **GameCanvas** extends the default scalafx's ScrollPane, allowing for panning and zooming. A canvas is also attached and used to display the game's grid. **InfoPane** is the left pane that is in charge of display other relevant stats/information of the game, such as the score of each player and the number of weights left in the round.
- *balacner.objects* contains objects that model different aspect of the game's mechanics. The relationships between them are shown in Figure 4. **Command** models a command or a move

Figure 3: The balancer package structure

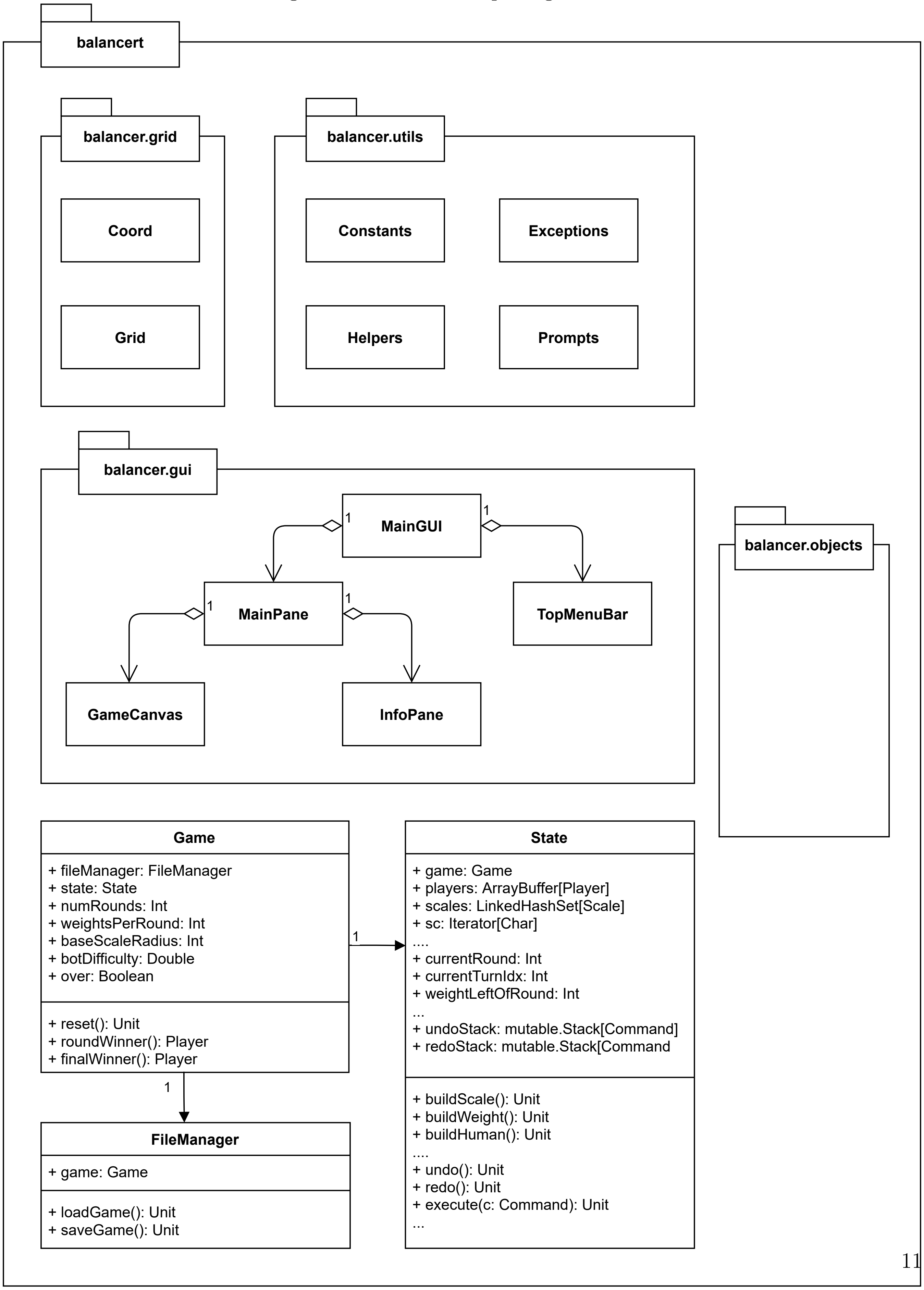
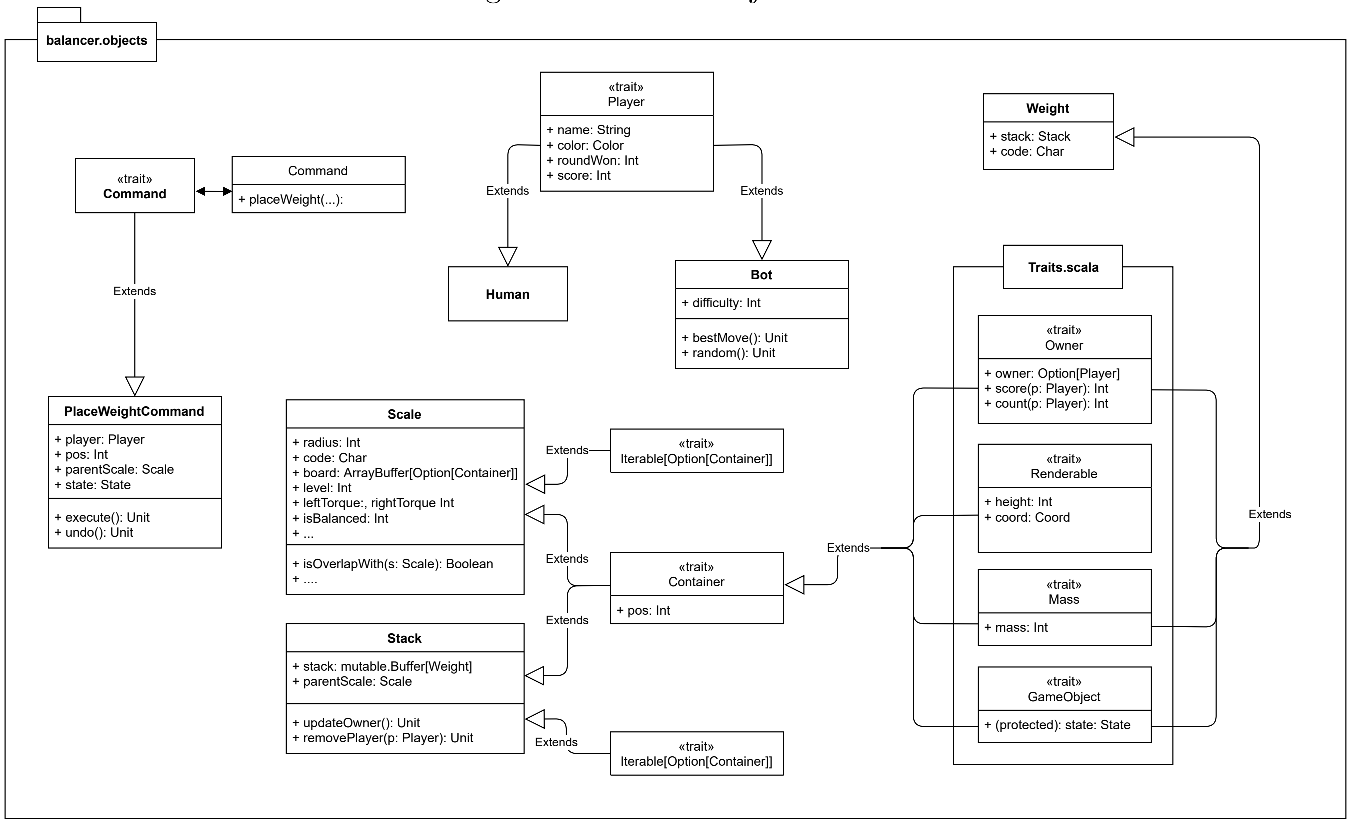


Figure 4: balancer.objects UML



from a player. Here, there is only one the place weight command is implemented. **Scale** and **Stack** models the in-game scale and stack of weights. **Traits.scala** stores share traits between **Scale**, **Stack** and **Weight**. **Stack** and **Scale** also implements the methods of the **Iterable** trait, allowing to be looped over and its content accessed by index, similar to scala's **Seq**. **Player** models a generic player, with a unique name, color and basic stats such as the number of round won and score. **Human** and **Bot** extend from **Player**, modeling a human player and a bot player respectively.

# 4 Algorithms

## 4.1 Attributes

Due to the recursive nature of the game (scale on scale on scale), most methods and attributes are either call other recursive methods or recursively defined themselves:

- scale's *mass* = the sum of its stack's *mass*
- stack's *mass* = the sum of its weight's *mass*
- scale's *level* = the parentScale's height + 1 (0 if at the bottom)

- *imbalance*:

$$\text{Imbalance} = \sum_{i=-r}^{r} i \, m_i$$

  where $r$ is the radius of the scale, $m_i$ is the mass of the object at distance $|i|$ from the center. Negative value of $i$ indicates the object is on the left arm of the scale.

- *score*: player $p$'s *score* on an object $o$ is a piecewise function conditioned on $o$ and $p$:

$$s(o,p) = \begin{cases} s(o,p) = \sum_{i=-r}^{r} |i| \; s(o_i,p), & \text{if } o \text{ is a scale} \\ s(o,p) = \sum_{w \in S} s(w,p), & \text{if } o \text{ is a stack, with weights } S \\ s(o,p) = 1 & \text{if } o \text{ is a weight owned by } p \\ s(o,p) = 0 & \text{if } o \text{ is a weight not owned by } p \end{cases}$$

  where $o_i$ is the object at distance $|i|$ from the center the scale. Negative $i$ indicates the object is on the left arm.

## 4.2 Intelligent bot

### 4.2.1 Description

- **randomized:** The randomized bot simply searches for all moves and filters out moves that would tip a scale, then randomly pick a move.
- **semi-greedy:** The semi-greedy bot also searches for all moves and filters out moves that would tip a scale, but then pick the move that yields the most point for the bot instead.
- **mixed:** a random number generator is used to determine which algorithm to use among the above twos.

The **randomized** bot can check if a move will result in any scale flipped by "faking" the move: place the weight, recursively check if any scale is flipped then revert the move. Similarly, the **semi-greedy** bot can implement a similar method and calculate the score before reverting back, thus finding the move that yield the highest score.

Computing time is nearly instant for most game with only human players, which should be at most 5-6 scales per round.

### 4.2.2 Limitations

Since the bot will never tip a scale, human players can exploit this characteristic of the bot to their advantages and flip a scale deliberately to come out on top. A true greedy bot will also takes moves that will tip a scale into consideration.

## 4.3 Randomized Generation

**Avoiding collision when generate scale**: A simple brute force algorithm is implemented to place random scale without them overlapping each other: finding all possible empty positions, randomly pick a position, "fake" placing the scale and check if collide with any scale of the same level, if it is, revert the placement.

**Weight generation:** A similar algorithm to the **random-bot** algorithm is used: Search for all possible moves that wouldn't tip any scale if played, then randomly execute one of those move.

## 5 Data Structures

**Grid** is implemented as a 2D **Array** of character (**Char**) due to it performance (constant access and insertion time). Since **Grid** will be modified frequently (whenever there is a change in the game state), quick access, insertion and initialization is crucial for performance. Since most of the grid entries are empty, the grid is *sparse*, and some optimization can be done. Other data structures such as HashMap and R-Tree was also considered for more efficient memory usage of sparse matrix, however since the dimension of our grid is not too large ($< 10000$), the memory saved/performance trade offs due to overheads might not be desirable. Implementing a grid as a 2D **Array** also help in development due to its familiarity.

**State** also contains a cached collection of references to the scales, since traversing the scale recursively might not be efficient for some operations. The scales collection is implemented as scala mutable collection **LinkedHashSet**, since we need a mutable collection of unique elements with constant access time and good insertion time. **LinkedHashSet** is only used during development, however, since the elements are ordered (scale inserted first will be accessed first), which will help during debugging and testing. During deployment, the default **HashSet** will be used instead to reduce overheads (linking). The program is designed to better handle large amount of scales. However, it is known that the average number of scales might not exceed 10 in most game with human players, thus **ArrayBuffer** might be a better alternative. **State** will also stores a collection of **Player** which be implemented as scala's **ArrayBuffer**.

**Stack** is implemented as a **Buffer**, since it provides much of the functionality of a stack and easy to work with. The *board* of **Scale** will be modeled as a immutable **Array** since its length is constant i.e the scale can not shrink or grow in length. Raw access speed and low overhead is also one reason to choose the default **Array** over **Buffer** for *board*.

## 6 File and Internet Access

The state of the game is saved in a semi-human-readable format:

```
BALANCER 1.0 SAVE FILE
# Setting
WeightPerRound: 15
NumberOfRound: 5
BotDifficulty: 0.5
```

```
# Meta
Human: Ben
Bot: Ken,Jaiden
Round: 1
Turn: Ben
# Scale
_,0,7,a : -5,? | -1,B | 1,B | 2,J,J | 4,B,B | 5,J | 6,J
a,7,2,b : -2,K | 2,B,B
a,-3,3,c : -3,K,K,K,K,K | -2,K,K,K | 1,J | 2,?
c,3,2,d : -2,B | -1,J
d,1,4,e : -4,J | -3,K | 1,K | 2,K
b,-1,1,f : -1,?
END
```

- The first line of the file is of the form:

  BALANCER (VERSION) SAVE FILE

- The file is split into blocks. Each block of data has a header and a body. A header is identified by having "#" prepended. The header marks the start of a block.
- Each row in the body contains information about the game and is of the form:

  IDENTIFIER : DATA

- The IDENTIFIER and DATA is different for different blocks. There are 3 types of block: *Setting, Meta* and *Scale*.
- The *Setting* block contains different configurations of the saved game.
- The *Meta* block contains metadata of the saved game.
- The *Scale* block contains the position of each scales and weights when the game is saved. The IDENTIFIER is the scale's position and the DATA is the position of each weight on the scale:
- For example, this line in the *Scale* block:

  ```
  a,7,2,b : -2,K | 2,B,B
  ```

  from left to right indicates: the parent scale code ("a"), the position on the parent scale (on the right arm, distance 7 from the center) and the radius (= 2)
- After the colon are the weights' position on the scale, it must be listed from left to right (thus the indices must be from smallest to biggest).
  - (-2,K) indicates on the left arm, distance 2 from the center, there is a weight belong to Ken (K).

  - (2,B,B) indicates on the right arm, distance 2 from the center, there are 2 weights stacked on each other, both belong to Ben (B). The bottom weight is on the right.

- The END keyword represents the end of the saved file. Any line after this keyword is not interpreted by the parser.

- *Note: for the bottom most scale, the parent scale code is "_ ", and the position is* 0. *"?" indicates this is a wild weight or scale, i.e, with no owner. Any excess white space will be ignored by the parser.*

## 7 Testing

The program is tested manually by typing onto the console (for text-only version) or interacting with the graphical UI directly (in a graphical window). These tests include:

- The weight is placed at the correct position according to the user mouse position.
- Panning and zooming work correctly.
- Information/Status of the game such as the score, the round number and the number of weights are displayed correctly.
- **GameCanvas** accurately displays the current state of the game and update when a new weight is placed.

There are automated tools to system test graphical user interfaces. However, running the game directly is the most straightforward one. Different save files are created and purposely designed to test different aspects of the game, specifically, accurate balance-checking of the scales, bot algorithms and random weight, scale generation.

Important "core" methods and attributes that are used frequently through out the program are unit-tested. Example of these "core" methods are the *buildWeight()* and *buildScale()* of **State**. They have multiple test cases designed specifically to test there accuracy. Some tests also aim to test how "solid" these methods are by presenting edge cases and invalid inputs.

Beside "core" methods, algorithms is also tested frequently for accuracy, i.e they do what they are supposed to do. The two algorithms that will be tested extensively is the **semi-greedy** bot's algorithm and the rendering algorithm for update the game's grid.

Only some methods and algorithms passed the tests during their first implementation. Most of them failed and reveal many obscure bugs and faulty logic in the code. The semi-greedy bot algorithm is most tested and also failed the most. Testing really helped speeding the debugging process.

The loading and saving of game file is also tested intensively. There might still be bugs in the implementation since there are a lot of edge cases for human-readable/modifiable save file.

# 8 Known bugs and missing features

## 8.1 Bugs and Temporary Fixes

A possible area where bugs can occur is during the loading and saving of the game. The custom parsing function *loadGame()* is done rather "hacky" and unidiomatic, which may contains obscure bugs that hard to identify. However, a general try-catch safeguard is implemented to at least catch these errors and notify the users.

Another possible bug that might occur is when there are players with similar first letter name (e.g Jack and Jason, Mike and Mary...). Since the *weightCode* is the first letter of the its owner's name, their weights will be represented by the same **Char** on the grid and considered as belong to the same player. This is likely to cause many weird behaviors in the game. Thus, to prevent this, the game will throw an error and notify the player if such naming collision happen.

There are also extreme game state that can be achieve if users want to purposely breaking the game, such as adding infinite number of scales or players. There is a max-depth limit for recursion in Java/Scala, thus, exceeding this limit will cause the program to throw an error when executing/calculating any recursive methods/attributes. The number of scales might also exceed the ASCII limit for lowercase letter ($z = 122$) and thus, assign weird characters to new scales. However, both of these problem can be fixed by limiting the number of scales to 26.

On the opposite end, the game might also have 0 player or only consists of bots. This is also likely to cause an error and thus, at least one human player are required for the game to start.

## 8.2 Missing Features

*Multiplayer supports:* The original plan includes implementing a basic multiplayer version of the game using *Play framework*'s Websocket. However, due to the time constraint, overlapping schedules and heavy workload from other courses, it was not realized.

*Setting Menu:* It was also planned, in the first version of the technical plan, to implement a setting menu, which would allow users to modify the settings directly in the GUI. However, it has not been implemented in time of submission.

*Full Undo:* Undoing is not fully implemented for scale. When a scale is flipped, no undoing can be made to reinstate that scale.

# 9 Best sides and Weaknesses

## 9.1 Best sides

1. The program is quite organized and can be easily to expanded/build upon in the future. Each packages is separated according to its roles in the program: **Helpers** contains helper functions and **Constants** stores different constants that are used. GUI elements are separated and explicitly named after their roles: **TopMenuBar**, **MainPane**, **GameCanvas**, ...

2. Undoing and Redoing are not planned in the original plan but are a result of refactoring

to the class **Command**. While there are only one command (**PlaceWeightCommand**) for now, other commands such as **AddingPlayer**, **RemovePlayer** , **AddScale** , **RemoveScale** . . . can easily be added using the same framework.

The program can be divided into different areas/sections, grouped by its role/concept that needed to be built, tested and documented. Here we also introduce the concept of **"core"** classes and objects, parts of the program that many other objects depend on them. In this project, they are the main Game class, Weigh-Scale-Stack classes and the Player class. These represents the fundamental ideas of the game and with them, gameplay is achieved. Other classes will be build on top of them to either adding additional functionality like save/load file or display them to the user either in a graphical UI or on the console.

Different areas/sections will require different amount of time to go through the 3 phase mentioned above (built, test, document), with the core classes and the UI being the most demanding ones.

- The first two weeks (**22.2** - 7.3) will be dedicated to build and test different functionalities of these core classes. The bot algorithm may or may not be implemented in this period since it is not immediately essential and can be separated.
- The week after (8.3 - 14.3) will be dedicated to implement and test the save/load file functionality of the game. This will be important for testing the UI and the bot algorithm since generating test case will be much easier. Noted that the first interim report is due on 17.3.
- The next 2.5 weeks (15.3 - 31.3) will be dedicated to build and test the UI for the game, on the console and graphically in a window. Noted: the second interim report is due on 31.3.
- The next two weeks (1.4 - 14.4) will be dedicated to system testing of the whole program, debugging, tuning, optimizing for performance and adding additional features where needed. The final interim report is also due at the end of this period, on 14.4.
- The final two weeks (14.4 - **28.4**) will be dedicated to writing the required documentation and act as a "buffer zone" if more time is needed to complete certain features.

## References

[1] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[2] Scala collection. https://docs.scala-lang.org/overviews/collections-2.13/introduction.html.