Assignment 4

Temple Run

Group 5

TI2206 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology.

L. van Aanholt levi.vaan@gmail.com 4084101
M. Wolting matthijswolting@gmail.com 4356063
M.E. Kuipers marcelkuipers1996@gmail.com 4389042
N.E. Hullegien nilshullegien@gmail.com 4389069
H.M. Yeh H.M.Yeh@student.tudelft.nl 4386027

Exercise 1 Scene2D

General explanation
Scene2d can be seen as a play where there is a stage with actors interacting. The stage is the root of the hierarchy of actors. All the actors that will be displayed on the screen will be added to the stage. The stage will then make the actors draw themselves. These actors can be added as groups or individually to the stage.
Also actors can contain other actors. That way the stage will make the actor draw itself and in turn the actor will make his children draw themselves. All the rotations and scalings applied to the parent will also be applied to the children.

Advantages scene2D
Scene2d is better for implementing the User Interface of the game, otherwise the User Interface will need to be positioned with the right coordinates on the screen. Also the User Interface also needs to move inside the world, because of the moving camera in the game. While in the case of scene2d an UI will be built on a stage that doesn't move and stays on the same place of the screen.
Further it helps when objects are grouped together, because without scene2d the grouped objects will need to be sized individually. In scene2d only the parent will need to be sized and all the children will be sized accordingly.
Also the freetype font generator is used for the main menu and for the game over menu. This gives more freedom in what kind of fonts can be used as well as the scale of the fonts. Otherwise scaling the regular bitmapfonts could result in oversampling or undersampling..

Implemented features
The features implemented in the main game is an UI with a highscore table and a score table. The highscore table is for the player to see the highest score achieved. The score table is for the player to see the score that player has reached at that moment in the game.

Apart from that, the main menu was also reimplemented in a table-based way, which makes it more flexible to scaling and easier to add buttons to, and the game over menu that is laid over the main game if the player dies was split out into a separate class to improve separation of features.

Exercise 2 Software Metrics

1.
The analysis is located in the *incode* map which is located inside of the *documentation* map

2.
Our project has a total of 3 design flaws according to the analysis.

1: Our *highScoreScreen* class is considered *feature envious*.
The reason why this class is considered feature envy is that it uses a nested for loop to find scores with the correct rank. This was done because of our design choice to keep the *ScoreContainer* class simple (not add many methods). The *ScoreContainer* class doesn't have a *getRank()* method which would return the *ScoreItem* object with the corresponding rank. Because of this, other classes have to look for ways to implement this method themselves, resulting in possible duplication and confusion.

We fixed this design flaw by moving the sorting of the ScoreContainer list to the ScoreContainer class, instead of keeping it in our highScoreScreen class.

2: Our ScoreItemClass is considered a *data class.*
The reason why this class is considered a data class is that it gives data to too many classes, namely the HighScoreScreen, ScoreContainer and ScoreWriter class. It serves as a container to hold a set of values of different types. These values always exist in a group and are all used by multiple classes.

We decided not to delete this class from our project, but instead make only a small correction. Our implementation uses a scoreItem object as a type, like e.g. an int. The idea is that you can set the values once (you can't modify them). If you want to change something you will need to replace the whole object by creating a new instance.

One thing that violated this idea in our previous implementation wat that the *scoreItem* class contained a *setRank()* method. This was convenient when score had to be shifted (rank 2 becomes 3, rank 3 becomes 4). We have deleted this method and impelemented the *scoreContainer* class to create new objects instead of altering previous ones.

The *scoreItem* class we now have is still recognized as a Data Class by inCOde, but isn't like the Data Class described during the lecture. The class doesn't 'control' the data, it only provides a convenient to store four values of different types. A *scoreItem* object is only associated with one group of values (rank,score,name,date), which cannot be altered after assignment.

Our GameSlice is considered a *data class*.

The reason why this class is considered a data class is that it gives data to too many classes, namely the Director and the Spawner class. It serves as a container to hold a set of values of different types. These values always exist in a group and are all used by multiple classes.

In the old implementation the director class had multiple behaviours, it generated the gameslice and randomized the representation at it's instantiation, it also managed when objects should appear. In practice the director contained all the logic to create a new GameSlice with all the possible configurations it can have, and the gameslice was just a class void of behaviour that contained the data set by the director. The spawner class had methods to add procedural elements used by the director and some methods for the director to manage where it should put procedural elements.
The new GameSlice implementation has been restructured in the following way. The director has the only responsibility of checking whether a new gameslice should be added given the status of the game. It adds a GameSlice through GameSliceCasting class, that maps the current status of the game (mainly the current score) with the GameSlice that would be appropriate to add. There is a new class GameSliceQueue, that contains a set number of gameslices. It handles gameslices given by the director and pops gameslices when a new one has to come in. It also gives worldobjects from the gameslices it has back. GameSlice is a abstract class, with methods to retrieve worldobjects and manage observes (through update()) what it's status is compared to the game. GameSlice can be extended to create procedural level elements. It handles it's own randomized representation at it's initialisation.

Because Gameslice creates the WorldObjects it has through randomization within the class, spawner class can be deleted. Spawning objects and randomely placing them in a GameSlice is actually something the Gameslice should do itself, because these objects are at the core of the GameSlice behaviour, namely, the representation of a procedural level in the game. This new implementation is Open for extension and closed for modification. A new procedural level element (GameSlice) can be added by just extending from the abstract class GameSlice.