TI2206
Software Engineering Methods
Group 5
Temple Run
Assignment 3

Marcel Kuipers
Nils Hullegien
Matthijs Wolting
Levi van Aanholt
Hao Ming Yeh

Exercise 1 - 20-Time, Reloaded

The extension we chose to implement for this exercise was a high score system.

Requirements

- The game shall be able to calculate the score of the player according to the distance the player has covered.
- The game shall be able to store multiple high scores of the player in a separate file.
- The game shall be able to read the separate file to show the multiple high scores.
- The game shall be able to rank the high scores of the player.
- The game shall be able to store the date when the player has achieved the associated high score.
- The game shall be able to store the name of the player that has achieved the associated high score.
- The player shall be able to go to the high scores from the main menu.

Scores are stored as *Score Item* objects, which contain the following information: Date, rank, score and the name of the player. The *Score Item* objects are stored in the static class *ScoreContainer*. This class is useful because it allows you to look up scores without having to load in a score file, thus making it faster to perform operations like checking whether a score is the new high score.
When starting up the game all scores are read from a file by the *ScoreReader* class, which uses Java Properties to easily do this. This list of *ScoreItem* objects retrieved from the file is inserted into the *ScoreContainer* class.

Whenever it is game over the user is able to set the name he wants to connect to the score, by filling in a text field. He is also able to see a visual indication when his score is the new high score. When the player starts a new game or goes to the main menu, the score gets inserted into the *ScoreContainer.*

Each time a score is added it also get written to a file by the *ScoreWriter* class, so that the file is always up to date.

CRC cards

ScoreReader	
Responsibilities	**Collaborators**
Read scores from file and insert them into the ScoreContainer	ScoreContainer

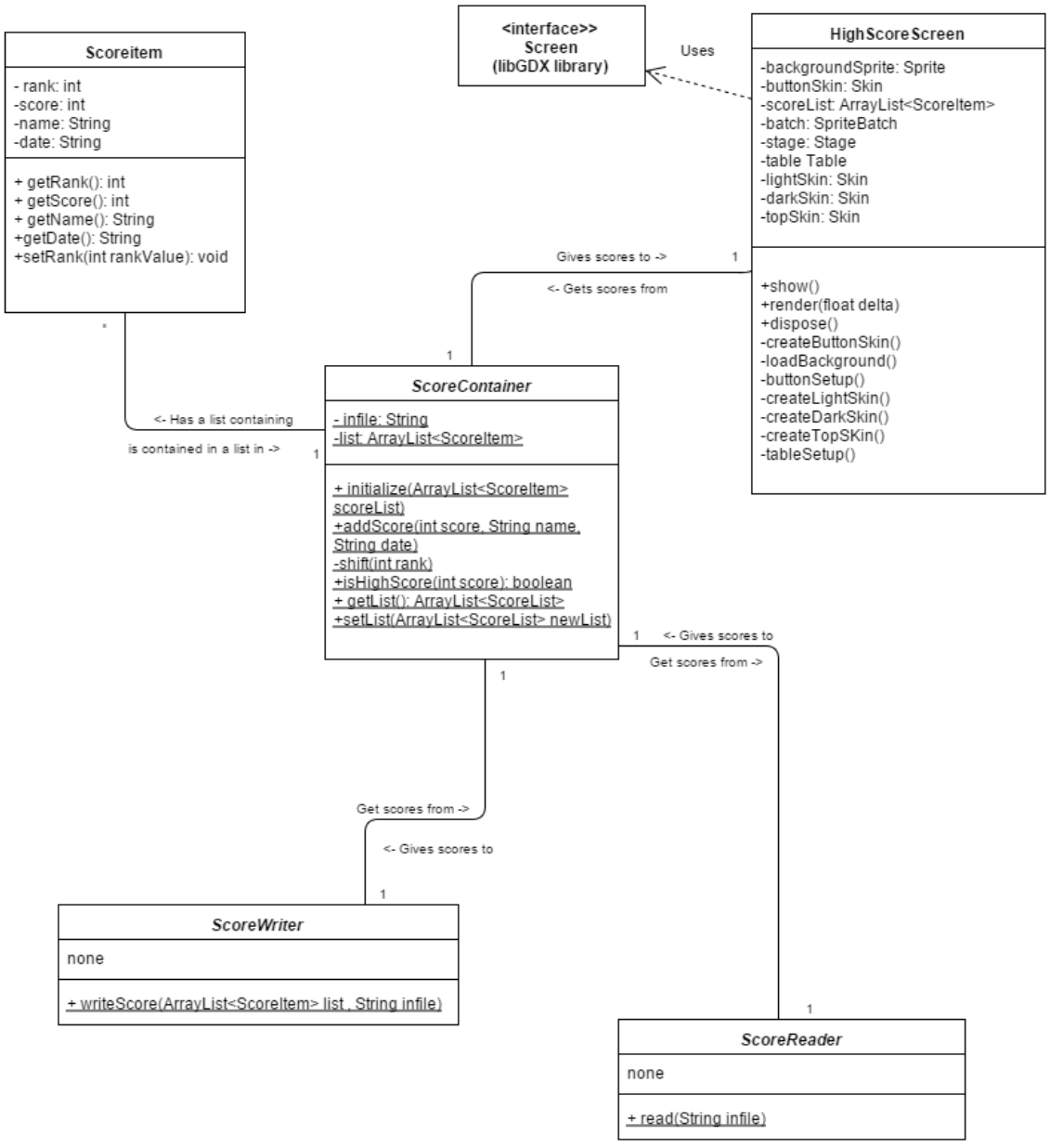
ScoreWriter	
Responsibilities	**Collaborators**
Write scores to file	ScoreContainer

ScoreContainer	
Responsibilities	**Collaborators**
Provide access to classes in game so that the file doesn't have to be read every time. Gets scores from ScoreReader	ScoreReader

ScoreItem	
Responsibilities	**Collaborators**
Contain relevant information about scores, that is: name, rank, date and score.	ScoreContainer

HighScoreScreen	
Responsibilities	**Collaborators**
Make information about high scores visual.	ScoreContainer

Class Diagram

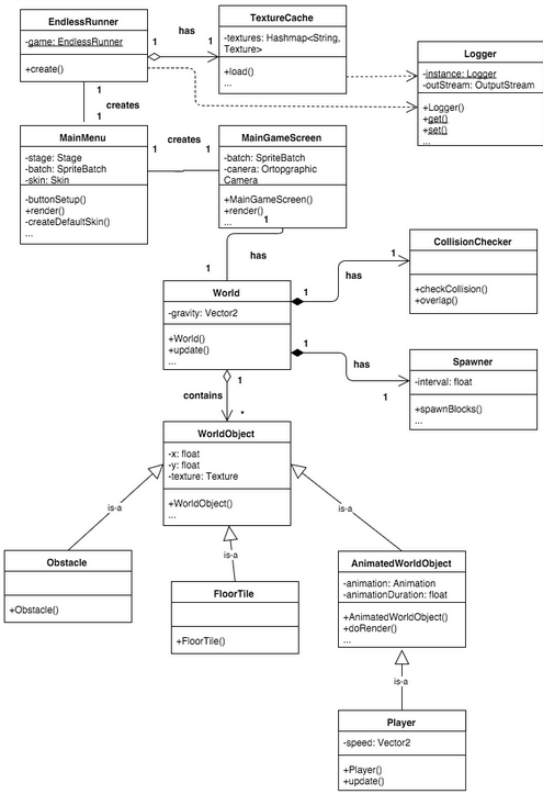


Exercise 2 - Design Patterns

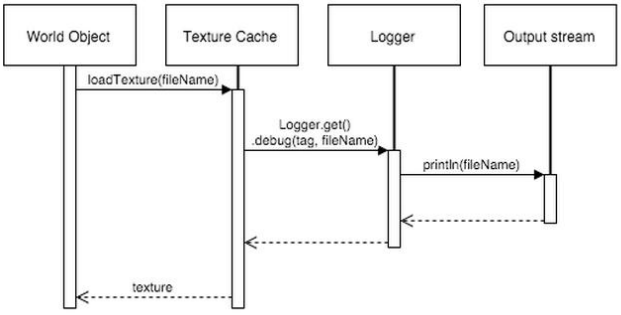
The singleton pattern

Description
We implemented the Singleton pattern through our logging system. This was done because there is no real reason to have multiple loggers in a system, as it is logical to have all logging output be written to the same output stream(s) and configuration of the logging can be handled globally. The pattern is implemented by giving the Logger class a static member variable, instance, that contains the logger instance for the game. It is initialized in a static{} block, which guarantees that a logger is present at the start of the game. The instance can be obtained using the static get() method on Logger, and changed (to alter configuration or used implementation dynamically if needed) using the static set() method on Logger.

Class diagram



Sequence diagram

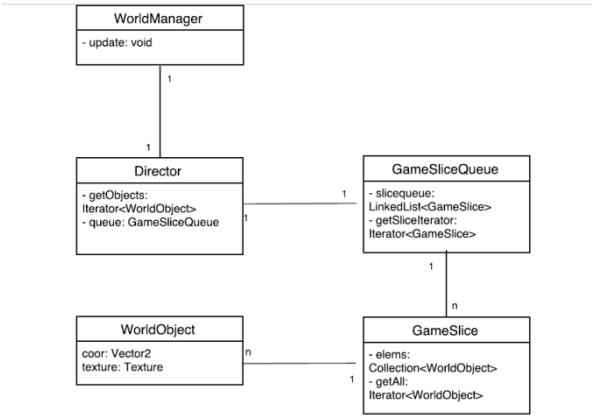


The Iterator pattern

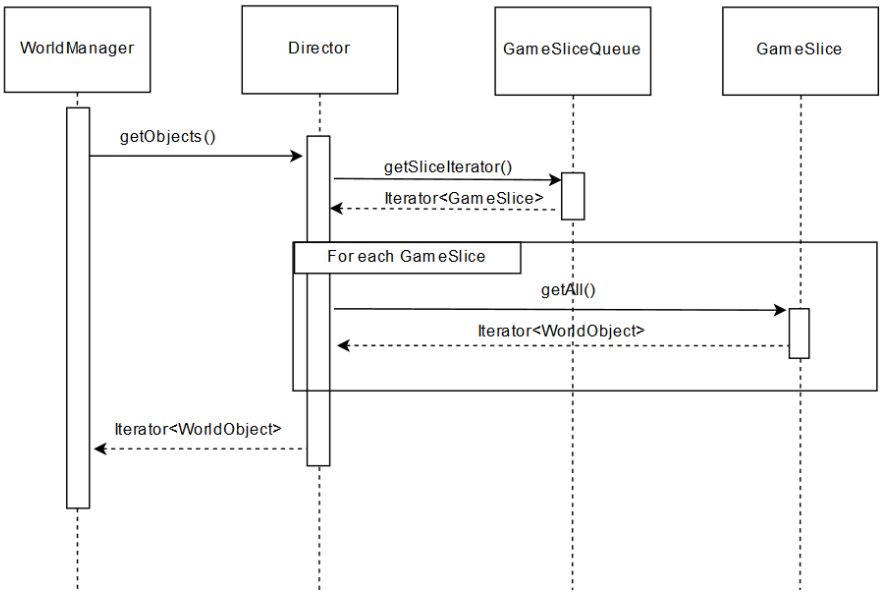
Description*
The iterator pattern is used to handle receiving world Objects from the directorclass for updating them, collision detection or/and rendering. The director handles returning an iterator of WorldObjects and handling the GameSliceQueue, in which the active procedural level elements (called GameSlices) live. The director is the only connection from which the objects in the GameSlices can be gotten, and this is through an iterator.

It is very handy to use an iterator in this case, because the Director can do all the operations to collect the items, iterating over the queue, and for every element in the queue the elements there are iterated etc untill the Director receives all WorldObjects. It then returns an iterator of this list. It returns this iterator without showing it's underneath implementation.

Class Diagram



Sequence Diagram



*note, due to CL server not correctly running the tests for the new GameSlice Queue, the merging of the branch this iterator pattern moved to the next sprint.

Exercise 3 - Software Engineering Economics

Recognizing good and bad practices
Good and bad practices are recognized with the help of software metrics. Analyzing the overall average performance of all projects that will be researched with regard to project size, project cost, and project duration. A good practice is recognized if the cost and duration is better than the average cost and duration for the applicable project size. A bad practice is recognized if the cost and duration is worse than the average cost and duration for the applicable project size.

Visual Basic not interesting
Because there were a low number of projects that did visual basic, namely six projects. Of these six projects five succeeded as being a good practice and the last one ended up being cost over time. The paper also says that that could have happened for the reason that the visual basic skills from the project teams were better than other skills, but it is more likely that the projects were just less complex than the average of the researched projects.

Possible factors for good or bad practices
Test Driven Development: good practice, because starting to make the tests first and then making the code pass the tests helps with finding bugs in the code and when the code passes all the tests then the code is working. That way no not working code will be added to the project and the final build will probably pass.
Bad working conditions: bad practice, because when the working conditions are bad it is harder for the project team to work motivated on the project. The team might in those cases be more prepared to put more redundant code in the final build.
Drastic changing requirements.: bad practice, because with any change in requirements during the implementation period the team might have done unnecessary coding and needs to make something else instead. That extends the duration of the project. This is especially the case when the requirements are changing drastically, because then bigger parts of the code is unnecessary and a lot of new code needs to be written to adhere to the requirements/

Bad practice factors description
Dependencies with other systems: If the project relies on other systems the other systems need to be working and not changing a lot, because if something changes in the other system the own project might have to be adapted to the other system or in the case that it is not working the project will probably not work anymore.
Once -only project: Because these kinds of projects will not be written very well or designed, so the project might have a lot of redundant code that makes the project much more complex than it needs to be. It could also have intensive coupling which makes it worse when the god class is not working.
Many team changes: Because every time a new team member emerges that person needs to be up to speed with what is happening and what he needs to do, which will require some time before he/she can actually do something. For many team changes that will delay the project for a long period of time.