Space Invader

Requirements Specification

A. B[illegible]
T. Westerborg
T. Oomens
N. van der Laan
D. van Tetering

Delft University of Technology

TU Delft

Preface

Preface is yet to be written

A. Breurkes
T. Westerborg
T. Oomens
N. van der Laan
D. van Tetering
Delft, September 2015

Contents

1

Functional Requirements

For the game Space Invader, the requirements concerning functionality and service are stated under the *Functional Requirements*. Within the functional requirements, we identify four categories using the *MoSCoW* model to prioritize the requirements:

1.1. Must have

- The player shall be able to move left and right, but will never be able to leave the board.
- The invaders shall move a fixed amount to left and right.
- On startup, a fixed amount of invaders will be spawned in a grid.
- Invaders and players shall be able to shoot a projectile.
- When a projectile hits a player, the game shall end.
- When a projectile hits an invader, the invader shall disappear.

1.2. Should have

- The player shall be able to (re)start a new game of Space Invader.
- The player shall be able to end the game of Space Invader.
- The game shall show a menu when the game starts.
- The game shall end when the player loses the game or stops.
- The game shall increment the score when the player hits an invader.
- The game shall play a sound effect when the player dies/ hits an invader/ moves. There shall also be background music.
- There shall be motherships among the invaders during certain levels.
- The game shall show the player's game statistics after losing a game.
- The game shall reset the player's score and other game statistics when a game ends.

1.3. Could have

- The game shall have several levels.
- There will be upgrades available (the specific kind of upgrades have yet to be determined).
- The game shall have a multiplayer mode.
- The game shall have a storyline.
- The game shall have increasing difficulty, based on the player's current score.
- The game shall play a music theme when in progress.

1.4. Would have

- The game shall have a switching theme, according to current level and the current score.

2

Non-functional Requirements

Besides the functional requirements and services, design constraints and other *non-functional requirements* need to be included in the requirement specification as well. These requirements indicate the constraints that apply to the system or the development process of the system.

- The first playable version of the game shall be finished within two weeks.
- The game shall be implemented in Java.
- The game shall be designed, implemented and documented in a group of five people.
- The game shall be playable on Windows (7 or higher), Mac OS X (10.8 and higher), and Linux.
- For the iterations after the delivery of the first fully working version, the Scrum methodoly shall be applied.
- The implementation of the game shall have at least 75% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the game and for example do not justexecute the methods involved).

3

Assignment 1

3.1. The Core

Following the Responsibility Driven Design, start from your requirements (without considering your implementation) and derive classes, responsibilities, and collaborations (use CRC cards). Describe each step you make. Compare the result with your actual implementation and discuss any difference (e.g., additional and missing classes).

Candidate classes: From the must haves, we can extract the following classes:

- Player
- Alien
- Level
- Projectile

As the classes Player and Alien are both very similar in alot of ways, these can be subclassed to a superclass, Sprite.

To have a runnable structure, we need to following classes:

- Main
- Game

In our should-haves, we state that the game includes motherships as well, this results in the class Mothership. From the could haves, we can then extract the class Upgrade.

CRC cards:

Class Name	Main
Super Classes	None
Subclasses	None
Purpose: x	Collaborator: x

Following the Responsibility Driven Design, describe the main classes you implemented in your project in terms of responsibilities and collaborations.

The ‘Main’ class is responsible for executing the game. This means it should not perform anything else other than initializing the game. This can include things like add all states to the game container, create a new game object and set all initial settings of the application.

The ‘Game’ class is responsible for updating our application. This class will keep track of all things that change, are added, are deleted, and so on. To do so, game collaborates narrowly together with the other main classes Player and Level.

The ‘Level’ class is responsible for containing all information about the level the user is currently playing. This means Level collaborates with Classes Player and Alien, as these two classes basically define the level.

The ‘Sprite’ class is responsible for containing general information about a sprite. These things should include aspects like size and and location. More specific information about the sprite does not belong in Sprite and should be added to subclasses

The ‘Player’ class is responsible for all interaction with the sprite that is controlled by the player. This means Player keeps track of things like movements and shooting. To do so, collaboration with Game is essential, as Game is responsible for updating the game and thus making sure the Player sprite actually moves. In order to shoot, collaboration with the Projectile class is needed.

The ‘Alien’ class is responsible for the behaviour of the aliens that the user is playing against. This means it contains general information about an alien, such as movements and shooting. Like Player, shooting requires collaboration with the Projectile class

The ‘Projectile’ class is responsible for everything being fired by the Player or an Alien. It is responsible for aspects of the projectile, such as speed and direction. Collaboration with classes Player and Alien necessary, as these make use of the Projectile class.

3.2. UML in Practice

What is the difference between aggregation and composition? Where are composition and aggregation used in your project? Describe the classes and explain how these associations work.

If the relationship between a child and its parent is an aggregation, this means that the child can exist without its parent, if we were to delete the parent. E.g, the relation between a person and his/her home address. If we were to take away the parent from the child, in other words, the person moves to another address, the address still exists.

If the relationship between a child and its parent is a composition, this means that the child cannot exist without its parent. E.g., the relation between a car and its engine. If we were to destroy the car, the engine will be destroyed along with it.

In our code, the relationschip between the parentclass Alien and its children MiniAlien,

MotherShipAlien, SmallAlien and BigAlien is a composition. Since the children all inherit essential parameters and methods from their parent, they cannot exist without it.

The relationship between the parentclass Game and Player, LevelFactory, Level and HighScoreManager is an aggregation. A Game consists of a Player, a LevelFactory, a Level and am HighScoreManager. However, if we were to delete the Game, the Player and all the other attributes still exist.

Is there any parametrized class in your source code? If so, describe which classes, why they are parametrized, and the benefits of the parametrization. If not, describe when and why you should use parametrized classes in your UML diagrams.

There are no parameterized classes in our source code. These classes should be used in your UML code when the source code contains parameterized classes. Parameterized classes are classes which takes another class as a parameter, e.g. ArrayList<Integer>. These classes should be used in our UML diagrams to indicate which classes are parameterized.

Draw the class diagrams for all the hierarchies in your source code. Explain why you created these hierarchies and classify their type (e.g., "Is-a" and "Polymorphism"). Considering the lectures, are there hierarchies that should be removed? Explain and implement any necessary change. Please refer to the end of the document for all the UML diagrams.

3.3. Simple Logging

Extend your implementation of the game to support logging. The game has to log all the actions happened during the game (e.g., player moved Tetris piece from position X to position Y). The logging has to be implemented from scratch without using any existing logging library. Define your requirements and get them approved by your teaching assistant.

3.3.1. Functional Requirements

- For each event, the logger must create a new Log.
- Each log must be formatted to HTML.
- A Log must contain a message.
- A Log should contain a date of writing.
- A Log should contain a level of importance.
- When the game is started, an HTML Header is generated and the Logger will start logging.
- When the game is finished, an HTML Footer is generated and the Logger will stop logging.

3.3.2. Non-Functional Requirements

- The generated HTML file can easily be understood by a user that is not necessarily involved in the development of the Logger tool.
- The Logger tool does not affect the state of the game in any way.
- The Logger tool does not affect the speed of execution of the game in any way.

During the analysis and design phases of this extension use responsibility driven design and UML (push to the repository a single PDF file including all the documents produced)

The logger is a feedback tool. With it, users of the game will receive a log containing all the actions that were logged during play.
When the game is started, an HTML header is generated and written to an HTML file. While the game is active, the logger will keep adding new logs whenever new events occur. When a new event occurs, the logger creates a new log. This log contains a message, the time of writing and a level of importance.
When the log is created, the logger then makes use of an HTML formatter. This HTML formatter parses logs to HTML and adds it to the HTML file. When the game ends, the logger stops logging, generates an HTML footer and adds this to the HTML file.

Candidate classes:

- Logger
- Log
- HTML Formatter

CRC cards:

Class Name	Logger
Superclasses	None
Subclasses	None
Purpose: create Logs	Collaborator: Log

Class Name	Log
Superclasses	None
Subclasses	None
Purpose: contain Messages	Collaborator: Logger
Purpose: contain Date of Writing	Collaborator: none
Purpose: contain Importance of Message	Collaborator: Logger

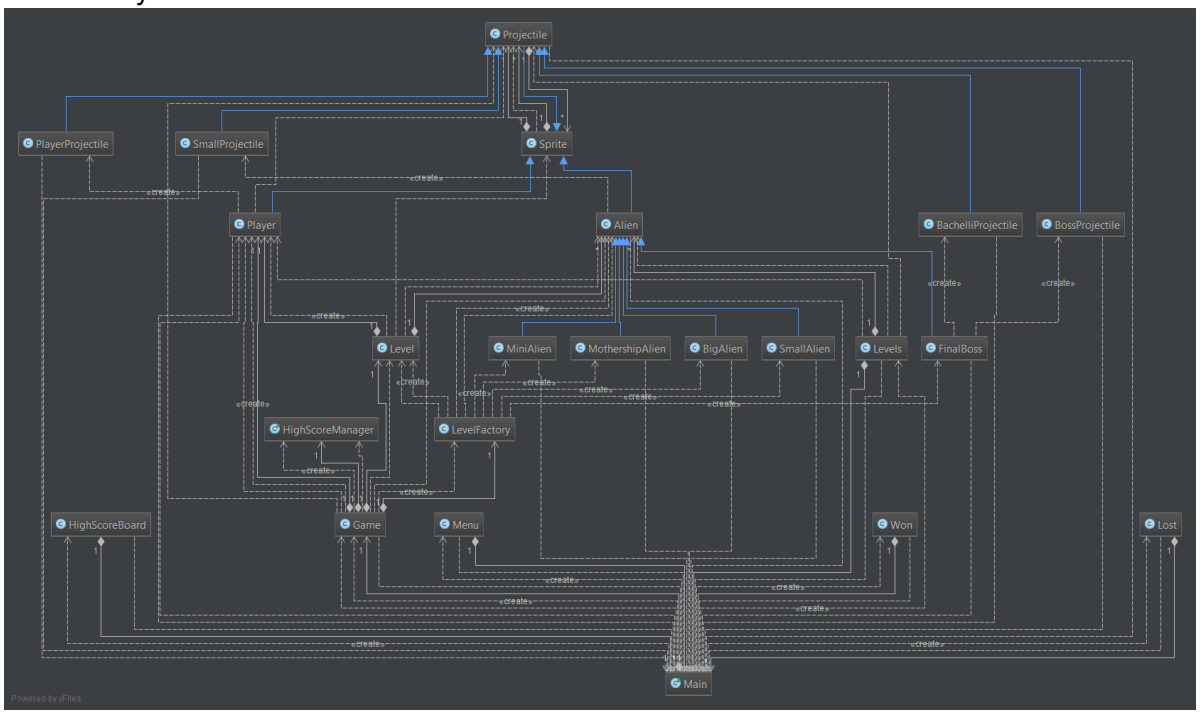
Class Name	HTML Formatter
Super Classes	None
Subclasses	None
Purpose: format Logs to HTML	Collaborator: Log, Logger

Please refer to the end of the document for all the UML diagrams.

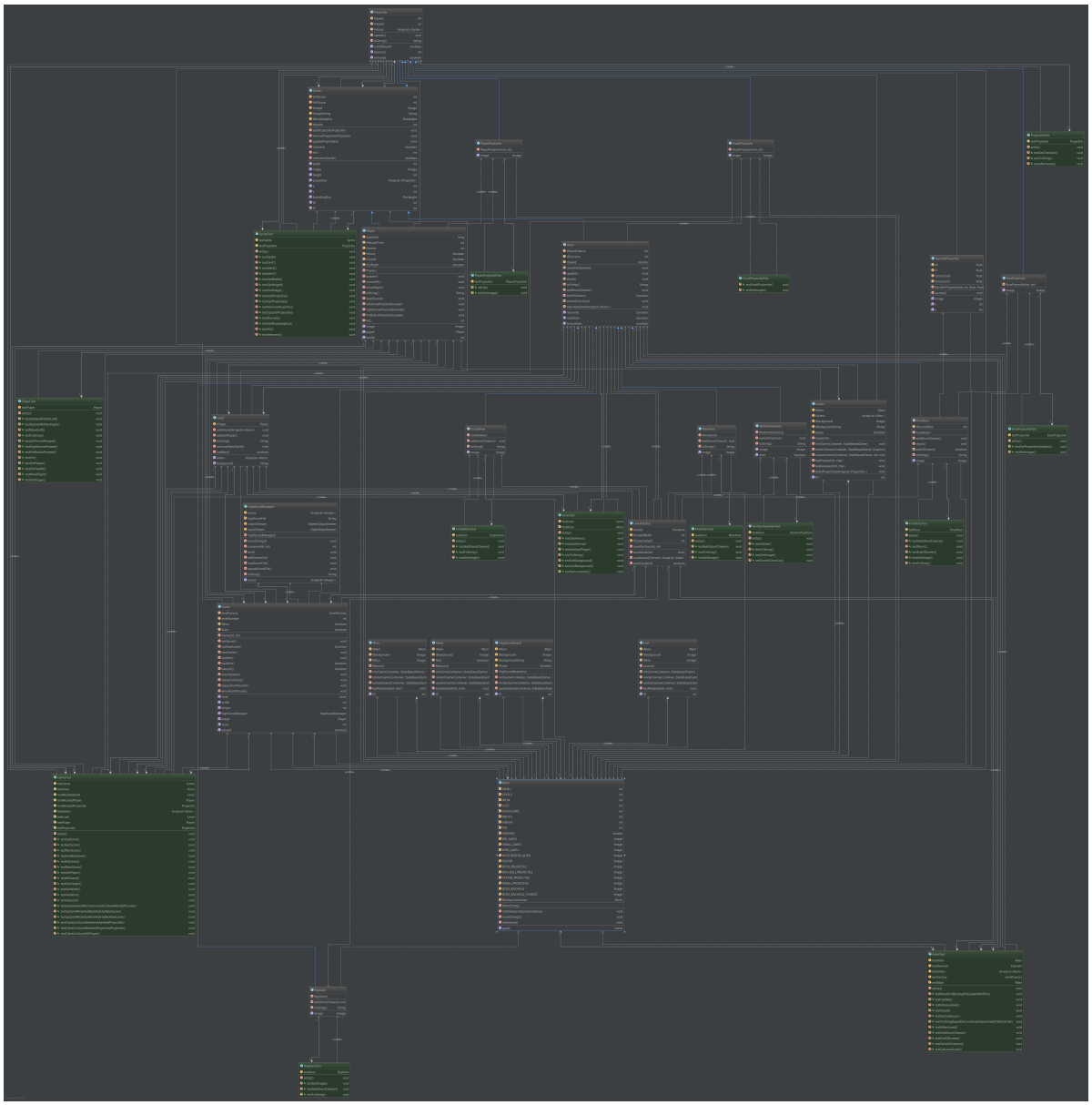
4

UML

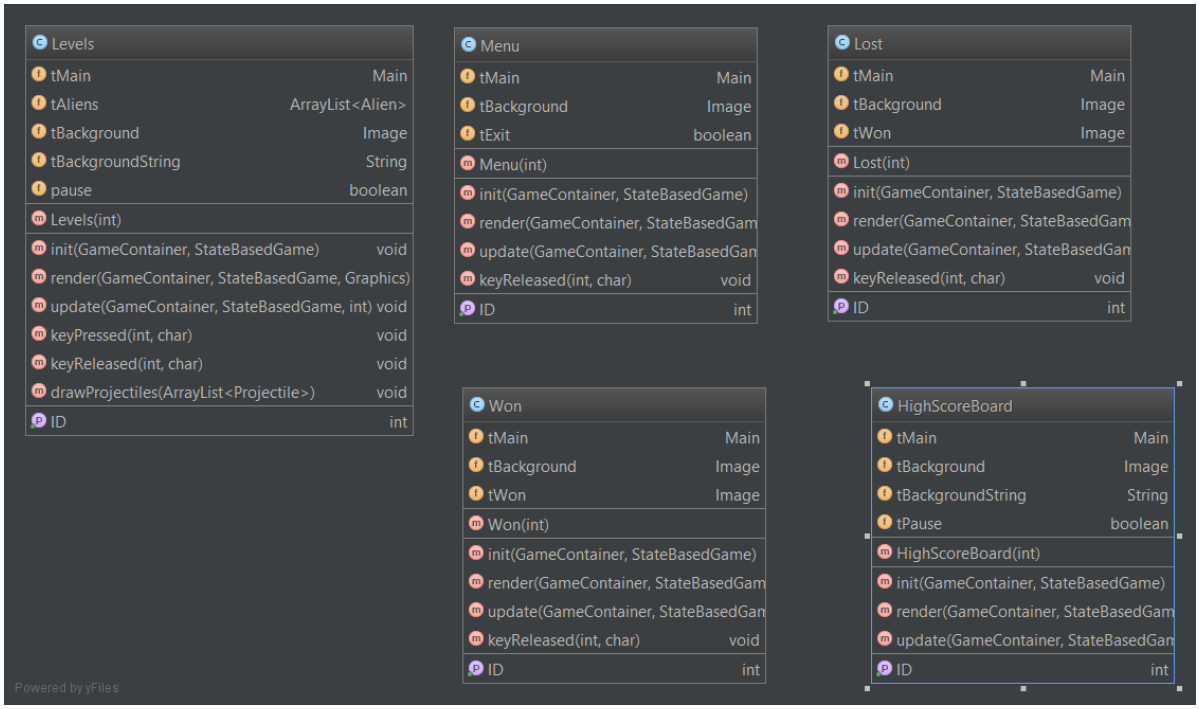
Please note that all the UML files are included in the source code, if a closer look is needed. The .uml files can be found in the directory src/UML. The .png files (the images) can be found in the directory deliverables.



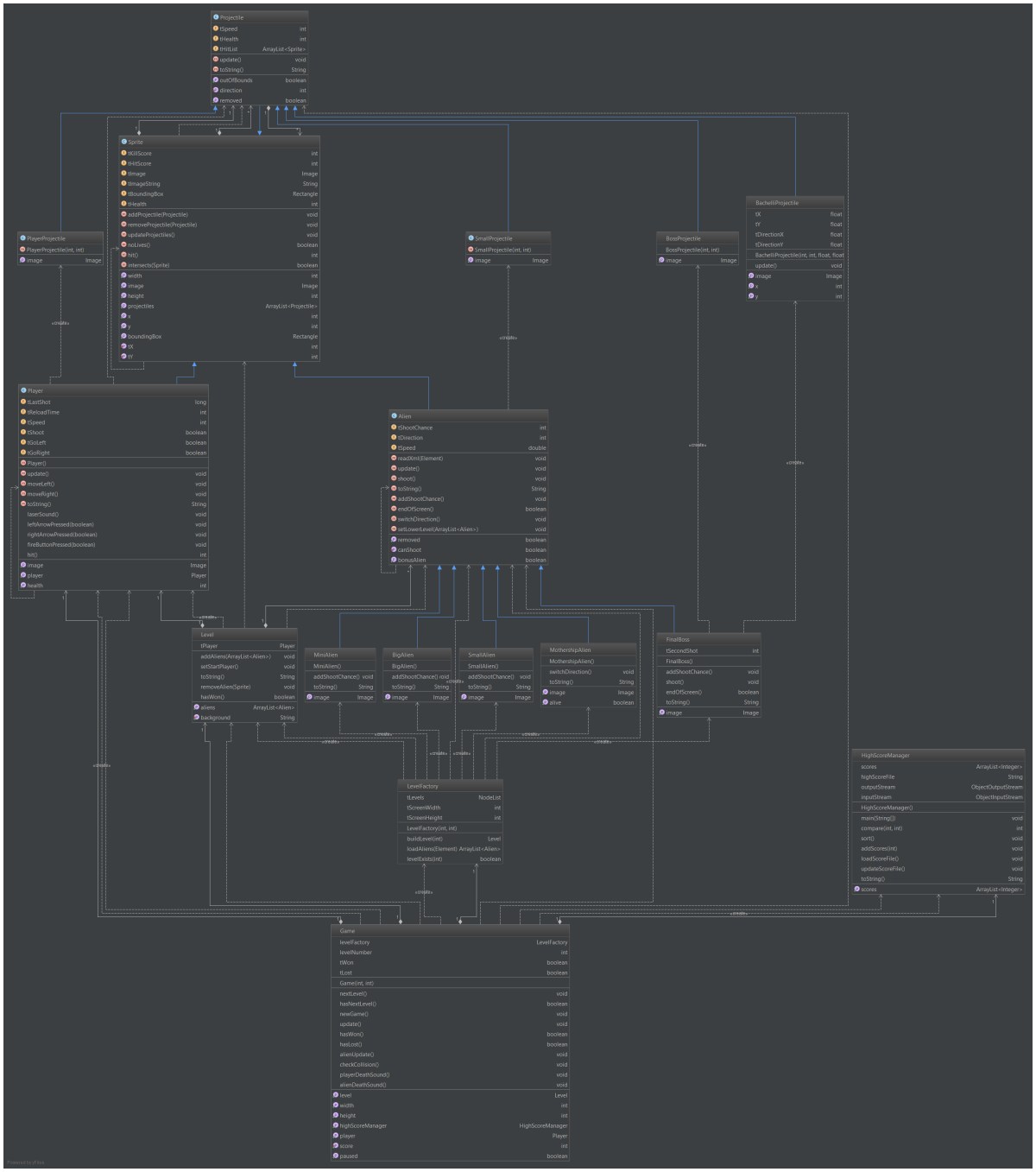
Above, there's an overall simplified UML of our code, with this UML an overview of the structure of the game is visible.



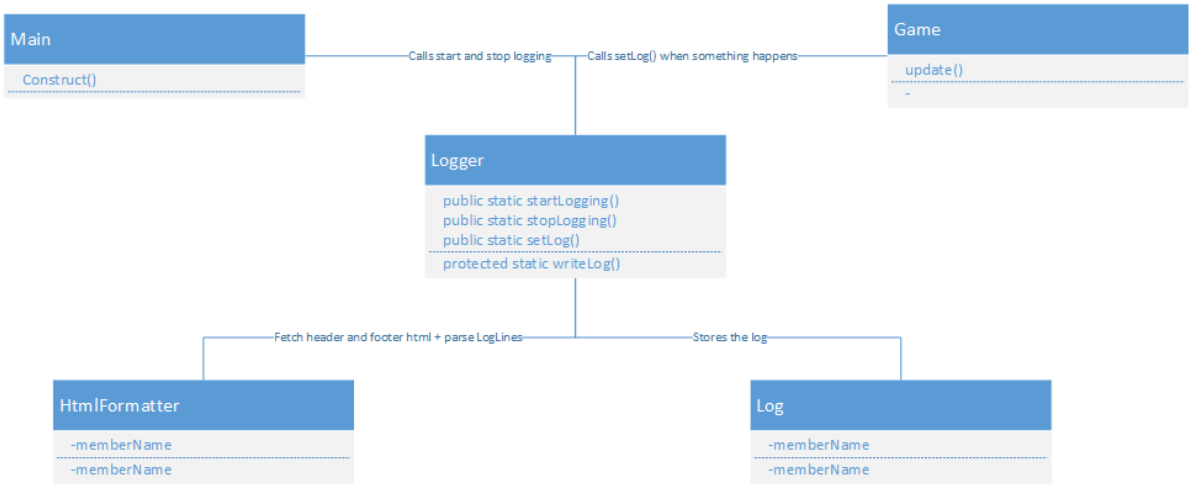
Above, our entire structure is visible in the UML, since this is very confusing, below are a few components layed out.



Above, the structure of our controllers is visible.



Above, the structure of the core of the game is visible.



Finally, we conclude with the UML presentation of the Logger.