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

Assignment 2

4.1. 20-Time

We decided to add a Story Line feature. The Story Line feature is a user experience improver. With it, the user should get more involved in attempting to complete the game. Not only is high score a real goal, defeating the final boss 'Bachalien' is one as well.

The following requirements are mandatory in our feature:

- The story is finished when the final boss is defeated
- If the player dies, the storyline should terminate, pause or change
- After each boss iteration, a text field containing the story should be shown
- Aliens, backgrounds and music in-game correspond to the story
- A 'skip' function allows the user to fast-forward to action instead of reading the story
- Text of the story is displayed in a downwards sliding way, like movie credits
- Finishing a level leads to level-transition animations

From this, we extract only one candidate class:
Storyline

The following CRC-card comes with the class StoryLine:

Class name: StoryLine	
Superclass(es): BasicStateGame	
Subclass(es): None	
Display Text	LevelFactory
Know Background	
Know Music	
Skip Story	
Level Transition	

The contract of our Storyline should contain at least methods for:

- Skipping the storyline
- Display text
- Read from XML file
- Level Transition
- Set Background
- Set Music

The other feature we decided to add is the Level Builder. The Level Builder feature is a sandbox. The purpose of it is initially to aid us in developing new levels at a high pace and with more room for creativity. Later on, it may be extended to be made available to users as well.

The Level Builder has the following requirements:

- The Level Builder environment should only be accessed when asked to do so.
- The environment should feel the same as a normal level to enhance nativity.
- The user can choose which type of alien he wants to put on the field.
- The user is able to remove any placed alien.
- After completion, the user is able to save the level to an xml-file.
- The user is also able to give the level created by him a name before saving.
- Levels are automatically added to the levels folder upon completion

From this, we extract only one candidate class: LevelBuilder Below is the CRC-card for LevelBuilder:

Class name: LevelBuilder	
Superclass(es): BasicStateGame	
Subclass(es): None	
Display all operations available	
Know background	
Control circle	Circle
Write to XML	
Delete Alien	
Add Alien	

4.2. Your wish is my command

Another extension we added this week is the high-scoreboard and its visualization. During the process of designing the high-scoreboard we followed the Responsibility Driven Design Principle.

The following requirements are mandatory in order to have a well-functioning high-scoreboard:

- The highscores should be maintained in a data-file

- The highscores should be visible to the player at the end of the game
- The highscores should be ordered from high to low
- A score should include the name of the player who achieved it
- Submitting your score should be optional
- The player should enter his/her name at the end of the game
- The highscore should be accesible from the main menu

From this, we extract the following candidate classes:

- HighScoreBoard, which will display the list of highscores
- HighScoreForm, used to submit a score to the list
- HighScoreManager, needed to maintain the HighScoreBoard
- ScoreComparator, needed to sort the scores in descending order
- Score, needed to represent the players' score with name and value

Below, the CRC-cards are listed for each candidate class:

Class name: HighScoreBoard	
Superclass(es): BasicStateGame	
Subclass(es): None	
Display List of HighScores	HighScoreManager, HighScoreForm

Class name: HighScoreForm	
Superclass(es): BasicStateGame	
Subclass(es): None	
Save name of player with score	HighScoreManager, Score, ScoreComparator
Have player enter his/her name	

Class name: HighScoreManager	
Superclass(es): BasicStateGame	
Subclass(es): None	
Maintain highScoreBoard	HighScoreBoard, Score, ScoreComparator
Read scores from file	
Write scores to file	
Sort scores in descending order	
Add new scores to list	

Class name: ScoreComparator	
Superclass(es): Comparator>	
Subclass(es): None	
Compare two scores	Score

Class name: Score	
Superclass(es): Serializable	
Subclass(es): None	
Represent score in pairs of (name, value)	

The contract of HighScoreBoard should at least contain methods for:

- Displaying the high-score list
- Updating the high-score list

The contract of HighScoreForm should at least contain methods for:

- Entering the name of the player
- Responding to keyboard-input

The contract of HighScoreManager should at least contain methods for:

- Sorting the scores in descending order
- Adding new scores to the high-score list
- Writing new scores to the datafile
- Reading scores from the datafile

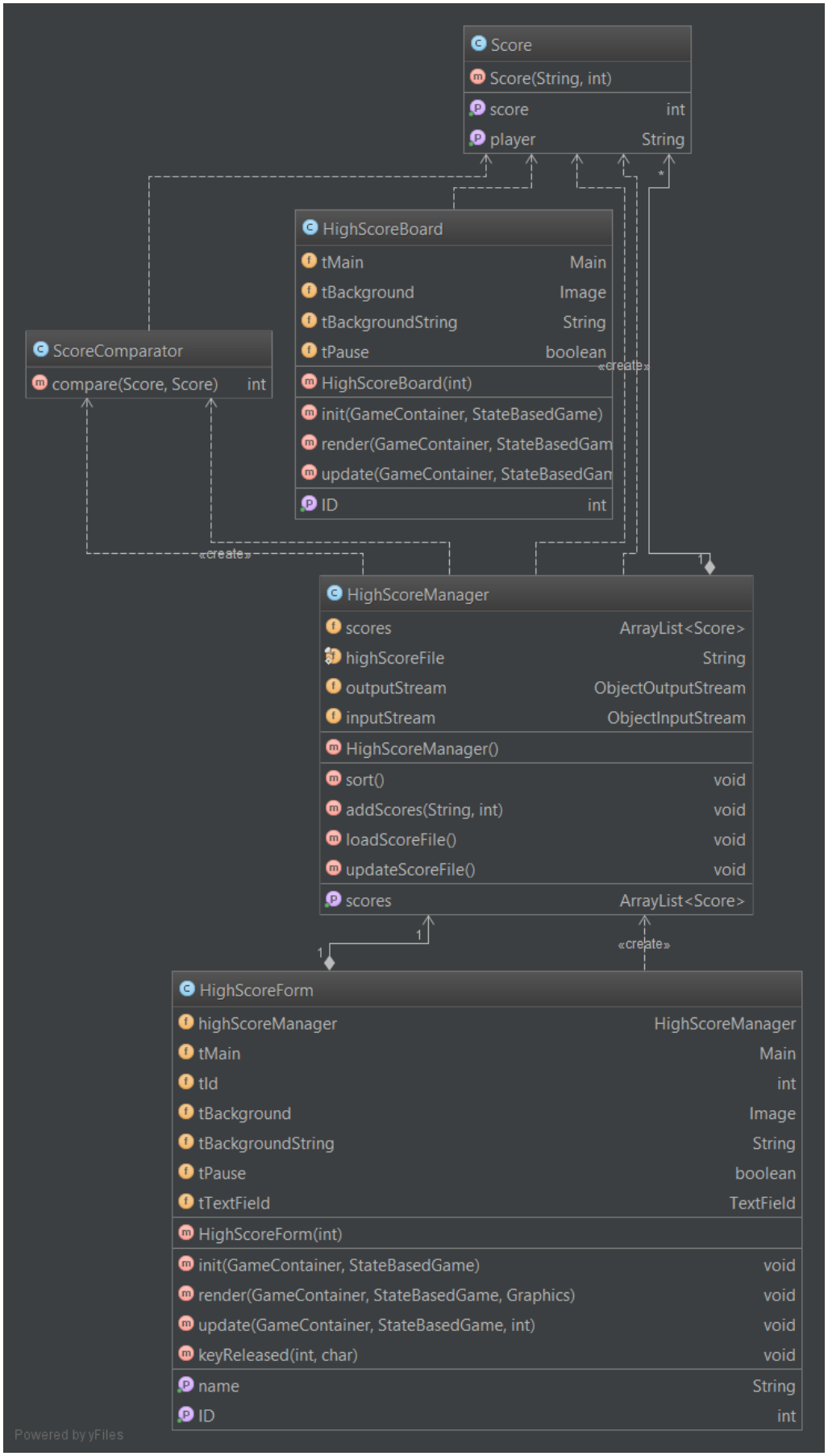
The contract of ScoreComparator should at least contain methods for:

- Comparing two scores to determine their position in the high-score list

The contract of Score should at least contain methods for:

- Getting the name of the player
- Getting the value of the score
- Setting the name of the player
- Setting the value of the score

On the next page, the UML of the newly generated classes is visible. The UML is generated with IntelliJ. In the image we can clearly see that ScoreComparator uses the class Score in its compare-method. Also, we see that most of the essential methods which build the highscore-list are in HighScoreManager, while the graphics-methods such as init, render and update are included in the HighScoreBoard and the HighScoreForm.


Score
Score(String, int)
score int
player String
HighScoreBoard
tMain Main
tBackground Image
tBackgroundString String
tPause boolean
HighScoreBoard(int)
init(GameContainer, StateBasedGame)
render(GameContainer, StateBasedGam
update(GameContainer, StateBasedGan
ID int
ScoreComparator
compare(Score, Score) int
«create»
«create»
«create»
HighScoreManager
scores ArrayList<Score>
highScoreFile String
outputStream ObjectOutputStream
inputStream ObjectInputStream
HighScoreManager()
sort() void
addScores(String, int) void
loadScoreFile() void
updateScoreFile() void
scores ArrayList<Score>
1
1
1
*
HighScoreForm
highScoreManager HighScoreManager
tMain Main
tId int
tBackground Image
tBackgroundString String
tPause boolean
tTextField TextField
HighScoreForm(int)
init(GameContainer, StateBasedGame) void
render(GameContainer, StateBasedGame, Graphics) void
update(GameContainer, StateBasedGame, int) void
keyReleased(int, char) void
name String
ID int
Powered by yFiles

5

Assignment 3

5.1. 20-Time Reloaded

One of the features we decided to add is a Multiplayergame. This feature is an extension of our basic Game-class. By this, the user will be able to play our game with a friend, which will improve the user experience.

The following is required in order for the feature to work well:

- The game will be played by two players.
- Both player can move a ship from left to right and shoot.
- There will be two ships in the game.
- Both players have independent lives.
- Both players will be able to get upgrades.
- Both players will achieve points together.
- The players will cooperate in defeating the aliens.
- The players will be able to save their score in the highscoreboard.

In order to build a multiplayergame, we do not have to create a new class. We only have to create the possibility to add an extra player to the game, therefore we will extend the Game-class. This leads to no candidate classes.
The following CRC-card belongs to the Game-class:

Class name: Game	
Superclass(es): None	
Subclass(es): None	
Two players	Player
Two moveable ships	Sprite (and subclasses)
Independent lives	Player
Get upgrades	Upgrade (and subclasses)
Score	Score
Save highscore	HighScoreBoard, HighScoreForm, HighScoreManager, Scorecomparator

We also fixed our upgrade system, below we provide the CRC-card of the upgrade class:

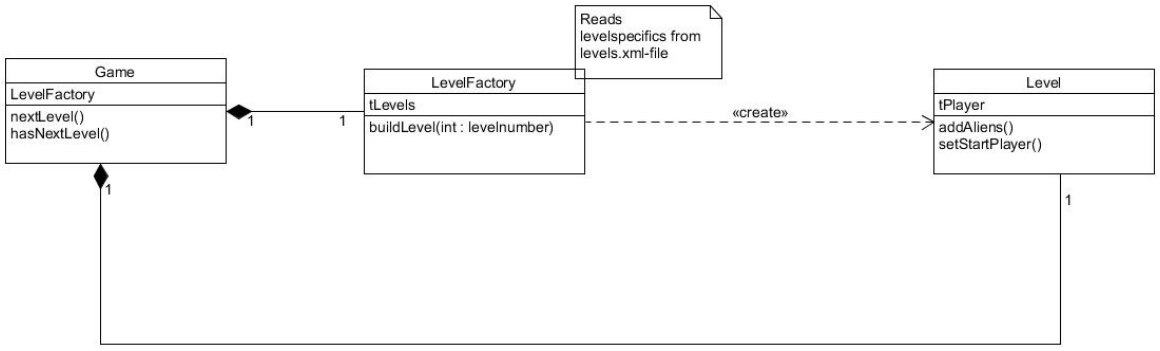
Class name:	Upgrade
Superclass(es):	Sprite
Subclasse(es):	HealthUpgrade
	PlayerUpgrade
	SpeedUpgrade
	WeaponUpgrade
Know direction	
Know speed	
Know if active	

Class name:	*Upgrade
Superclass:	Upgrade
Subclasses:	None
Know direction	
Know Speed	
Apply difficulty	Main
Know image	Main
Know if active	

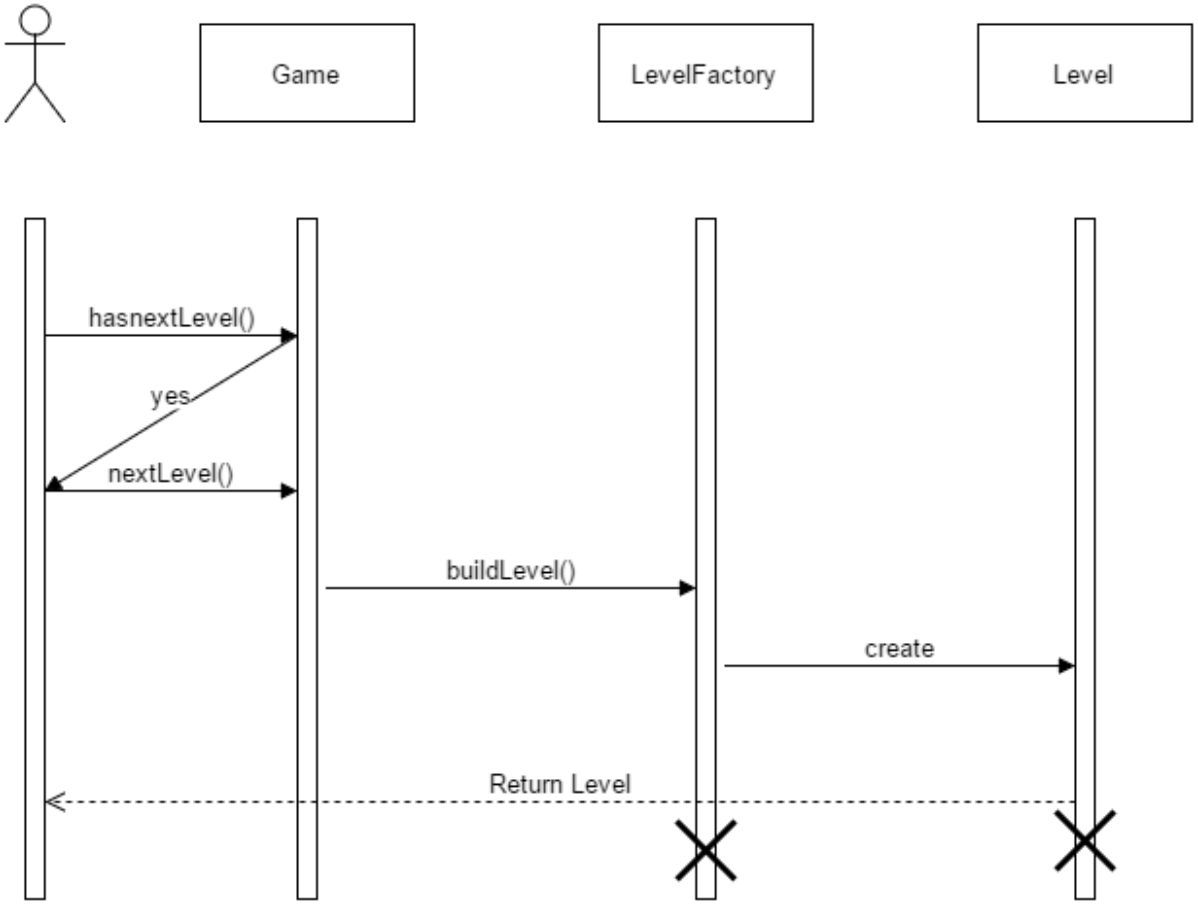
5.2. Design Patterns

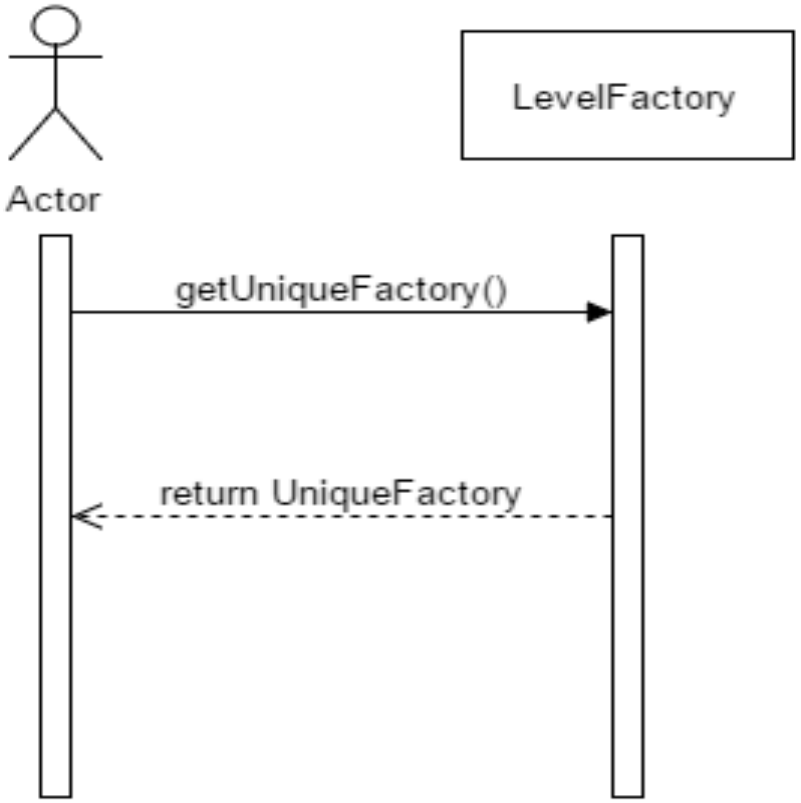
The first pattern we implemented is the Factory Pattern. We used the pattern in building a level factory. Also, in implementing the LevelFactory, we used the Singleton design pattern. We did this because we don't want to create several instances of LevelFactory. If we would have, for example, two instance of LevelFactory, the same level would be build twice, which could cause a confusion in the game, which could cause the game to crash. To ensure this doesn't happen, we used the Singleton pattern.

Below both UML-diagrams are visible, first is the diagram of the Factory Pattern. Second is the diagram of the Singleton Pattern. Both of the diagrams are simplified, we only show the parts which are relevant.



Now we show the Sequence-diagrams belonging to our implementations. First is the diagram of the Factory Pattern, last is the diagram of the Singleton Pattern.





5.3. Software Engineering Economics

Read the paper "How to Build a Good Practice Software Project Portfolio?" available on Blackboard in the 'Readings More' folder within the 'Content' section and answer the following questions:

1. Explain how good and bad practice are recognized

To determine whether something is good or bad practice, cost and duration of the project are taken into account. Something is a good practice if both cost and duration are lower than average. Something is a bad practice if both cost and duration are higher than average.

2. Explain why Visual Basic being in the good practice group is a not so interesting finding of the study.

Since Visual Basic projects are very often less complex, they are more likely to show up in the good practice quadrant, which means the probability that Visual Basic will be classified as good practice is higher than for other practices.

3. Enumerate other 3 factors that could have been studied in the paper and why you think they would belong to good/bad practice.

1. Unclear project structure: this could lead to confusion among the team members, thus this is a bad practice.
2. A good practice could be a good relation with the external supplier, since this lead to a comfortable working sphere.
3. Another bad practice would be not writing any documentation. This is bad because it will later on in the life-cycle get harder to read the code and understand what it does exactly.

4.Describe in detail 3 bad practice factors and why they belong to the bad practice group.

1. Many team changes or an inexperienced team is a bad practice since this will enlarge the duration of the project. Every time a new team member enters the team he/she has to read all the necessary code before he/she can start working on the project.
2. Using a new technology or framework is a bad practice because if not everybody know how to use the framework this will lead to extra bugs in the code.
3. Dependencies with other system is a bad practice because if the system you're depending on fails or changing your system will fail as well.