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

Assignment 5

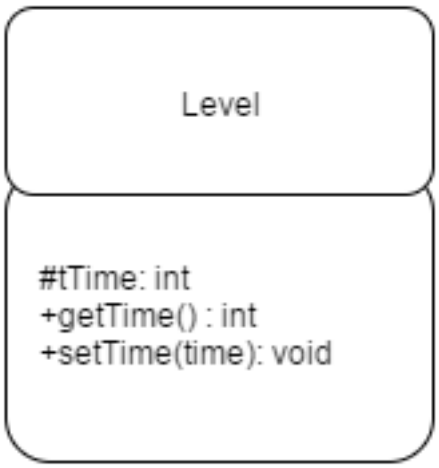
1.1. 20-Time, revolutions

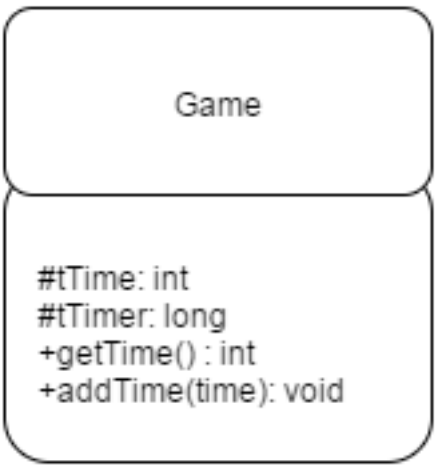
The feature we decided to add is a timer: the user now has limited time to play a level. If the user finished the level but still has time left, the score is then incremented with the amount of time that is left over.
Below we show the CRC-cards and the UML belonging to this new feature.

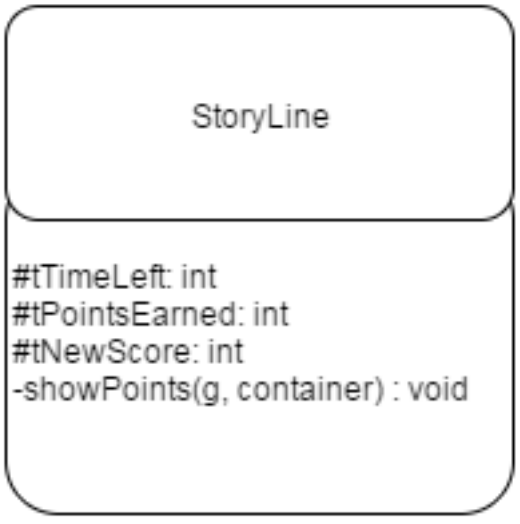
Class name: Level	
Superclass(es): None	
Subclass(es): None	
Know time received for this level	None

Class name: Game	
Superclass(es): None	
Subclass(es): None	
Knows time left	None
Add or remove time	None
Count seconds	None
Know points earned in this level	None
Know points earned in previous level	None
Lost when out of time	Main

Class name: StoryLine	
Superclass(es): BasicGameState	
Subclass(es): None	
Knows time left	None
Display time	None
Know points earned in this level	Game
Know points earned in previous level	Game
Lost when out of time	Main
Know difficulty	Main
Update points	Game





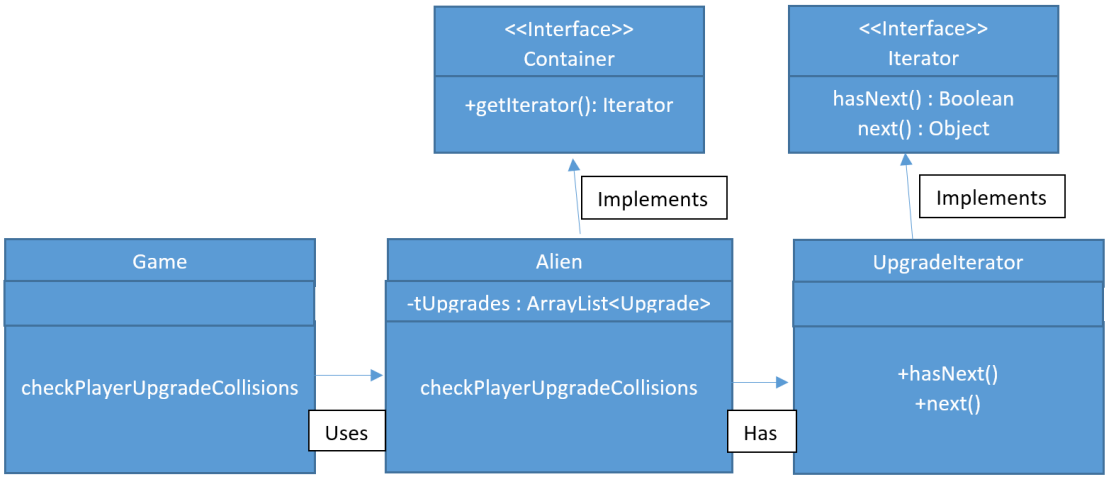


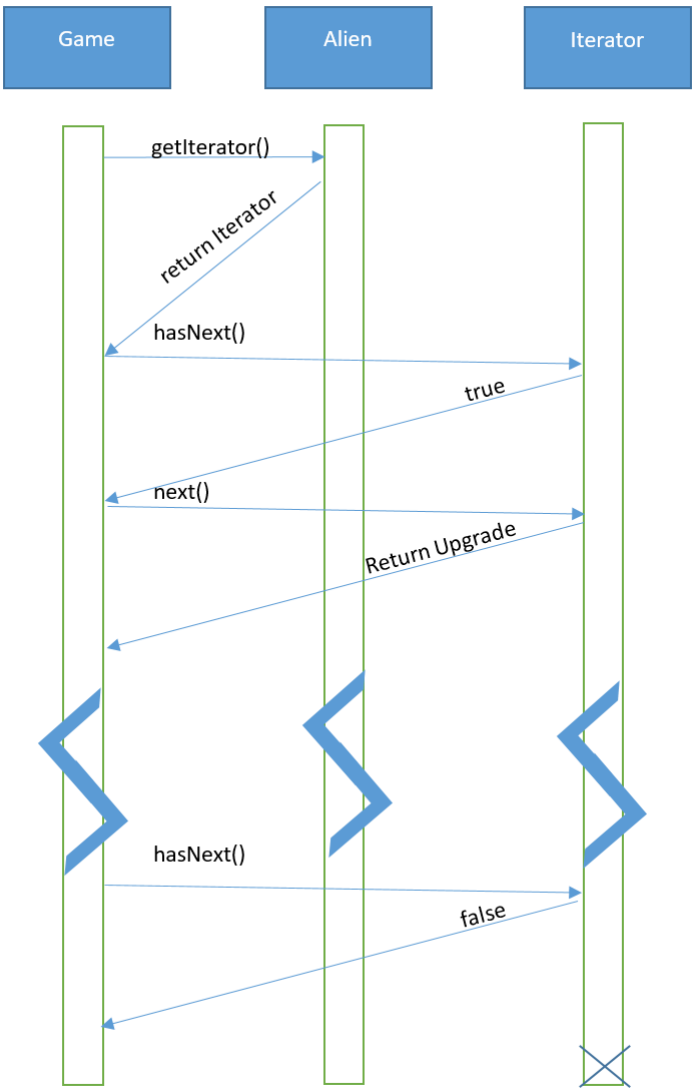
1.2. Design Patterns

The third design pattern we implemented was the Iterator Design Pattern. We chose to implement the Iterator design because this design pattern provides us a way to access elements in an ArrayList in such a way, that we are able to delete elements if we want to. When iterating through a list with a (for each) loop, this cannot be done.

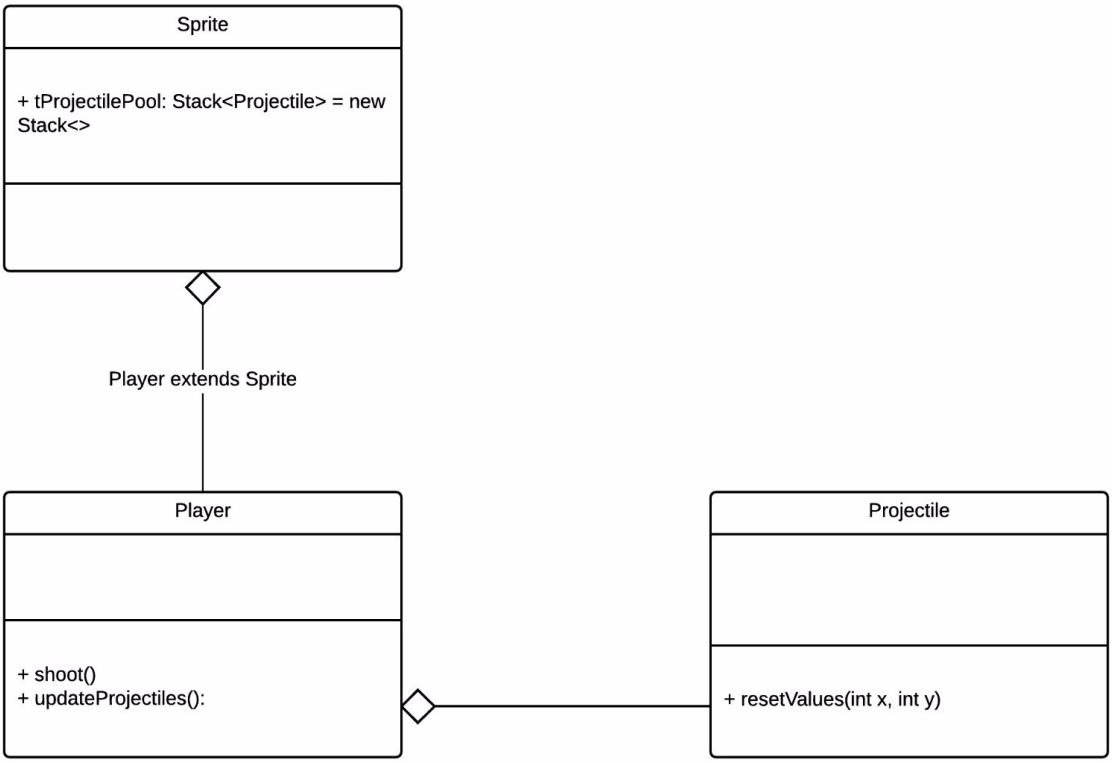
“The essence of the Iterator Factory method Pattern is to ”Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation” - Slides of lecture

Below we provide an UML and a sequence diagram of the Iterator design pattern, used in the Alien class, to iterate through the upgrades that are available.



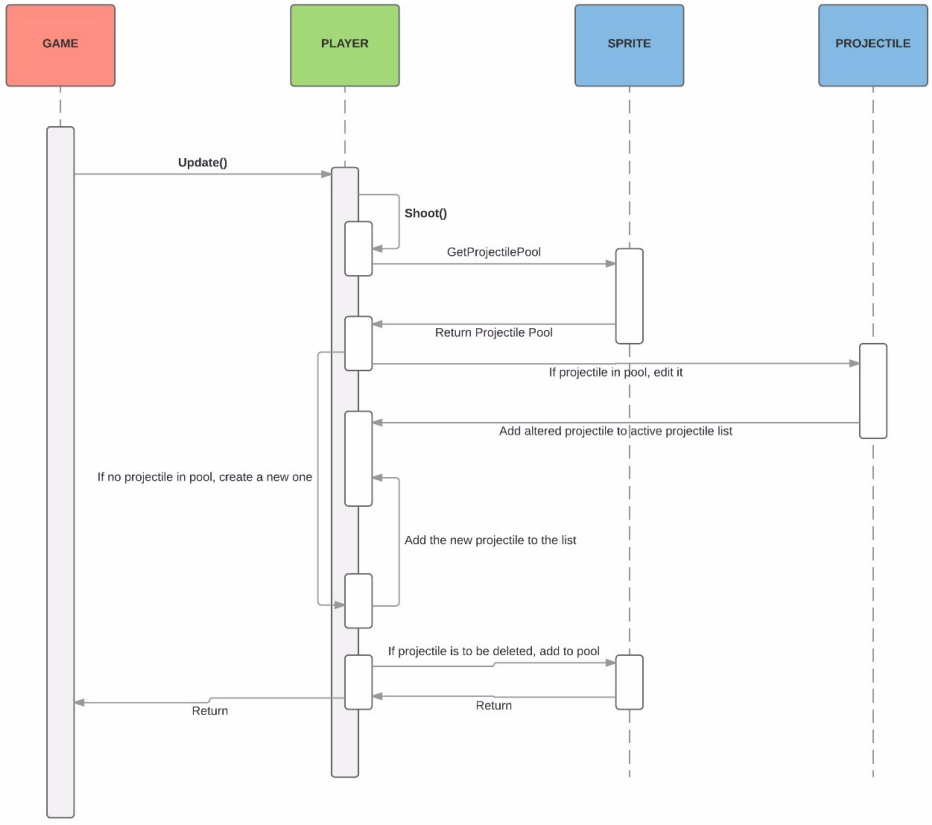


Both the players and the aliens constantly shoot projectiles. For each shot, a new projectile instance is created and is deleted after hitting an enemy or leaving the screen. In this revision using the 'object pool' pattern, we added the projectiles that were to be deleted to a special pool. When a new projectile is shot, the pool is checked first. If there is still a projectile in the pool, it is reset with the new values and reused again. In doing so we are saving up space and saving creation time of the projectile.
Below, the UML and sequence diagram of the object pool pattern are showed.



BASIC SEQUENCE DIAGRAM

Thomas | October 23, 2015



1.3. Wrap up - Reflection

When we first started the SEM lab, we needed to figure out what to do, and what the best approach was. What became clear very quickly was, that we could better work in pairs, rather than work alone. This so-called 'pair programming' is something we had also learned during our first year, and it seemed good practice to continue working like this. We decided that a team of two would think of the design and then present this to the rest of the group. This way, everyone knew what was going on, but we didn't have to be together as a group all the time. After the design plan had been approved, the team would divide the tasks into sub-tasks, which they would individually carry out. When our tasks were completed, we would get back together as a group to discuss difficulties or changes to the initial plan.

During the iterations of the Software Engineering Methods lab, our game has improved a lot. We started with our initial version, which was a very well-known version of Space Invaders: it contained one player and an army of aliens which the player has to kill. We also had a final-boss level, which contained a boss which had to be beaten in order to win the game. On top of that, our game also had background-music and sound effects which were played when shooting, dying or killing an enemy. Trough-out the iterations, we added a storyline feature, the ability to play a multiplayer game, the ability to change the difficulty and an high score board.

Since every new feature had to Responsibly Driven Designed, we learned a new way of designing a piece of software. Before, we would discuss the way we would like to implement it, and then start coding right away. This often lead to changes, because sometimes we ran into a problem which prevented us from implementing the feature the way we were set out to do. With Responsibly Drive Design, we were forced to think very precisely about how our code would we be implemented: by taking into account the classes the new feature collaborates with and the functions the new class needs to perform, we have a clearer overview of where to put it and how to code it. Also, by creating CRC-cards and UMLs, it is easier to look back on the design process later on.

Of course, this much programming leads to a lot of lines of code. In order to prevent our code from being messy and unreadable, we tried to clean it up every week. For every feature, new classes were written and new methods were added. This can lead to duplicate methods, redundant code and unused variables. Therefore, we had to look through the code and evaluate which methods could be removed or which methods could be split into two separate ones. Besides a nice and clean code, this also leads to a better understanding of why and how certain decisions were made during the design process of the code. As a result of this, every member of our team knows what each piece of code does. We think this is a good thing, because in the end, we are all responsible for the code that has been written.

Another important part in the creation of our project is testing. In order make to sure everything is working properly, we have to test every part of the code. In the beginning we struggled to get the test-coverage up to the standard.This was because our maven-builds continued to fail. Once that was fixed, we could start writing the tests and we reached the standard quickly. From that moment on, it was a matter of testing the newly added code every sprint.

Overall, the main problem we encountered was getting finished in time. Since we are all bachelor students, we have a lot of work that needs to be done for other courses as well. In the beginning, we ran into the problem that we needed to hurry in order to get things done. We

then decided to have a more strict planning, which means that we tried to have our code done on Thursday, so that we could test everything on Friday and then finish the report. This helped us a lot in managing our time and getting things done in time.

In other words, we needed a little time to get the project running, but when we got on track it was fine. We all really enjoyed working on the project and we are happy with the result. The thing we liked the most was the ‘20-Time’. This gave us the opportunity to really make the game our own and put our own inspiration into the game. We gave it our own spin and added some features we really like. We learned a lot during the lab, almost everything covered in the lectures was visible in the lab, which was a really nice thing. Also, we got to know each other a little better, which is also a good thing.