Assignment #5

Week 8

Group 2 - Bubble Bobble

Remi Flinterman
4362950, remiflint@live.nl

Arthur Guijt
4377338, a.guijt@student.tudelft.nl

Laurens Kroesen
4350286, l.kroesen@student.tudelft.nl

Thomas Overklift
4080890, t.a.r.overkliftvaupelklein@student.tudelft.nl

Lars Stegman
4365801, l.s.stegman@student.tudelft.nl

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

Supervisor: Dr. A. Bacchelli
Teaching Assistant: Aaron Ang

Table of Contents

Exercise 1 – 20-time, Revolutions

Rebindable key bindings

This week we decided that we want to be able to change the key bindings of the game in a menu. This should work in the single- and multiplayer. The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- The user shall be able to assign a new key to an action in the game.
- The game shall have a menu where the key bindings are displayed.
- The game shall have a menu where the key binding can be changed.
- The menu shall be displayed before the game starts.
- The key bound to an action shall change to undefined once that key is bound to another action.

1.2 Should Haves

- The menu shall be accessible when the game is paused.

1.3 Could Haves

- The key bindings shall be graphically represented.
- The key bindings shall be saved permanently.

1.4 Won't Haves

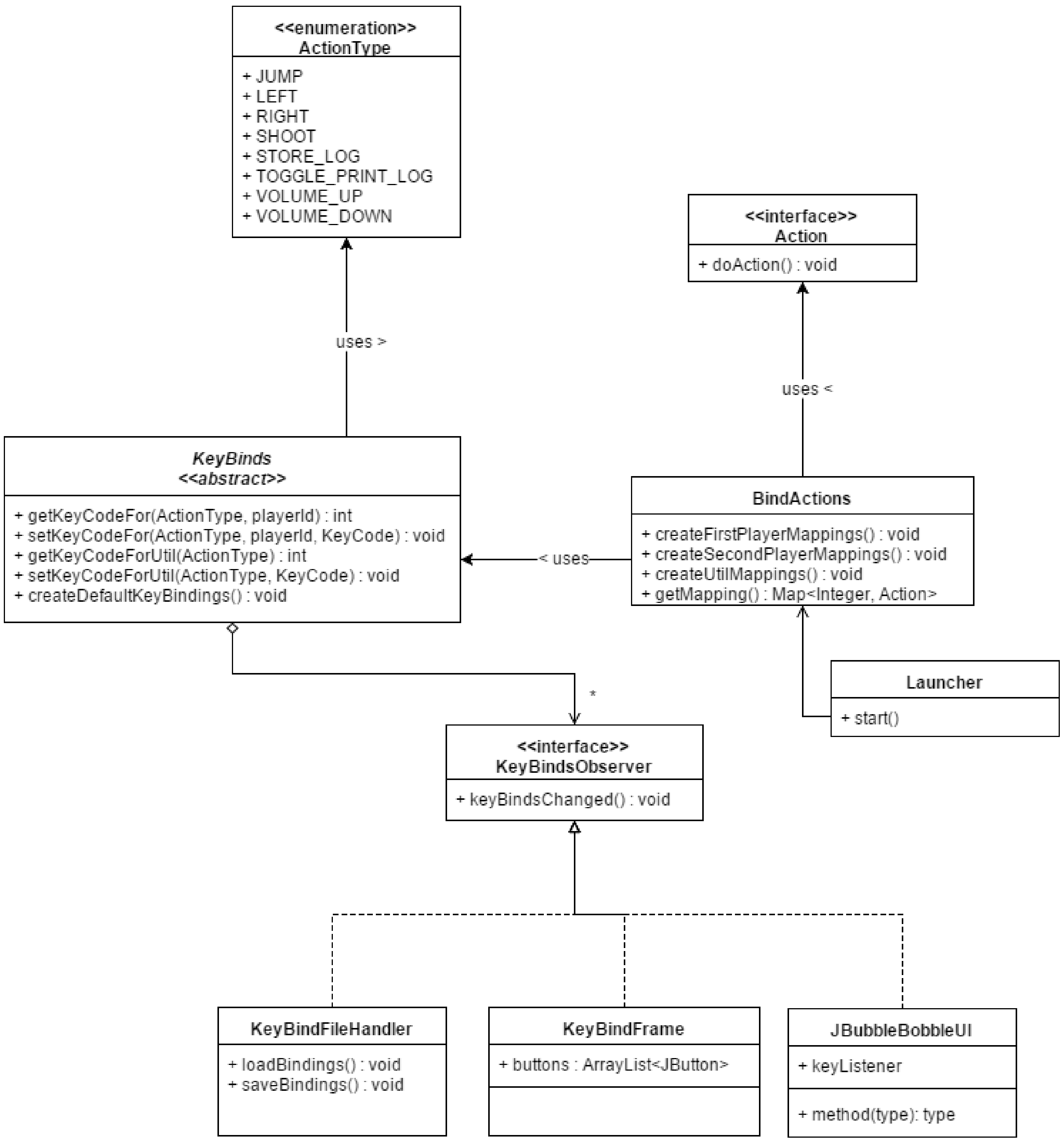
- -

2 – RDD, UML & explanation

For the key binds we had to come up with a complex system, which unfortunately does not work perfectly yet. We have a class KeyBinds which contains all the key binds of the game. Upon launching the game the KeybindFileHandler checks whether the user has already stored certain keybindings. If they did, it loads the bindings into the KeyBinds class, however if the user has not made any bindings, it makes the KeyBinds class load the default binding.

The bindings are stored using the type of actions, ActionType, as the primary key and the player index as secundary key. The KeyBinds class also contains a list of observers which are notified when a binding has changed. The filehandler receives a notification to store the bindings, the KeyBindFrame receives a notification to update the button labels. The UI receives a notification to update the key listener, so the correct keys are listened to.

The class BindActions binds the keystrokes to certain actions, which can move a player or make a player shoot.



<<enumeration>>
ActionType
+ JUMP
+ LEFT
+ RIGHT
+ SHOOT
+ STORE_LOG
+ TOGGLE_PRINT_LOG
+ VOLUME_UP
+ VOLUME_DOWN
uses >
<<interface>>
Action
+ doAction() : void
uses <
KeyBinds
<<abstract>>
+ getKeyCodeFor(ActionType, playerId) : int
+ setKeyCodeFor(ActionType, playerId, KeyCode) : void
+ getKeyCodeForUtil(ActionType) : int
+ setKeyCodeForUtil(ActionType, KeyCode) : void
+ createDefaultKeyBindings() : void
< uses
BindActions
+ createFirstPlayerMappings() : void
+ createSecondPlayerMappings() : void
+ createUtilMappings() : void
+ getMapping() : Map<Integer, Action>
Launcher
+ start()
*
<<interface>>
KeyBindsObserver
+ keyBindsChanged() : void
KeyBindFileHandler
+ loadBindings() : void
+ saveBindings() : void
KeyBindFrame
+ buttons : ArrayList<JButton>
JBubbleBobbleUI
+ keyListener
+ method(type): type

Exercise 2 – Design patterns

Part A – The factory pattern

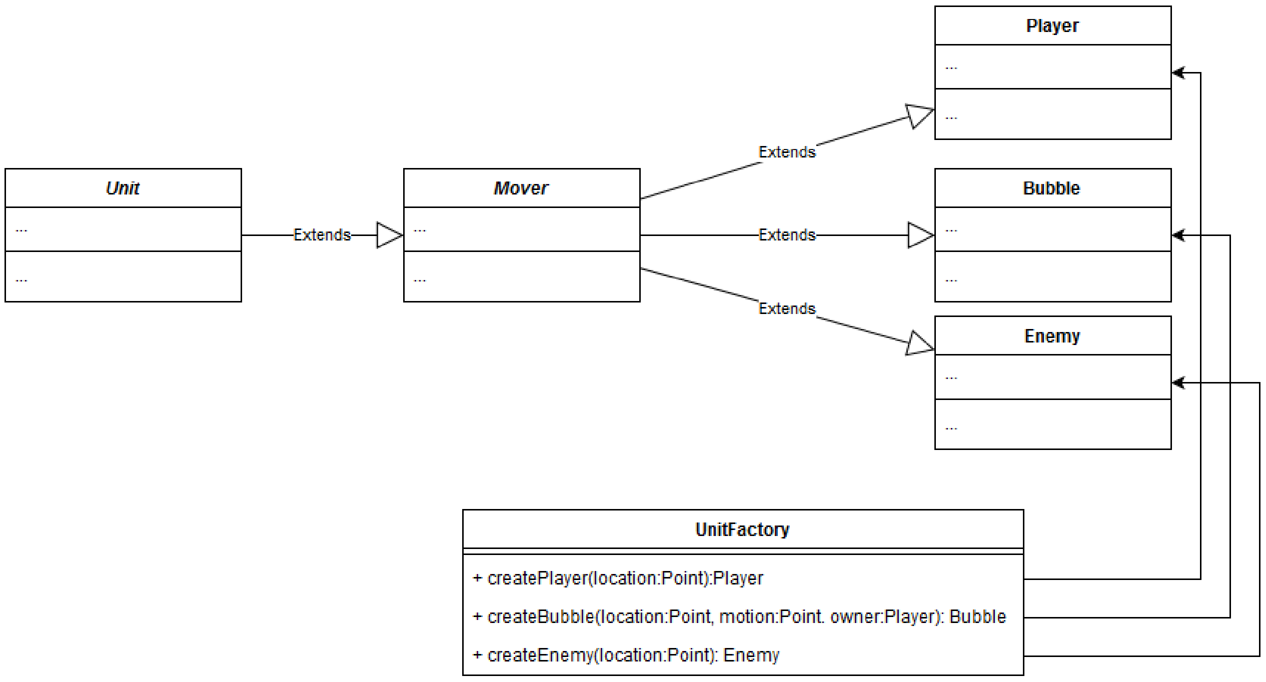
1 – Explanation in human language

This week we added the factory design pattern to the codebase so that it would be easier to swap the bubbles, players and enemies out for something else.

However, due to the link between player logic and what is currently implemented in the player class, creating an interface / abstract class at this point in time would cost way more time than is available right now and not give any significant benefit (as the major benefit of the factory pattern is the fact that you can swap out which objects are created) but rather cause a major headache.
The current player class is mostly a basic / standard implementation without any extras attached to it. Not extending the current implementation as-is right now would cause more work than needed in more ways than one, for just the formal implementation of the factory pattern.

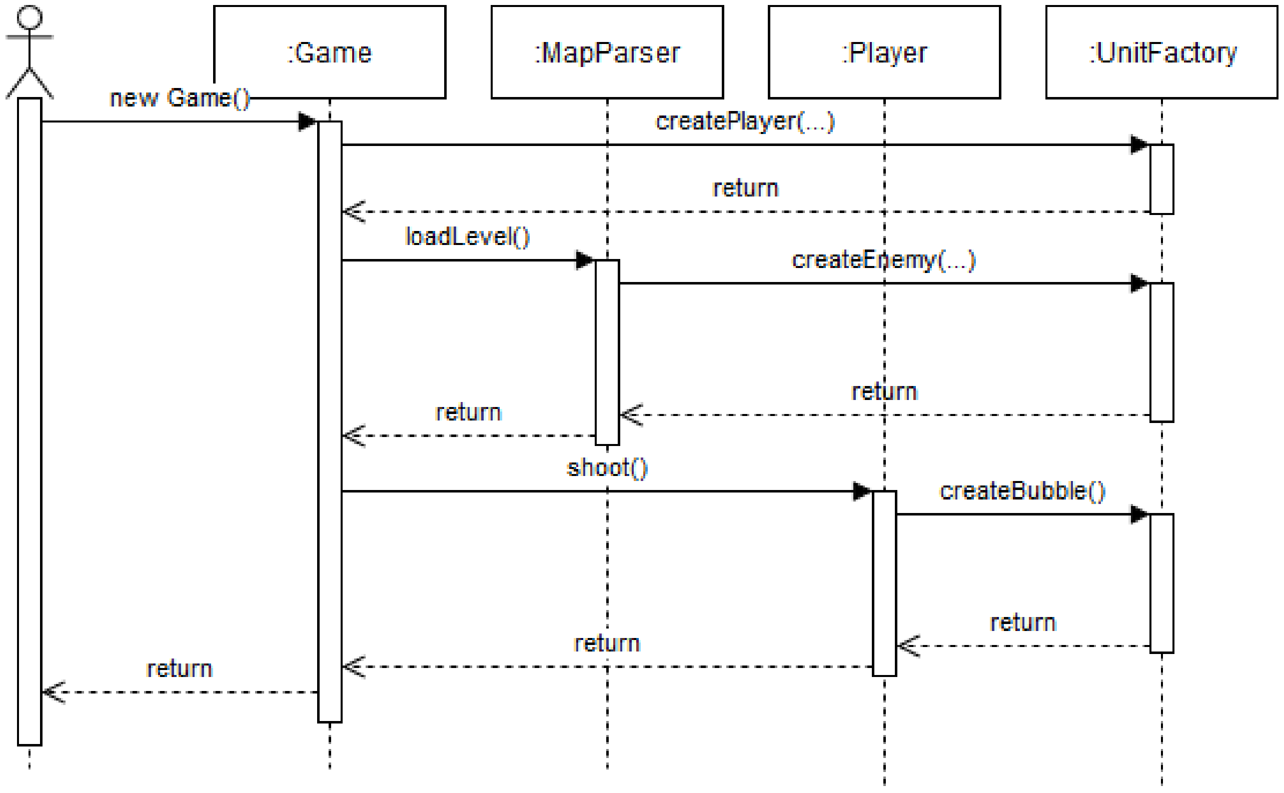
The swapping out however adds a new range of possibilities, from making enemies larger (do not recommend), making them faster or just changing their sprite. It is now possible to extend a large part of the games gameplay.

2 – Class Diagram



Player
...
...
Unit
...
...
Extends
Mover
...
...
Extends
Extends
Extends
Bubble
...
...
Enemy
...
...
UnitFactory
+ createPlayer(location:Point):Player
+ createBubble(location:Point, motion:Point. owner:Player): Bubble
+ createEnemy(location:Point): Enemy

3 – Sequence Diagram

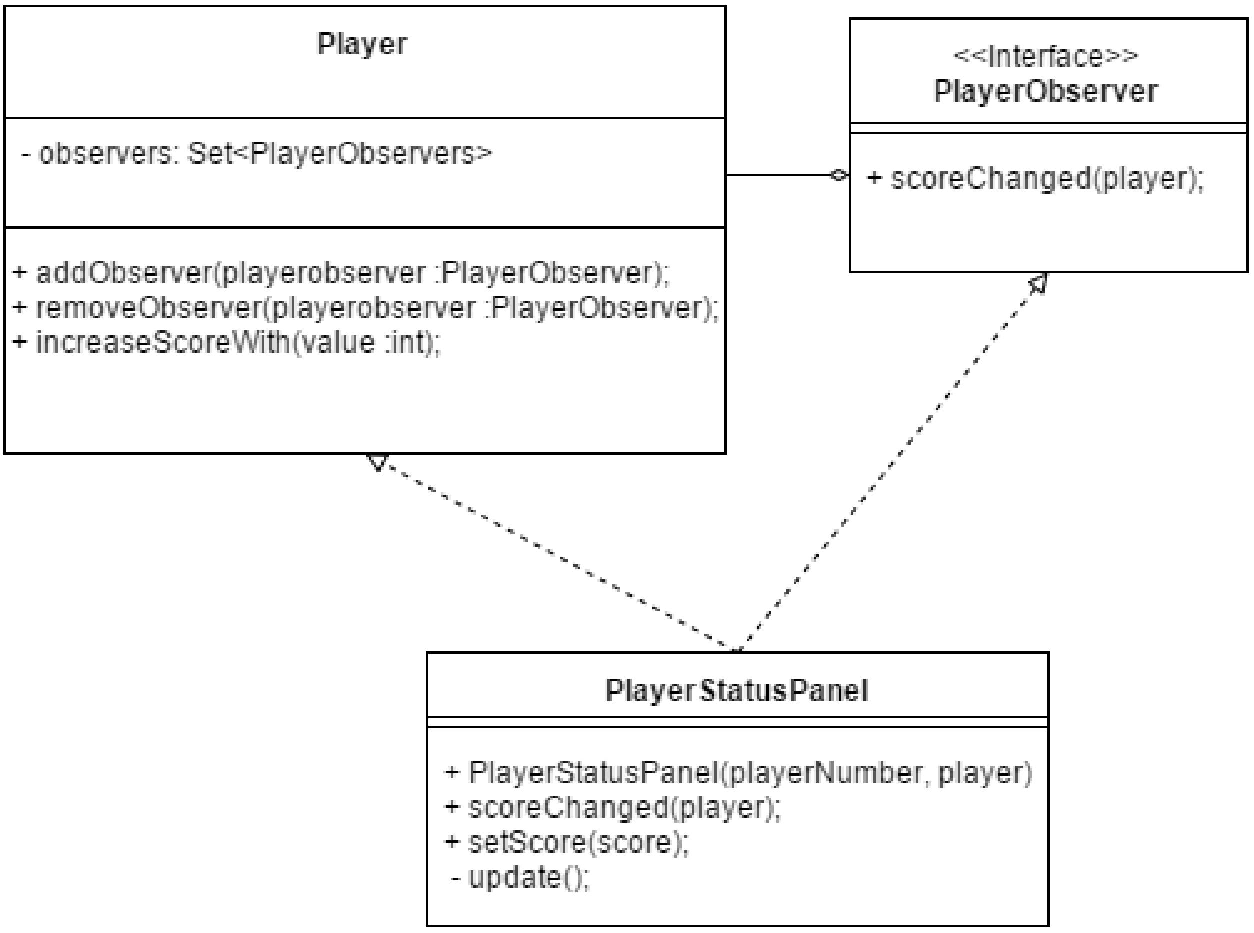


Part B – The observer pattern

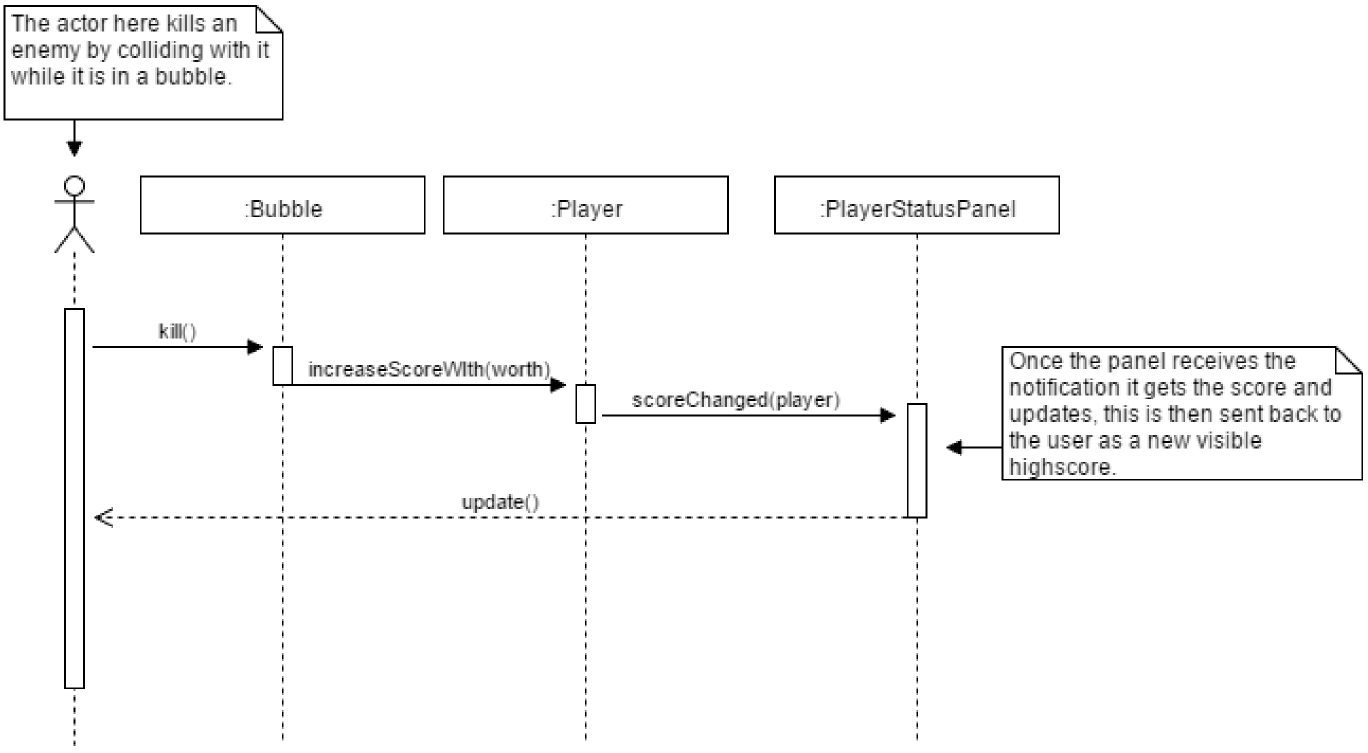
1 – Explanation in human language

We decided to use the observer pattern in the game in several instances. Here we want to talk about the one that updates the score specifically. While the game is running, we show the score of the player(s) on the side. That score needs to be updated every now and then because the player gets points. The best way to do this is using the observer pattern. This way the score gets updated whenever there is a change, because the UI is observing the player. Whenever the player gains points, the player high score is updated, and the observers are notified. The UI then knows it needs to request the new high score and show it on the screen. This way the score is always up to date without having to request the high score each tick or having too strong a coupling between players and the UI.

2 – Class Diagram


Player
- observers: Set<PlayerObservers>
+ addObserver(playerobserver :PlayerObserver);
+ removeObserver(playerobserver :PlayerObserver);
+ increaseScoreWith(value :int);
<<Interface>>
PlayerObserver
+ scoreChanged(player);
PlayerStatusPanel
+ PlayerStatusPanel(playerNumber, player)
+ scoreChanged(player);
+ setScore(score);
- update();

3 – Sequence Diagram


The actor here kills an enemy by colliding with it while it is in a bubble.
:Bubble
:Player
:PlayerStatusPanel
kill()
increaseScoreWIth(worth)
scoreChanged(player)
update()
Once the panel receives the notification it gets the score and updates, this is then sent back to the user as a new visible highscore.

Exercise 3 – Wrap up – Reflection

First off we'd like to say that our experience with SEM as a whole has been fairly positive and stimulating. We liked the fact that you have to build an entire system from the ground up, and that you get to improve it over time, while learning new things during the lectures, that you can afterwards apply to your own project in order to improve it even further.

What we've learned from the lab

The lab has been the most educational part of the course for us. This has to do with the 'hands on' factor that building a game like this has. You get to apply all the knowledge you've gathered over the past year(s) in a project, and see it flourish.

It really helps to have frequent feedback moments during the project, because the TA is actively involved with the development of the project. The TA also gets to decide what you have to do in several cases. Because of that the chances of a derailed project are reduced significantly. That is the beauty of this setup: you get to build an entire game by yourself, you can decide on a ton of features yourself, but in the meantime you learn about good and bad code & design practices while your TA guides you in the right direction.

The one week sprint iterations are very useful for feedback and making sure your project is still going in the right direction. However the effective sprint length is only 4-5 days, which is a very short time for even an agile project iteration. This means there isn't a lot of time to patch up things that've gone poorly or you'll fall behind. The assignments force you to implement certain things in a short amount of time.

While this benefits productivity it also harms the code quality: once the deadline draws near you start patching things together quickly. Then again the '20-time' presents an opportunity to amortize any technical debt that has been taken on over the course of the project. But the fact remains that if you set up the project poorly you will have a hard time fixing that, even with if you use the '20-time' for code refactoring / improvement. We decided to use our '20-time' for other features, so we haven't exactly helped ourselves there.

What we've learned from the teacher

We think it was quite useful to attend the lectures, as the material that was explained during the lectures was of great importance for the group project. It is pleasant to be able to practice the material of a lecture during the project. The fact that there were several guest speakers who had different perspectives on the effectiveness of an agile development model (SCRUM in particular), was a nice touch. This way you can form your own opinion on several of the methods presented during the course.

Another example of this was the case regarding the Unified Modeling Language. The language itself was covered quite extensively during the lectures as a great way to communicate with clients or even among developers; but Dr. Bacchelli did on the other hand also present us with a paper that shows UML might not be the most used thing in practice. This gives a balanced perspective and an opportunity for students to decide for themselves whether using UML is a good practice or not.

The lectures on design patterns were also very engaging and clarified a lot about the structure of OOP-systems, and even the way some of the Java libraries work. The material was explained in a clear manner using great examples. To us these four lectures were the best in the entire course

We also think that it's great that the teacher cares about what we think is important and that he tries to fill the lectures with content that we find stimulating. This way it's easier for students to learn new things and to have fun while taking this course.

What we've learned about each other

We must say that we're fairly happy with our group and the way our game has turned out (so far). We've all had projects before where things didn't go as smooth so we're all happy with this change of pace. It turns out our project group has a mix of people who excel at different things, so all of the important things for this course are taken care of nicely. This really helps with all kinds of things: from with setting everything up without too much hassle, to making the weekly deadlines without having to get overly stressed out. This means on the other hand that we should watch out for the fact that people do too much of the things they are good at and nobody learns anything new. We've tried to mix the tasks up a bit where possible to let everyone do a bit of everything, in order to avoid this problem.

What we'll do with this knowledge in the future

We think that we've learned a lot of things during this course that can be applied to other projects.

The fact that an agile development model is a really good idea has been made very clear during this course. This can then be combined with responsibility- and test driven development. The design patterns are also nice to keep in mind, they may not be applied each and every time, but it's good to know that they're out there and that they can be used when things get complicated. Then there is the UML language that can be used during any phase of a project to clarify things and help making choices or presenting information to a client.

So all in all, the project has been great and there's certainly a lot of material from this course that we'll use again in future projects.