Wat toelichting bij het spelen van de game:

Vuur spugen

↑

Naar links vliegen ← ↓ → Naar rechts vliegen

spacebar Vuur spugen

L Roep de logger aan

Esc Pauzeer/stop de game

Wat toelichting over onze builds:

De builds werken niet online, we hebben hier veel tijd ingestoken om uit te zoeken hoe we het kunnen fixen. We weten nog steeds niet wat de oorzaak hiervan is.
Lokaal werken de build wel. Dan kunnen we gewoon een maven site genereren die passt.

Exercise 1:

Toelichting bij wat we geimplementeerd hebben:

We hebben ervoor gekozen om level abstract te maken en daarvanuit een normal- en een survivallevel te maken. Dit is overzichtelijker en logischer.
Een normal game heeft dan meerdere levels en een survival game heeft maar één level. De levels in een normal game verschillen van elkaar. Ze worden ook steeds moeilijker (meer/grotere ballen en verschillende muren). De bedoeling is om alle levels uit te spelen. Bij de survival game is de bedoeling om zo lang mogelijk te overleven.

We hebben ervoor gezorgd dat een normal game aangmaakt wordt met behulp van een NormalGameCreator. Die maakt alle levels met bubbels en muren aan en zet ze in een list.
Bij de SurvivalGameCreator wordt een survival game aangemaakt. Daarin spawnen bubbles om de zoveel seconden en ze worden ook steeds groter.

Er is nu een optie ‘select level’ op het startscherm erbij gekomen, aangestuurd door LevelOverviewScreen. In de normal game heb je meerdere levels. De nieuwe classe LevelCompletion houdt bij hoeveel levels er uitgespeeld zijn. Je kan dan vanaf het meest verre level beginnen als je uit de game bent gegaan en je zou weer verder willen spelen.

Level (abstract)	
Superclasses:	
Subclasses: NormalLevel, SurvivalLevel	
Defines the building block for a level.	- Bubble - Player - Powerup - Wall - Rope

	- Settings

NormalLevel	
Superclasses: Level	
Subclasses:	
Creates a standard level with a fixed number of bubbles.	- Level

SurvivalLevel	
Superclasses: Level	
Subclasses:	
Creates a level in which infinity bubbles will spawn over time.	- Level

NormalLevelFactory	
Superclasses:	
Subclasses:	
Creates multiple NormalLevels.	- NormalLevel - Player - Bubble - Wall

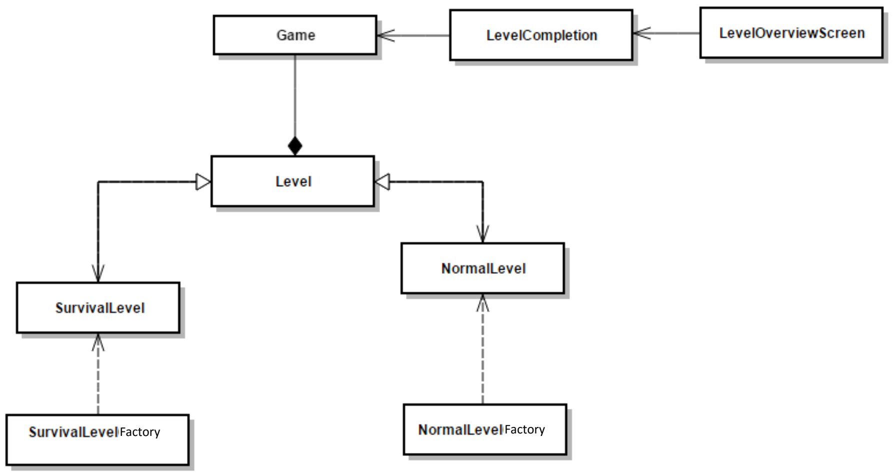
SurvivalLevelFactory	
Superclasses:	
Subclasses:	
Creates a SurvivalLevel.	- SurvivalLevel - Player - Bubble

LevelOverviewScreen	
Superclasses: JFrame	
Subclasses:	
Creates a screen where you can select levels based on how far you have got in normal mode.	- StartScreen - MainRunner

LevelCompletion	
Superclasses:	
Subclasses:	
Keeps track of which levels a player has completed.	- Game - LevelOverviewScreen

UML Diagram:

Dit diagram geeft de klassen weer die we toe willen voegen, gebaseerd op de requirements voor een nieuwe functie.



Exercise 2:

*1. Write a natural language description of **why** and how the pattern is implemented in your code.*

We decided to implement the following three design patterns: Factory Pattern, Observer pattern and Singleton pattern. We put the most emphasis on the first two patterns and a little less on the (extra) third one.

Factory Pattern

We had a lot of classes that could use a factory pattern. This because we a lot of instances of classes like Bubble are made within the game. When new bubbles are created for example we would like a special class to take care of that responsibility and that's where the factory comes in. This is also the case with the other factory implementations. When using a factory it is really easily and clean to see what and how you want to create an instance of a class. The implemented factory patterns adopted a lot of the responsibilities that were in the classes before. The classes that got a factory pattern implementation are:

- Driver
    - Since we have two different drivers for the two different game modes it was useful to get them created by a factory, because this cleaned up the mainRunner and also made the code more structured and cleaner.
- Game
    - Same argument as Driver. We have two different kinds of games. These two games need different builders. From the driver you want to be able to create the correct game according the type of driver you wanted to build. You should be able to ask for different kinds of games so we implemented methods that facilitate those needs. We used the method it was implemented in the drivers before and moved that to the factory class.
- Level
    - In a game you have more than one level (usually). Some class had to be accountable for the creation of levels. In a factory you can create as many levels as your heart desires and if you do so you won't have to change any code outside this factory. This way you can ask for any level you want and for all the levels. This factory is split into

two sub classes that facilitate the needs of levels for a survivalgame and levels for a normalgame. Those subclasses need to be there since the demand for levels is different between those two modes.

- Bubble
  - This factory was mainly usefull for the survivalgame, because we need a dynamic creator of new bubbles. Those bubbles should be randomly created and randomly placed. This responsibility was first facilitated by the survivallevel class, but a factory would handle this better and in a more structured way. So we made that happen. The factory can create a random bubble from our set of 5 different bubbles and can also place those random bubbles on a random place on the screen. Using this method the survivalgame will get random bubbles in random places that will spice up the gameplay of the survivalgame quite a bit.
- Powerup
  - We need random powerup drops. What could create random powerups? A factory! This factory can create random powerups and when we want to add a powerup we only have to create a new powerup class (which holds the image etc.) and add that new powerup to the factory and Tadaah! a new powerup is added to the game. That is very neat in a structural way and makes the game much more expandable and adaptable. This factory only holds that one responsibility so it is a good practice for responsibility driven design.
- Rope
  - When the player picks up the icerope powerup we want an ice rope to be created this check is done within the rope factory so when we need an ice rope we get one. If we don't have the icerope powerup we should shoot a normal rope. This is done with an if statement in the createrope method. We can easily expand on the different types of ropes this way.

These factories are instantiated in our project and have methods that fullfill the needs of it's connected class and responsibility that the factory has. All the methods return a desired instantiation of the desired class. The implementations of these instantiations differ between the factories. Every instantiation has other rules (duh) those rules are defined within the corresponding classes themselves.

On a side note; We (heavily) use our Settings class, especially for the handling of powerups. We find this Settings class a great way to let classes communicate on how to behave on certain things. This also prevents us from declaring instances of game in the Player class for example. The Settings class is a clear and pleasant way to invoke certain behaviour on objects. End of side note.

Observer Pattern

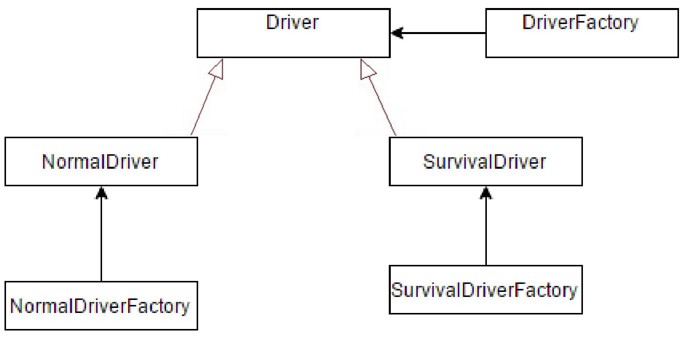
We implemented the observer pattern on a smaller scale. We wanted to have our collisions handled in one class called Collisions. It was a challenge to get that responsibility completely inside that class, but we did it. When collisions happened we ofcourse wanted something that would result from that collision, but if we added that to the collision class we would add a responsibility and that is not supposed to happen. We started to think of a way to solve this problem. We decided to use the observer pattern for this. When a collision occurs we want a observer to be notified. This is done by calling an update method on the observer we want to update from the state change. We searched the internet for a couple examples and we found that the observers mainly were in one list and updated all at once. In our case that would end up buggy and game breaking. This led us to build different observers from an observer superclass that forced subclasses to implement a certain update function, but since these different observers should have different items in their parameters we had to use Object instead of a specific class in the parameters of the update function. This is not a neat solution, but it works. The implementations of the update functions typecast the objects into the right objects that were given in the method call. We could afford to do so since we know for sure that those typecasts will never be wrong since we strictly monitor the the method calls.

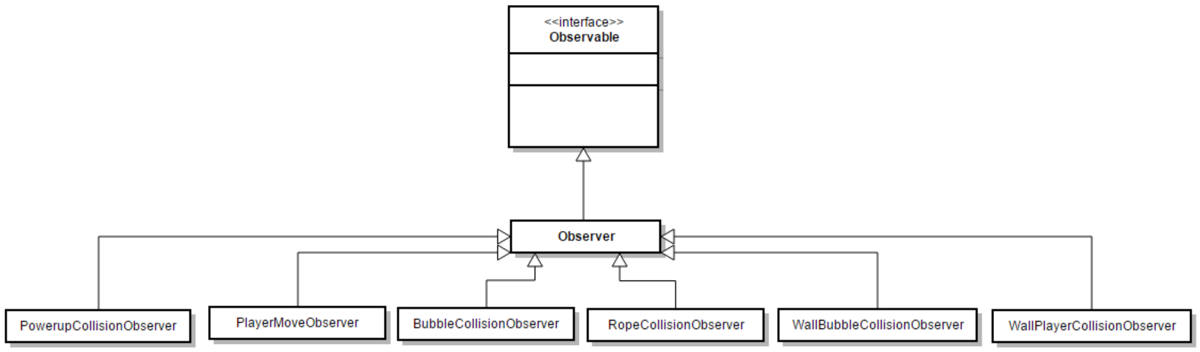
We also thought of another way to fix this problem, our second fix was to just give a game in the method parameter call or to declare a game variable inside the observer. The problem with this solution is that in case of (for example) a collision between a bubble and the rope occurs we have to search for the colliding bubble a second time, which is quite inefficient. So it was better to just give the bubble, the rope and the game (to handle the consequences of the collision) with the method call. The same holds for the other types of collisions.

Singleton Pattern

Last but not least we have the singleton pattern on the score. Using this pattern we are able to have only one instantiation of the score object that can be accessed from everywhere in the game. So when we need to update or check our score we can carelessly do that. This will prevent us from accidentally implementing a second score and therefore have a wrong resulting score. We implemented the pattern by having the constructor of score check whether an instance already exist, if not create a new one. All it's methods should be in static reach, so we implemented that.

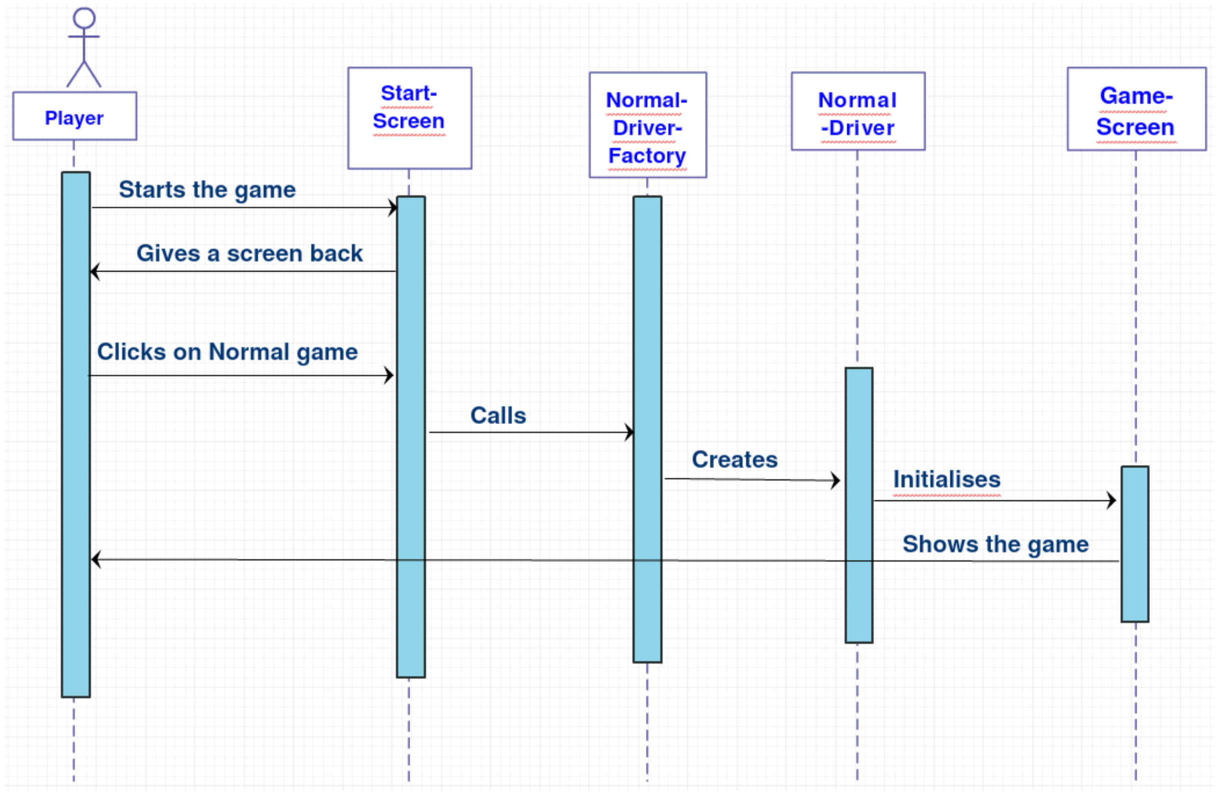
2. Make a class diagram of how the pattern is structured statically in your code.





3. Make a sequence diagram of how the pattern works dynamically in your code.

Example of what happens when a player starts a normal game. (Showing the Factory pattern)



Exercise 3:

1. Explain how good and bad practice are recognized.

First the overall average performance of a project is analyzed, with regards to project size, project cost and project duration, measured in function points, euros and months. A project is good practice when the performance on both cost and duration is better than the average cost and duration, corrected for the project size. A project is classified as bad practice when the performance on both cost and duration is worse than the average, corrected for project size.

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

First of all there were not many projects analyzed that used Visual Basic. Only 6 projects used Visual Basic so that is a very small sample size. It also helps that Visual Basic projects environments on average are less complex than others, and therefore and up in the Good Practice quadrant more often.

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

<u>Full time/part time</u>

This would distinguish between projects that get the developers undivided attention, and the projects which are not the only project that the developer is doing.

It would be interesting to see if there was a difference between these projects. It would be logical to see that the 'full time' projects end up in the good practice section more often.

<u>Amount of team members</u>

The amount of team members would be an interesting to take into consideration. When analyzing many projects and taking team size into account, it would be possible to discover

the ‘perfect’ team size for a developing team. Small teams (<6) would probably end up in good practice, and big teams in bad practice.
<u>Culture of the country of the team</u>
Different countries have different cultures. These different cultures will also influence the way of how a team works. Some will have a good impact and some will have a bad impact. This would be a really interesting subject to add to the paper.

4. Describe in detail 3 bad practice factors and why they belong to the bad practice group.
<u>Rules & Regulations driven.</u>
When a project is driven by rules and regulation, the project is done because it is enforced by some other instance, for example the government. This brings down the morale of the development team a lot. They’re not working on a feature because they want to create that feature, but because they have to. Projects driven by Rules & Regulations also often have a very strict deadline, which can lead to features being implemented in a hurry, without following proper design patterns.

<u>Dependencies</u>
When a project depends on a lot of frameworks or other projects, it could happen that certain features can’t be implemented, simply because the framework or project your own project depends on is not built for it. This can lead to a lot of frustration within the team, and missing features in the final product. If the features are essential for the product, the team would have to find a way to work around the dependencies, which would cost more time and money.

<u>Many team changes, inexperienced team</u>
It is very important for a development team to be experienced, and used to each other. We have noticed this even in our own group, it’s much easier to work together when you know each other well. Many team changes will also lead to bad practice, as you would need to teach the new members everything about the project from scratch, which takes a lot of time.