Assignment 3

Alex Geenen, Bart de Jonge, Mark van de Ruit, Menno Oudshoorn, Stefan de Vringer

Software Engineering Methods, Group 1

Exercise 1: 20-time, reloaded

For this exercise, we first show the new requirements for the new features we wished to develop. Keep in mind that we spent most of this assignment on polishing and improving what was already there, while also adding some new elements.

Functional Requirements

Must haves

- A set of new powerups, which, when picked up by a player in either a normal or multiplayer game, will positively or negatively impact one or more players by applying one of the following effects:
    - Slow down the speed of all bouncing circles in the level to half their original speed for a set time.
    - Increase the speed of all bouncing circles in the level to twice their original speed for a set time.
    - Completely freeze all currently-existing bouncing circles in the level for a set time.
    - Add a life, so the player can get killed more often, but only if there are less than five lives.
    - Assign one of all the existing or new effects completely at random.

Should haves: none. Could haves: none. Won't haves: none.

Non-functional Requirements

- The game must always properly start or restart with the correct number of lives, the correct score, and the correct level.
- The game must no longer be able to start in multiplayer with only one player connected.
- Players in multiplayer display their movement correctly, or as close to correct as can be achieved over a connection, without much jarring movement or any laggy jumps
- The timeout-time for when connections drop in a multiplayer game, should be far lower than the current timeout-time, and should not exceed 3 seconds.
- The game must be unable to crash due to concurrency issues, which pose too great a danger right now.

Now that we have discussed the requirements for exercise 1, we show how we have continued using Responsibility Driven Design to develop these new features. We do this by displaying the new and updated classes in CRC cards. We have also created a (rather massive) class diagram using UML, but this is, once again, **too large for this document, and can be found in the appropriate place in our deliveries folder**.

To clarify: CRC cards contain the class name, superclass, and subclasses. Then the bottom-left part of a crc card describes responsibilities, while the bottom-right part describes collaborations.

Name: LevelFactory	
Superclass: none	
Subclasses: LevelFactorySinglePlayer, LevelFactoryMultiPlayer	
Create and return the correct level with the correct circles, gates and time	LevelFactorySinglePlayer

Name: LevelFactorySinglePlayer	
Superclass: LevelFactory	
Subclasses: none	
Create and return the correct single-player level with the correct circles, gates and time	LevelFactory

Name: LevelFactoryMultiPlayer	
Superclass: LevelFactory	
Subclasses: none	
Create and return the correct multi-player level with the correct circles, gates and time	LevelFactory

Name: Level	
Superclass: none	
Subclasses: Level1,Level2,Level3, etc.	
Be constructed as a level correctly	Level1/Level2/Level3 etc.

Name: Level1/Level2/Level3 etc.	
Superclass: Level	
Subclasses: none	
Be constructed as a level correctly	Level
Store information about a particular level.	BouncingCircle, Gate,

Name: Connector	
Superclass: none	
Subclasses: Host, Client	
Communicate basics during LAN-multiplayer	Client, Host
Update data in game	GameState
Send messages concerning changes during LAN Multiplayer	GameState, Client, Host

Name: ShutDownHook	
Superclass: none	
Subclasses: none	
Support logging while game crashes or shuts down	Logger
Support disconnecting LAN multiplayer while game crashes or shuts down	Connector, Host, Client

Name: SpeedPowerup	
Superclass: none	
Subclasses: FastPowerup, SlowPowerup, FreezePowerup	
Cause the speed of BouncingCircles to vary when activated	BouncingCircle

Name: FastPowerup	
Superclass: SpeedPowerup	
Subclasses: none	
Cause the speed of all BouncingCircles to increase when activated	BouncingCircle

Name: SlowPowerup	
Superclass: SpeedPowerup	
Subclasses: none	
Cause the speed of all BouncingCircles to decrease when activated	BouncingCircle

Name: FreezePowerup	
Superclass: SpeedPowerup	
Subclasses: none	
Cause the speed of existing BouncingCircles to drop to 0 when activated	BouncingCircle

Exercise 2: Design Patterns - singleton pattern

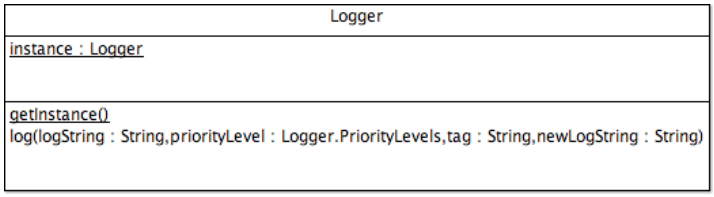
Natural language description

We implemented the Singleton design pattern for our game's logger, which handles all logging, both to console and to file. Before, we just instantiated a *Logger* object in the *MainGame* class, and used a getter method to retrieve this object when we needed it. However, this required having a *MainGame* variable accessible in every class that used the logger, and made testing difficult.

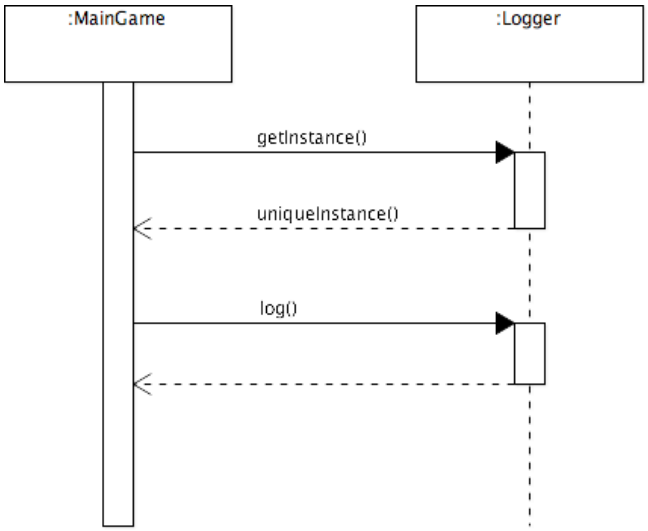
Thus we have chosen to implement this pattern, which is useful here because we only need one logger instance; we don't want to write to multiple files for example.

It is implemented as follows: In the *Logger* class, we have a private static attribute of type Logger called instance. This instance is instantiated directly using a private constructor, so this does not happen in a method. This is to prevent concurrent threads making more than one instance. The *Logger* class does not have a public constructor, so loggers can't be instantiated from the outside world. If you want to get the single *Logger* instance, you have to call the static method *getInstance()*, ensuring you always get this single instance. Now, if other classes want to log, they can just call *Logger.getInstance().log()*.

Class Diagram



Sequence Diagram



Exercise 2: Design Patterns - factory pattern

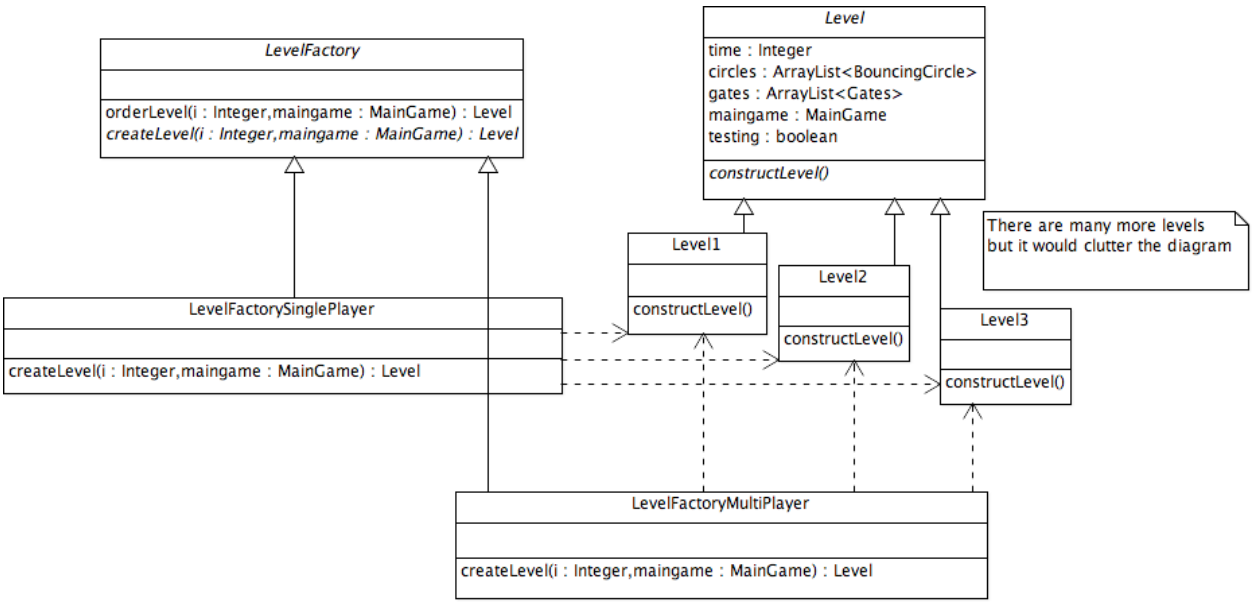
Natural language description

The factory pattern was implemented in our project because of the obvious utility to our game's levels situation. The way our levels were implemented was that there was one general class Level which was used to construct a level. All of the contents were then tacked on one by one in *LevelContainer*, which was a far from elegant solution. Seeing as there was one general class used for many different custom levels this was a good candidate for the factory pattern.

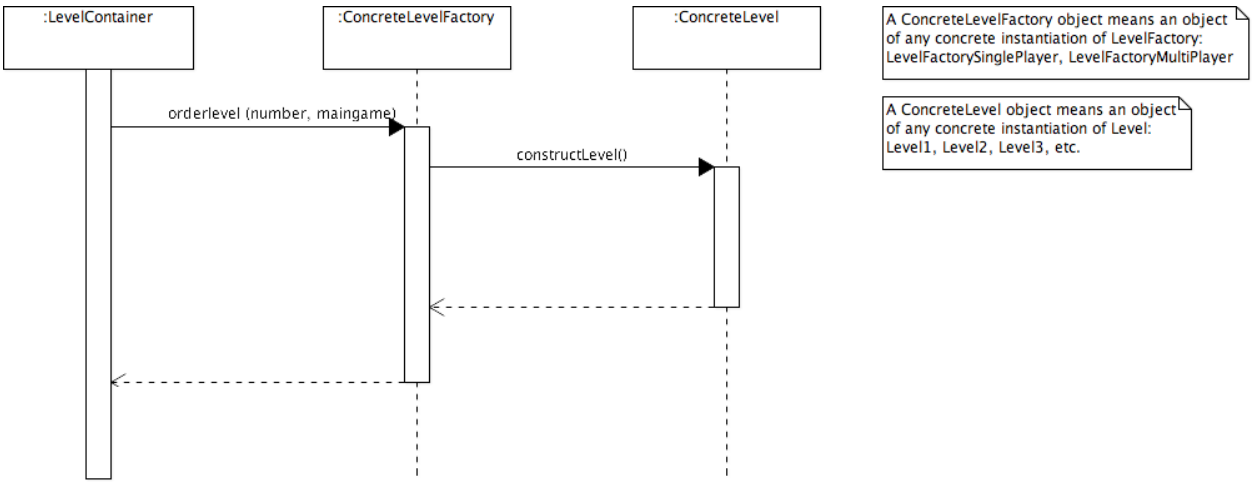
The factory pattern was implemented with two factories, one for the multiplayer and another for the single player game modes, due to the difference in level implementation for each. What was previously a class: *Level*, has been changed into an abstract class, with class implementations being *LevelN* (with *N* being a level number, e.g. *Level1* or *Level2*). These individual implementations of the different levels are now responsible for their own construction.

Levels are now fetched by the *LevelContainer* by ordering a level from one of the two factories, passing an integer representing the desired level number, which then delivers a constructed level to be used in the game. Inside the factory, the level number is then used to instantiate an object of the correct class, which is subsequently returned.

Class Diagram



Sequence Diagram



Exercise 3: Software Engineering Economics

Part 1: *Explain how good and bad practices are recognized.*

First, the size, cost and duration of each project in the dataset is calculated. The size of a project is determined by the number of function points. Determining the number of function points of a project is a rather complicated process.

Afterwards, all projects are put into a Cost/Duration matrix. A project that is on average worse in both cost and duration is considered a bad practice. A project that is on average better in both cost and duration is considered a good practice. Notice that all bad practices and all good practices are collected in their own quadrants inside of the Cost/Duration matrix.

Part 2: *Explain why visual basic being in the good practice group is a not so interesting finding of the study.*

This is not a very interesting finding of the study because the number of projects that used Visual Basic was insignificant when compared to the overall total of projects (i.e. 6 of 352). These 6 projects all had between 27 and 586 Function Points allocated to them, making them all small to medium projects. The combination of the small variance in Function Points and the small Visual Basic sample group means that this is not a statistically significant result.

Part 3: *Enumerate 3 other factors that could have been studied in the paper and why do you think they should belong in good/bad practice.*

Three other factors which could have been studied in the paper:

- Fundamental feature pivots: If the number of major changes made to the product requirements is large then the project may be at risk of delays, so this could be a bad practice.
- TDD: Test driven development in practice may help to decrease the number of bugs introduced during a project's programming lifecycle. As such, this could be classified as a good practice.
- Continuous Integration: The use of Continuous Integration may help to increase the overall code quality and decrease the amount of defects if implemented/used correctly. As such, this could be classified as a good practice.

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

The bad practice factors that will be highlighted here are:

1. Dependencies with other systems
2. Once only projects
3. Many team changes, inexperienced team

1. Dependencies with other systems

This is clearly a bad practice factor. When you are dependent on something else, you are much more vulnerable.

For example, when something changes in a system you are dependent on, you might need to adapt your own project to keep it compatible with the other system. If this is comes unexpectedly, it may cause time pressure, massive design changes, etc.. Combined, these may all lead to either instant project failure, or unmaintainable code, causing the project to fail after a certain time.

Another possibility is the system your project depends on failing. This would cause even more trouble, as the project has to either become independent, or switch to another system. This is disastrous if it happens in a later stage of progres.

2. Once only projects

The main reason for once-only projects mostly failing, is that there are no ‘deadlines’ between the start and final release of the project. This means you don’t get a chance to learn from experience, nor reflect on and act on mistakes made during the project while it is still in progress.

The opposite of once-only projects, release base projects, mostly succeed because they do accommodate the above-mentioned elements.

Furthermore, because the work you do will only be for a single project, you might be less motivated to write maintainable code, knowing you won’t need to reuse it. This can obviously lead to software failures.

3. Many team changes, inexperienced team

There are many reasons why this is a bad practice factor. First off, an inexperienced team means inexperienced developers, which generally results in bad code. As much as a developer can learn from theory, they can learn twice that through practice.

Secondly, something closely related to this factor is the so called “truck factor”, which indicates how disastrous it would be to a project if a team member were to be hit by a truck, so to speak. If a team constantly changes, this will add massive overhead - new team members will need to be introduced, for example. Even if all team members have equal responsibility, this will not help the project.