Assignment 5

Group 30

Novel feature: Difficulties

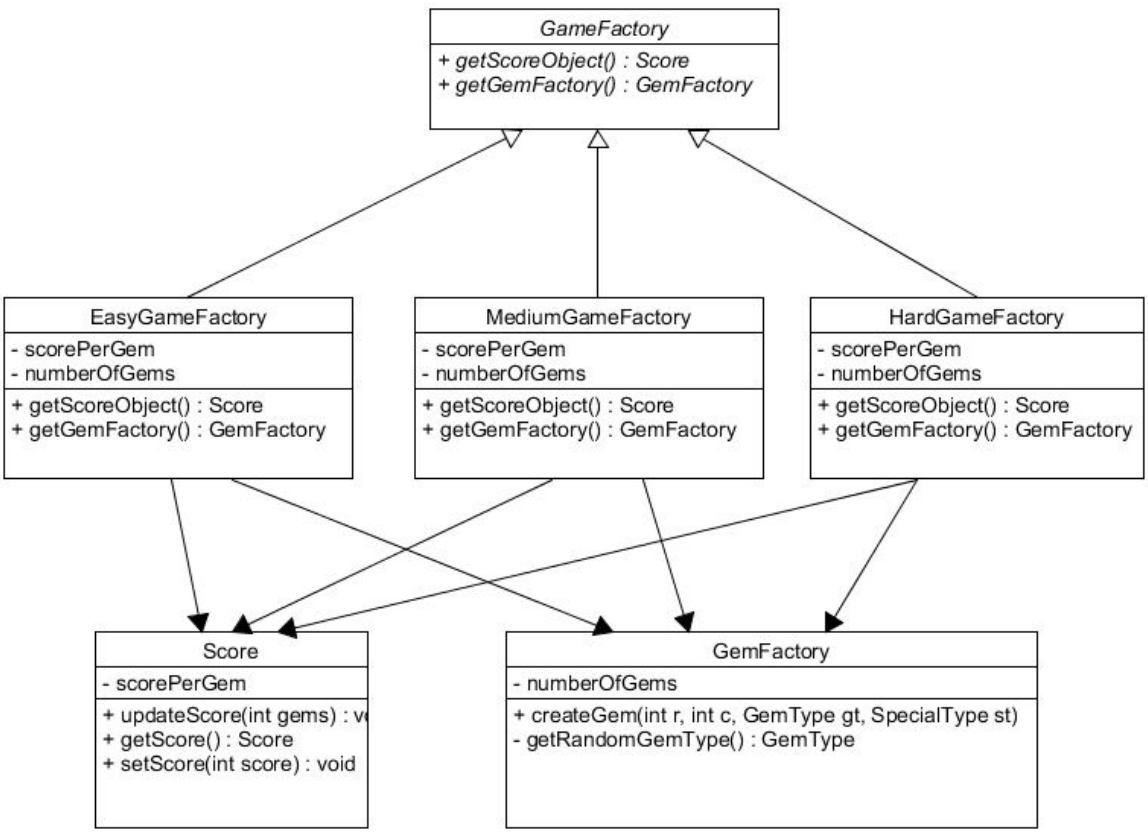
The general idea

The user will be able to choose a difficulty level from the main menu when starting a new game. The difficulty will influence how many different colurs of gems there are and how many points each gem will yield when a combination is made. The player can choose between easy, medium and hard mode. Easy difficulty will only use 5 different colours of gems, but in return, the gems will only give the player 5 points per gem. Medium difficulty will use 6 different gems and they will each give the player 10 points, which was the standard of our game before we implemented the difficulties. Hard mode will use 7 different colours of gems and the gems will each reward the player with 15 points.

Implementation

We will make different GameFactories, which each have the responsibility to set the score rewarded per gem and instantiate the GemFactory correctly corresponding to the chosen difficulty. This means that the GemFactory indeed only makes 5 different colours of gems in easy mode for example and the score will be 5 per gem. We will also have a general abstract GameFactory. Inheriting from the abstract GameFactory will be the EasyGameFactory, MediumGameFactory and HardGameFactory.

Class Diagram



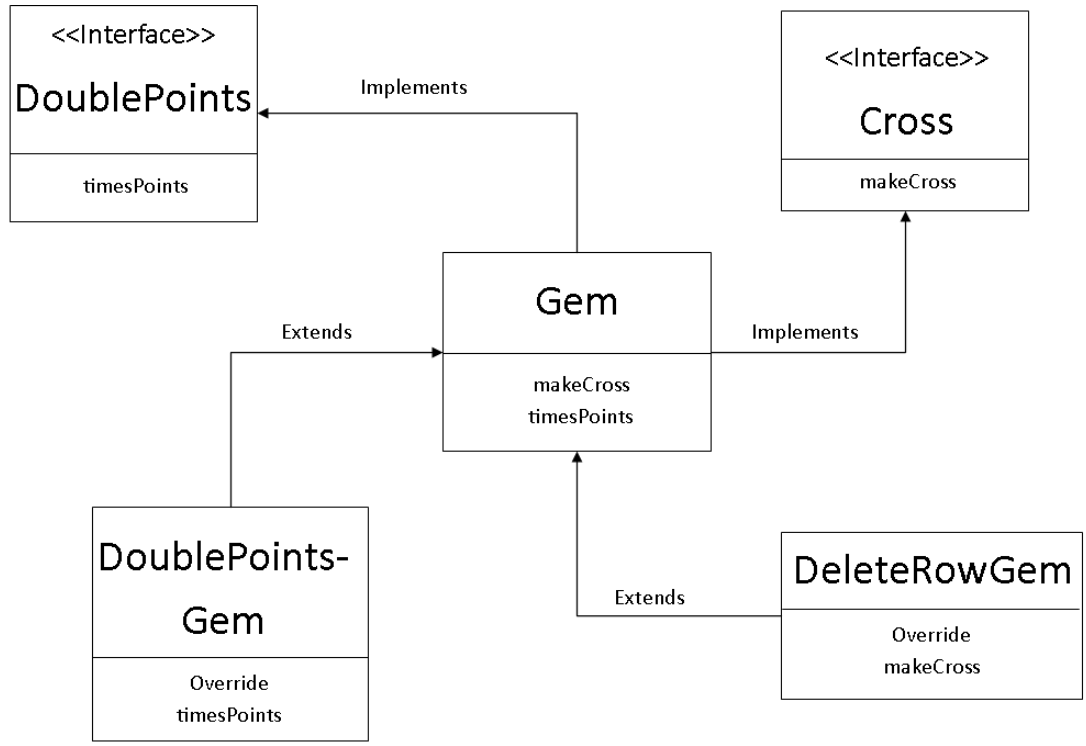
Implementing two Design Patterns

We've implemented two new Design Patterns in our game, the Strategy Pattern and the Iterator Pattern. Both patterns are explained and for every implementation a sequence and class diagram is made.

Strategy Pattern

We decided to use the strategy design patters on the different types of special Gems. We have two kinds of things that can happen when a special Gem is popped, it can give you double score or delete a cross of Gems. For both of these functionalities we created an interface, the Cross and the DoublePoints interface. They both have one method and both are implemented by Gem. In Gem they will not have much functionality, but in the classes that extend Gem (DeleteRowGem and DoublePointsGem), more functionality is assigned to them. Now they take a little responsibilty away from GameLogic and take it to their own classes. On top of that, we could stop using 'instanceof' and instead use the methods from the interfaces on all Gems, which is a nice improvement.

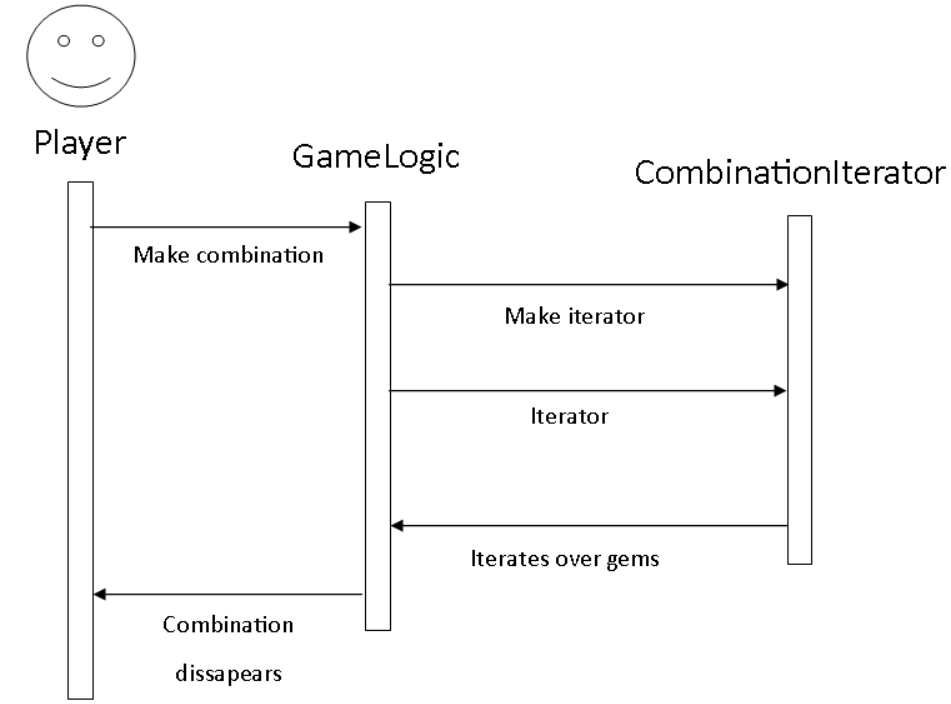
Class diagram



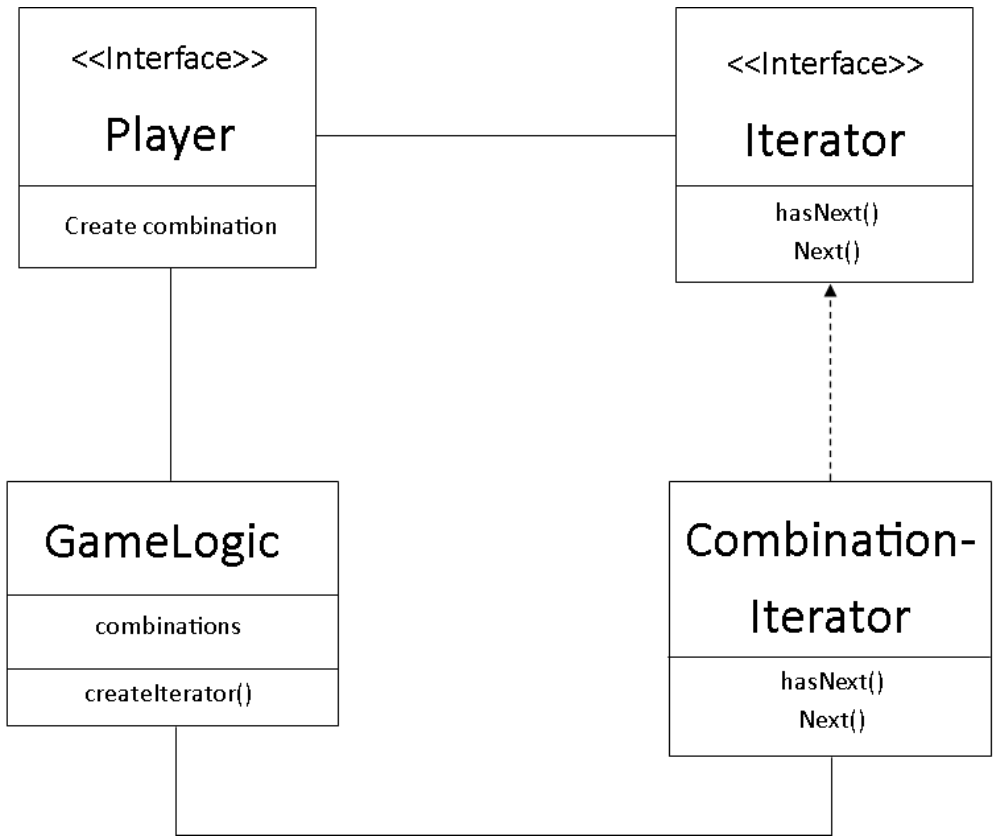
Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. We decided to use this pattern, so we did not have to iterate over a arraylist in a method. The iterator iterates over a combination, and thus returns the gems in the combination. We created one interface Iterator and implemented it with a CombinationIterator class. The iterator is created and used in GameLogic. In this way the GameLogic does not have to create an array en check if there is a next Gem.

Sequence diagram



Class diagram



Reflection on the project

Group 30

Introduction

In this reflection we look back at the project of Software Engineering Methods. After a summer holiday, building a fully working game in just 10 days was a challenge we had to deal with. After making teams we gathered together and talked about the game we wanted to make. Bejeweled was our choice and when this choice was accepted, we started thinking about how to build this game. Our team, consisting of people with different backgrounds (3 Computer Science, 1 Mathematics and 1 Systems Engineering Policy and Management student) made a list of requirements which had to be implemented in the first version.

Summary of sprints

The first sprint we took was the building of the initial game. Using our requirements we divided the work and built the first working version of Bejeweled. The game wasn't looking very good, but most of the the game's rules and logic was implemented. Testing was done in various JUnit test cases.

In the second sprint we mainly focussed on implementing tools in the code. We've tried to implement Maven, PMD, Findbugs, Checkstyle, Travis CI and Cobertura. The first four worked perfectly, but we couldn't get Travis CI and Cobertura to work. In this sprint we also implemented high scores and used Responsibility Driven Design for the first time. We also implemented a logging function. Additionally, we implemented background music and sound effects.

In the third sprint, the focus was back on the game. We've implemented new features like motivational popups, a hint button, the option to mute the background music, a pause menu and the option for the player to save the game. While we implemented those features, we used Design Patterns for the first time. We implemented the Singleton pattern for the sounds and Logger in the game, which resulted in clearer code.

The biggest change in the gameplay was done in sprint four. We managed to animate all the gem movements by using JavaFX. After this sprint, the player was game over when there were no moves left. New gems with special abilities were designed and implemented. Also, the Factory Design Pattern as well as the Observer Pattern were implemented.

In sprint five, we finally got Travis CI and Cobertura to work. Apart from that, we mainly focussed on bug and UI improvement. Most of the user interface was redesigned and was made more consistent. A lot of the player's options were moved to the pause menu, making the game look more simple. Also, a new analysis tool called InCode Analysis was used to detect flaws in our game.

In the last sprint before releasing the final version, we implemented more Design Patterns. The implementation of different game difficulties, which was our own novel feature, was

made using the Abstract Factory Pattern. Furthermore, we integrated the Iterator Design Pattern and Strategy design pattern into our code.

In the final sprint we are planning to spice up the animations, add some tests, redesign the ugly parts of the game and maybe add a small easter egg.

Learned so far

We've learned quite a lot in this project. The use of Design Patterns in software development was new for us and gave some new ways to improve our code. We unanimously agree on the fact that using design patterns in your code, can make your code better. We've implemented five patterns in our version of Bejeweled. Some were slightly redundant, others were really beneficial to the clarity and quality of our code. The Singleton Pattern was a good idea to implement for one-instance objects like the Logger. Same story for the Observer Design Pattern, which made updating the game state easier. We will definitely use these patterns again, as they are more commonly needed and easily implementable on a smaller scale. The other patterns were Iterator, Strategy and (Abstract)Factory. We could have done without the Strategy, Iterator and (abstract)Factory patterns, although we do understand that it would have proven more useful in terms of readability/maintainability of our code if our project would grow in size. We can each still see ourselves using them in the future.

It also became apparent that preparation beforehand and discussing non-functional requirements were not a joke. In the first sprint already, we all giddily implemented a working version of our game in Java Swing. We then changed our mind in the same sprint to make our project in JavaFX, which cost us quite a bit of stress.

The introduction of new tools like Travis CI didn't work out so well. The use of Travis CI did not contribute a lot when it was finally working. If the program was unable to run, Travis CI told us with annoying emails, while we already knew that. It also was hard to find help on Travis CI, which made integrating it with our code hard. Tools like PMD and Checkstyle were more pleasing to work with. The use of inCode is sprint five had also limited added value, we wouldn't use this tool in future work.

Using GitHub and the option to make different branches and merging those branches together is something we couldn't work without. Also the reviewing each other's code before implementing it in the master was a good addition to the workflow, we will definitely use that in future work.

The SCRUM method is a good method to start with. The workflow, listing requirements, make a sprint planning and reflect on the previous sprint, gave our team a good workflow. The making of a sprint planning was very useful. The reflection on the previous sprint however, felt like a waste of time. We already knew at the end of the sprint what wasn't done and had to be built in the next sprint.

Last but not least, the use of the Unified Modelling Language. Although UML wasn't totally new for us, this was the first time we were urged to use it. Again, the use of UML just for the

use of it is a bad idea, but it definitely helped us to design the implementation of new features and Design Patterns.

The team

The collaboration went well. We had no issues with members freeriding or not communicating enough. Everyone did what they were asked to do. Of course some differences were noticeable. Some members did more coding, some made more of the exercises and some focussed more on getting tools working. Looking back on the development of this game, we can conclude that our team did a great job.