Assignment 5 – Group 23

Exercise 1

1. Requirements:

   Must haves:

   - The game shall have a transition screen between each completed level.
   - The transition screen shall information about the level that it is about to start (such as level number and/or type of aliens)

2. CRC cards

Class name: Transition	
Superclass: Screen	
Subclasses:	
Purpose:	
Transition between level	Game, levelFactory
Show information about the next level	levelFactory

3. UML

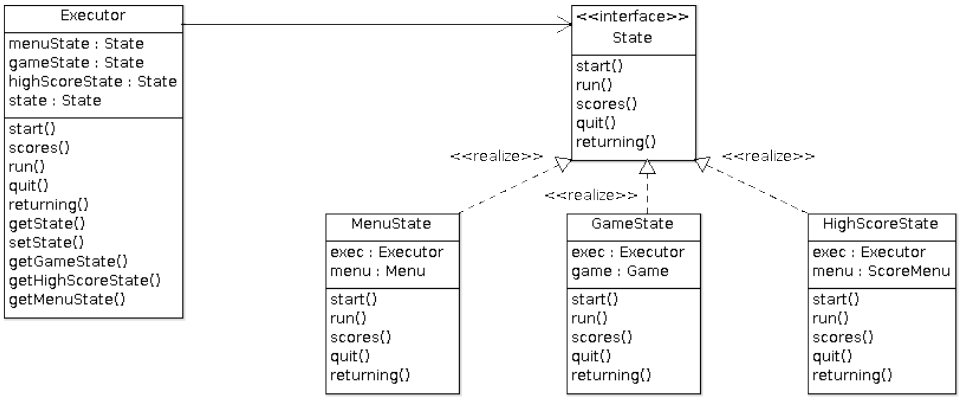
Same as the class diagram of the State pattern, with an extra state: Transition.

Exercise 2

1. State pattern:
   a. Description

The state pattern is a very useful pattern for us because our system can easily be devided in 3 different states: Menu, high score menu and game. Menu is the state in which the game starts showing the title and 3 buttons that all lead to another state/action when they are pressed. The 'new game' button makes the system go to the game state, the 'high scores' button makes the system go to the high score menu state and the 'quit' button quits the system. The game state starts a new game when entered and runs it. When the game ends or the screen is closed, the system returns to the menu state. The high scores state shows a screen with high scores and 3 buttons. The 'return' button makes the system go back into the menu state, the 'clear' button clears the scoreboard and the 'quit' button quits the system. The functions used in the State interface to make these actions possible are start, run, scores, quit and returning.

b. Class diagram

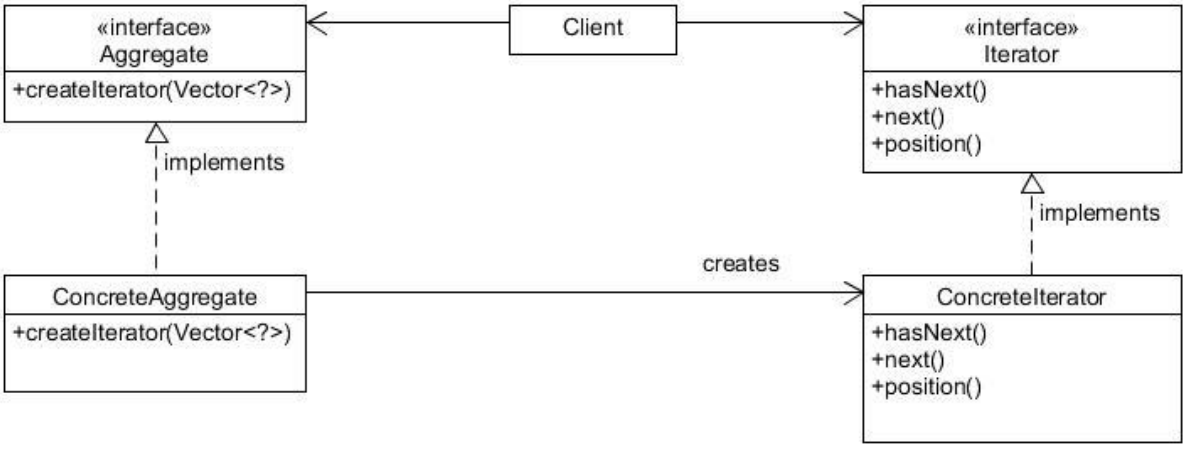


c. Sequence diagram

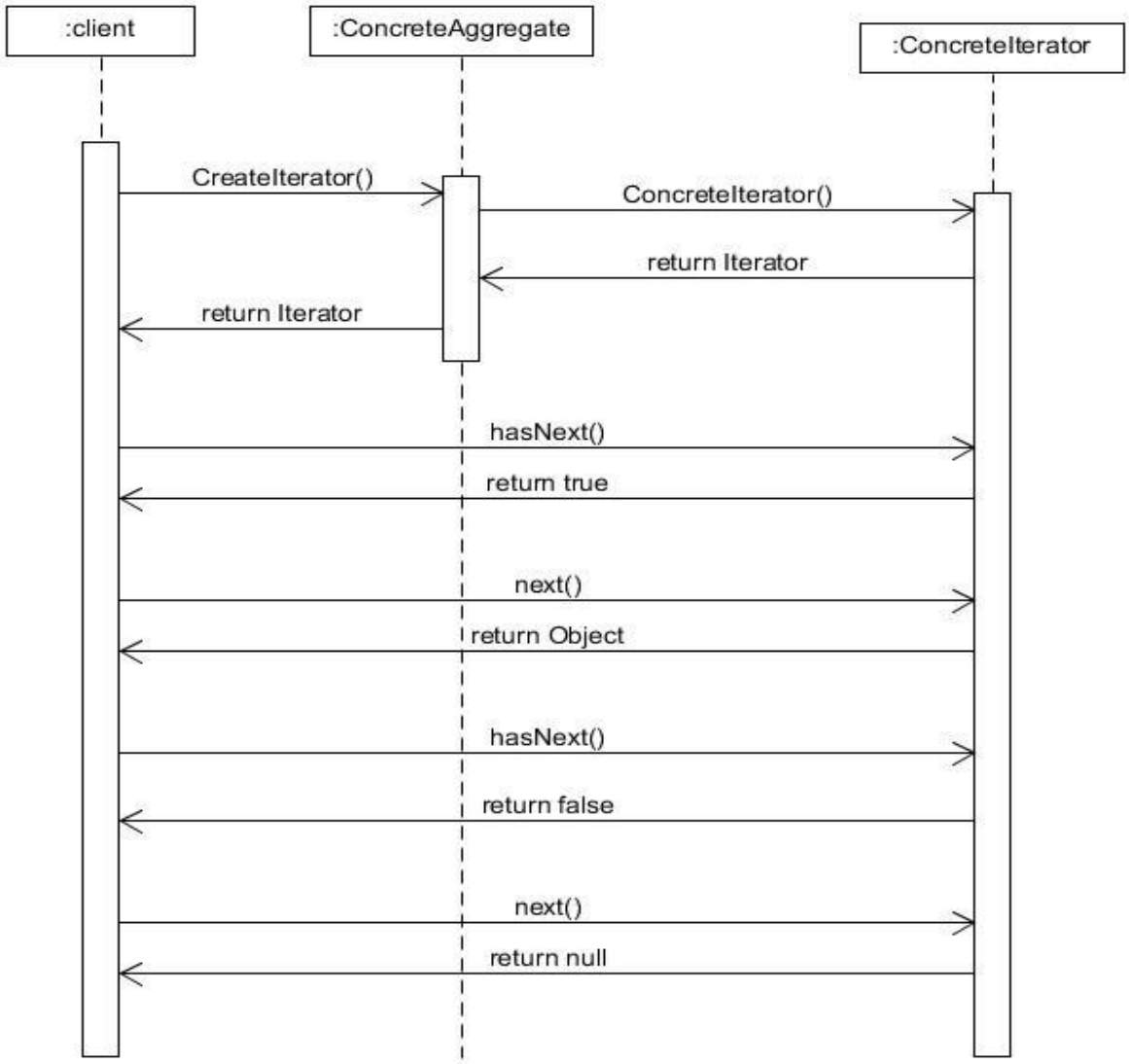
2. Iteration pattern:
   a. Description

Our game application contains multiple vectors each containing objects which are needed to make the game function correctly(for example bullets, aliens and barriers). We often need a specific object from the vector so often we have to iterate over the vector to find the specific object we are looking for. But instead of iterating over a vector we decided to implement an iterator which we use to iterate over. The reason for this is that the iterator pattern gives us a way to access the elements of the vector without exposing the underlying representation. The implementation in the code is simple we have two interfaces iterator and aggregator and two classes each implementing one interface ConcreteAggregator(which implements aggregator) and ConcreteIterator(which implements Iterator). The ConcreteAggregator allows the client classes to create the iterator. And the ConcreteIterator contains all the methods that are used to sequentially move through all the objects in the iterator.

b. Class diagram



c. Sequence diagram



Exercise 3

Reflection.

Eight weeks ago, we were assigned a (not so simple) task; create a game within two weeks time using java in teams of five. The first challenge that our team faced was the programing language itself, most of us was not familiar with Java or has ever created a game, we learned how to gather requirements and to delegate responsibilities, at the end of the first two weeks we had a playable version of our game with a well-structured code (or that is what we though at that time) with defined class per each element in the game and basic functionality, unfortunately we did not achieve to implement all the “must haves” described in the requirements document.

By our first iteration, one of the members of the team decided to drop the class, which changed the workload and team structure affecting for a moment the overall team performance, however we achieved to complete the task on time: extend the game implementation to support logging. We added a new class for this purpose and generated a file with all of the game events, which had to be changed after because of the size of the file and the difficulty to read it and distinguish the different types of events. As an additional feature we included the barriers into the game to keep fulfilling the general requirements and we started to implement more test cases. Our code was still readable and had a small amount of classes that were easily managed, however we struggle a few iterations to make the project a real maven project. We had not found our coding style yet (or at least we were coding still in our own way, instead of coding under the same style as a team), so despite our code was readable it was not as good and efficient as it could have been. This kind of situations are really typical when people who does not know each other from before and with different backgrounds (2 Erasmus students, 2 Dutch students) start working together. In a way we were mainly concern about making the code work within the time constraint.

By iteration two, we had clear that we should start writing cleaner code and that we should also start refactoring it; we wanted to follow as many good practices as we could. During this week the high score functionality was implemented and also the main menu screen, we keep having problems with the maven project (that were finally solved in this iteration) and the hurry to have something working caused that our code was more oriented to work than to be also pretty. We were also start realizing about a problem that will come up later on: We were writing most of the functionality into the game class, we tried to avoid it as much as possible but the initial structure of the code did not allow us to fix that problem right away. Instead we reduced the size of the methods by rewriting some of them and refactoring others.

By iteration three, we added new functionality to the code such as new types of aliens and we tried to add multiple levels, but this part did not belonged to the master code yet, this iteration we had to change the code to make it comply with design patters chosen by us, (singleton pattern for the log class and factory alien for the alien class), this make us change the structure of the whole alien class so, instead of just having one class to manage all the aliens, now we added the AlienFactory class, the Alien class and classes for each type of aliens, that made our code a bit more efficient and now with a complete and understandable structure for the alien creation, as well as a better delegation of functionality. We also create packages to give the classes a better organization having the main framework (which are the initial created classes) and the packages per each class.

By the iteration four, the levels were implemented into the master code, but they slightly changed from the previous iteration by applying the factory design pattern to them and adding the

functionality of having a boss level every five completed levels, so then, we had the LevelFactory class, the level class, the standardLevel and the bossLevel class to distribute the functionality among them. We also added a new type of alien called BossAlien to be match with the bossLevel. However this iteration we had to face the issue we already knew in advance, we had to fix the god class flaw in this iteration even if we had to rewrite the whole game, The Game class was a god class because it was extensively large, had a lot of (complex) methods and used a lot of external data. The purpose of the game class was to coordinate all the different aspects of the game but this got a little out of hand when everyone kept assigning it new responsibilities and more and more code was added to it every iteration without having any order. It was also a mistake to give it the responsibility of handling all the graphic aspects as well. This good class lead also to have s schizophrenic class but if we were able to fix one flaw, the other will be fixed too. The solution provided was take away a lot of responsibilities from the game class and create a new class called Screen to handle all the graphical aspects and key input. The main while loop and main function were taken out and put in a class Main. External data is not often used anymore, only a few get method calls remain. A lot of methods have been simplified or merged with similar methods and methods with responsibilities that could be done by another class have been moved. This made the game class shrink from 900 to 450 lines where about 100 of them are methods only used for testing. It also caused much better encapsulation and less complexity throughout the whole project being fully implemented and fully merged until the last iteration.

And by our last iteration before the final version, we decided to implement another two patterns (state pattern and iteration pattern) the implementation of the screen between levels and having one screen that will contain the menu, the game and the statistics would go smoothly. As a look back, the implementation of patterns helped a lot to improve our code and to make us realize that we did not have a good code structure since the beginning of the project, they helped us to delegate functionality between classes easily and forced us to write as clean code as possible. Definitely our code improved as so as our code skills by trying to add functionality using a pattern and at the same time trying to fix previous issues.