Slide 1

For my final project, I have decided to simulate a hockey database. In my project there are 3 parties, the individual teams in the league, hockey players and the league administration. The league administration is responsible to handle the transactions occurring in the league, such as new players coming in, leaving or trades. The league administration is represented by the Mediator class in my project. The individual teams in the league are represented by the Team class in my project and are responsible for notifying the league when a trade among them happens. The hockey players are represented by the forward, defense and goalie class in my project, which all inherit from the HockeyPlayer abstract class. Other classes/interfaces that I have in my project are State interface, Active and RetireState classes, which are responsible for implementing the State design pattern. I also have the Strategy interface, which is responsible for the Strategy design pattern, and the Main class, where I run my project.

Slide 2

For my new design pattern, I have decided to use the mediator design pattern. First, I will go over the mediator pattern and then I will go over how and why I have decided to implement it in my project.

So, what is the mediator pattern? The Mediator in java is a behavioral design pattern that reduces the coupling between different classes. Especially in bigger projects and with the use of Object-oriented programing, the developers have to deal with a complex set of dependent objects. The goal of the mediator pattern is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another. The Mediator pattern achieves that by creating a Mediator object, which is responsible for all the communication between the dependent objects, promoting loose coupling between the two dependent objects, as they now only refer to the Mediator object.

Some other pros of using the mediator pattern, apart from the lose coupling, include the Open/closed principle. This means that the developer can introduce new mediators, without having to change the actual components. The developer can reuse individual components more easily, which ties back into lose coupling. Another pro is that the mediator pattern promotes the Single Responsibility principle, which means that all the communications between various components happens in one place, which makes it easier to understand and maintain.

The con of using a mediator class is that with bigger projects, the mediator class can turn into a God Object, which means that it will become extremely large and hard to understand and maintain.

Slide 3

Now I am going to talk about how I implemented the Mediator pattern in my project. On the slide you can see a UML diagram representation of the Mediator pattern in my project. I have an abstract HockeyPlayer class, which serves as a parent class for the different hockey players. The Forward, Defence and Goalie class all inherit from the HockeyPlayer class. Then I have a Team class, which is representing the different team objects in the league. The Mediator class in my project represents the league administration. The Mediator class serves as a mediator between the Team and the different player classes. The mediator class contains an ArrayList of hockey players, which is representing the database of the league and methods to add player to a league, remove the player from the league, print the content of the database to the console and update the team for different types of hockey players. Therefore whenever the teams decide to make a trade between themselves, the mediator updateTeamPlayer() gets called and the player classes and the team classes don’t know about each other, as they are loosely coupled and all the work is done by the mediator.

Slide 4

For my project I have implemented more than one design pattern. Here on this slide, is the list of other patterns I have used for my project.

I have used the singleton design pattern in my project, to produce only one instance of the mediator. The singleton design pattern restricts the instantiation of a class to a singular instance. I have used in my project because I wanted there to be only one representation of the league administration, or the mediator object.

I have also used the Strategy pattern in my project. I have created a Strategy interface that has a method of getStats(). The getStats() method is then overridden by the individual hockey player classes, as the stats differ for goalies, defenceman and forwards in hockey.

Another Pattern I have used was the Factory Pattern. In my project I have an abstract class HockeyPlayer, that is a parent class to the Forward, Defence and Goalie classes, and is responsible for the factory pattern by using inheritance.

The last design pattern I have used in my project was the State design pattern. The two states of hockey player, that I have implemented were active state, which is when the player becomes active in the league, and the retired state, which is when the hockey player retires from the league. I have created a State interface, which contains the sendUpdate(). The Active and RetireState classes override this method to represent the state of the hockey player. I then created instances of the different states and put them in to the addPlayer() and removePlayer() in the Mediator class.

Slide 5

Now I am going to show you my code for the project, so everyone knows what I have been talking about.