Use of Patterns in the Design

Throughout implementing this project (the game of Ricochet Robots) we have used several different patterns that helped us to ease the process of design and improve the quality of the code. The patterns used and how they are used is described below:

- Information Expert: this pattern has been implemented throughout our coed in different classes. For example, the “Player” class in our code is in charge of the creation of players as objects. Each player (object of this class) had methods that are unique to them and by invoking those methods, these objects are able to hold and provide all the necessary information to fulfil the responsibility that is assigned to them. In other words, instead of having an object which does not have enough information needed to fulfil its responsibility, we assign the special responsibility to the proper object capable of accomplishing them.
- Cohesion: our design includes several classes where each class is focused in only one responsibility. While the classes maintain a high quality relationship with each other, they are separately in charge of accomplishing their own responsibility. Preservation of the high cohesion in our code gives us the opportunity of re-use and easier maintenance of our code.
- Controller: to handle input system events efficiently, we have implemented a controller class in our code which is a one (façade) controller. The “Window” class in our code does not do the work, but it is responsible for receiving system event messages and assign it to other objects of other classes to fulfil their responsibility with respect to the specific system event.
- Polymorphism: in our code there are two classes that will generate objects with the same behaviour. Both “player” objects and “AI (simple-smart)” objects do have the same behaviour since logically both of them represent a player behaviour. In order to handle the alternative behaviour that is caused by different object types, we used polymorphic operations where both of these classes are subclasses of a bigger class “Player”, where mutual methods such as “name”, “make a move”, or “make a bid”, will assign the responsibility for the different behaviours to the types themselves.
- Protected Variables: to get the required information for a method, sometimes we need to seek other methods form other classes. Instead of having a method that consists of other methods to find required information (which is fragile since is needs to traverse a path) we use new public operation in the required class to provide information but hides the process in which the information has been provided.