To implement the game and its domains we have used the GRASP design patterns. At creation of the Table which includes the squares and the cards required we have used Creator, Information Expert and Polymorphism and provide Low-Coupling and High Modularity. We also used Controller and Observer design pattern at implementing GUI. You can find a table of problems and alternative design patterns with their cons and pros.

|  |  |  |
| --- | --- | --- |
| Square creation  Or  Card creation | Creator, | Pros:  -Readable & Maintainable Code  -Easy to understand  -Low Coupling is supported  -Code reusability  -High cohesion  Cons:  -Some extra classes may prevent understanding the code  -Some extra words of code for the new classes.  Alternative:  -We can implement all creations at the Class Table,  However this is hard to implement and maintain, because the code would have a bad cohesion with bunch of lines, which would create likely high coupling between objects. Therefore we followed Controller Design Pattern. |
| Table implementation | We used Polymorphism and different Creator’s for different parts of the Game. | Pros:  -High cohesion  -Low coupling since the code has high cohesion.  -Easier implementation  - Readable & Maintainable Code  -Easy to share work within the team since we have a lot of low-coupled blocks with some basic opeartions.  Cons:  -More code and class  -Might be got lost in classes after some hour of coding.  Alternative:  -Doing all in a java file of Table, where you have every line of coding needed for the domains. If one can do that probably It would be more efficient in terms of code length, however it is not practicable in a team project since team-work is needed to be practiced and it is quite hard to work on a single file. At the same time one can get lost in such a code easily. Also one needed to do a lot of scrolling down and up within the file, which we all hate. |
| GUI implemantation | We used Controller and Observer Design Pattern | -In the design phase, two of the design patterns we used are Controller Pattern and Observer Pattern and we can discuss the advantages and disadvantages of these patterns. With the controller pattern, we gain the ability to separate UI and Domain level in an efficient way. The classes in UI level interact only with our controller class. This helps us for not bothering UI classes with “game logic”. The game rules are implemented in Domain Level and UI level classes can still work if the Domain level classes are implemented differently. This also helps us to test our Domain classes more easily. The disadvantages of Controller Pattern can be summarized as it has higher complexity, it is harder to find the right implementation and it is not suitable for small solutions. The alternative to the Controller Pattern a direct access to all classes in Domain level from UI level classes. But design-wise that is not a smart decision.  -About the Observer Pattern, the first advantage is that we can add new subjects and observers in an independent way from each other. It also provides minimal dependencies between subject and observer. Also subject doesn’t need to know the number of observers it has, and it can send a signal to every observer at once to make many changes. About the disadvantages, one of them is that because observers don’t know each other, we can create really long chains to update something. Thus, it can add an unnecessary complexity to our program. The alternative to the Observer Pattern can be, again a direct access to all classes in Domain level from UI level. |
| getSquare | Information Expert pattern implemented in Board and this function is used for any kind of direct access of Squares | Pros:  -Low coupling since any direct retrieval of Squares called by any other function should be via Board.getSquare. This allows us to reduce couplings and provide more sound system.  -These patterns serves as an important building block for the rest of the sytsem and therefore reduces the overall code length.  Cons:  -  Alternative:  -Implementing whenever we need it with HashMap searching, which doesn’t have the pros indicated above. |
| Square implementation | We utilize Polymorphizm to group similar Square’s into classes. | Pros:  -Easy to implement.  -Less code overall  -easier implementation at the rest of the system since we have all the basic abstraction with arrival\_proc, passing\_proc and next\_sq functions, which may be overridden if needed by the sub-classes.  -Easy to extend for further Squares  -Maintainable code  Cons:  -It was hard to decide the necessary classes and the polymorphism they obey. However we have Use-Cases and we can go over them to come up with an efficient implementation.  Alternative:  -We did this differently in the Prototype phase with only one class named Square and use type id’s for different class of squares. However this type of implementation was not meaningful since we had so many different squares and we needed to come up with an better abstraction |
| Moving on the board | High Cohesion provided via “Linked List” like abstraction. | Pros:  -Easy to understand  -Moving is so easy  -Easier to implement and debug.  -Provides an easy implementation for the Train Stations and reverse moving procedures.  -Provides an easy implementation for passing-triggered actions with the passing\_proc field.  -Less arithmetic during the game progress.  Cons:  -Harder to create and connect.  -More initial arithmetic.  Alternative:  -Implementing Square’s with 3 arrays and using bunch of different variables to hold the moving. This also requires a sound arithmetic at moving. Our design choices is easier and more reasonable and maintainable for our problem. So we chose that one. |
| Actions on arrival and passing\_procedures |  | Pros:  -Increased Modularity  -Easier to implement and debug.  -Provides an easy implementation for moving and squares actions, so we are able to write simple and generic code with these abstractions.  Cons:  -More initial difficulty.  -Harder to code  Alternative:  -Implementing these actions with different functions for each square would be much harder than our implementation choice, because such an implementation does not create the important building blocks we have for the similar squares like Deed Square. Around 30 deed square is implemented with only 1 arrival and passing proc. |