SWEN30006 Project 2 Report

Ethan Hawkins 1355842

William Spongberg 1354263

Joshua Linehan 1173490

1 Analysis of Current Design

The current design of LuckyThirdteen is highly problematic and in violation of multiple GRASP principles. While the Driver and PropertiesLoader present examples of highly Cohesive and lowly oupled classes, the LuckyThirdteen class handles the rest of the game's logic. In its original form, the responsibilities of LuckyThirdteen include defining the behaviour of the game, creating and controlling the players, outputting log information, and calculating scores. This plethora of responsibilities makes it hard to define simply what the role of LuckyThirdteen is, therefore it has low cohesion.

Furthermore, the intertwined responsibilities of the class make it difficult to extend or adapt functions of the game: code reuse is difficult when the methods serve a specific purpose. The current design escapes violation of some GRASP principles by over-assigning responsibility to LuckyThirdteen - for example, LuckyThirdteen creates the players because it contains them, which is consistent with the Creator principle, however it doesn't allow protected variation and doesn't utilise indirection or pure fabrication. As such, a revised version of the design should delegate the responsibilities of the current implementation of LuckyThirdteen to new classes in accordance with GRASP principles and responsibility driven design.

2 Justification of Extended Version

2.1 Scorer

The scorer package is responsible for handling the scoring logic and the game's access to it. The façade pattern was used to design how the LuckyThirdteen class interacts with the scorer package. Many different classes were used to handle the game's scoring logic, and having LuckyThirdteen interact with each of them would have resulted in high coupling. The façade pattern reduces this coupling by using the Scorer class as a single point of contact for LuckyThirdteen and having all scoring operations run through it.

The summingOptions subpackage utilises the template pattern. A template is created in the form of the SummingOption abstract class. Here, methods that are common to all subclasses of SummingOption are implemented, namely the isThirteen method. Additionally, coupling is avoided between Scorer and the subclasses of SummingOption by including a static getter in SummingOption. The template, SummingOption, also specifies abstract methods that are overwritten by the subclasses which define the behaviour of each subclass. These are used to represent option 1, option 2, and option 3 from the spec, and they override the hasThirteen and case3Score methods to implement the functionality of their respective option. The result is three highly Cohesive classes which are not coupled to the Scorer class. The use of the template pattern also provides protected variation, in that additional summing options could be implemented in the future by extending the SummingOption class, and no changes would need to be made to the Scorer class.

The scoringCases package also utilises the template pattern, in that an abstract score method is defined in the ScoringCase class, which is inherited by Case1, Case2, and Case3 respectively. This provides the same benefits as before; the ScoringCases are decoupled from

Scorer, the scoring cases are highly cohesive and lowly coupled, and protected variation is exhibited in that additional ScoringCases can be easily implemented.

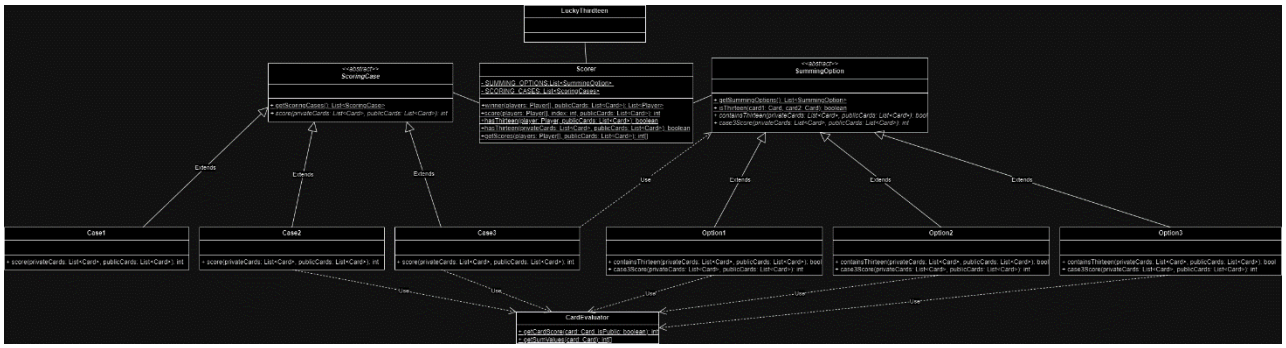


Figure 1 – Design class diagram for the scorer, demonstrating facade pattern structure. It succinctly communicates functionality to the developer through proven pattern usage.

2.2 PLAYER

The player package is responsible for creating and implementing the logic of the players of the game.

The responsibility of handling player information from the properties file was delegated to the PlayerCreator class due to a need for Pure Fabrication and to increase the Cohesion of LuckyThirdteen. The Creator principle also applies to this class since it has the initialising data for Player objects, is assigned responsibility for their creation.

The Player class and its controllers implement the Strategy pattern. The four different player types as defined by the specification differ only in how they choose which card to discard. As such, the rest of the code for players is written in the Player class, and a template PlayerController is defined, where the Player class has an attribute of PlayerController type. The PlayerController interface defines only the discardCard method. The type of player defined by the properties file is implemented as an implementer of the PlayerController interface, where that player type's method for choosing which card to discard is defined in the discardCard function. The type of PlayerController instantiated is determined in the PlayerFactory, and the PlayerController object is passed into the Player's constructor. There are many advantages to the use of the Strategy pattern in this instance, for one it provides protected variation and extendibility as any additional player types can be easily implemented by implementing the PlayerController interface. The Strategy patten also promotes low coupling, as the PlayerControllers are not coupled to Player as they are only accessed through the interface. The PlayerControllers are highly Cohesive, as their sole responsibility is to implement a Strategy for discarding a card. An additional benefit to using the Strategy pattern is that the individual strategies implement an interface instead of extending a template class which means they are not forced to have a specific structure: the Clever class stores the public cards as an attribute whereas the other PlayerControllers don't and the Human PlayerController facilitates mouse controls.

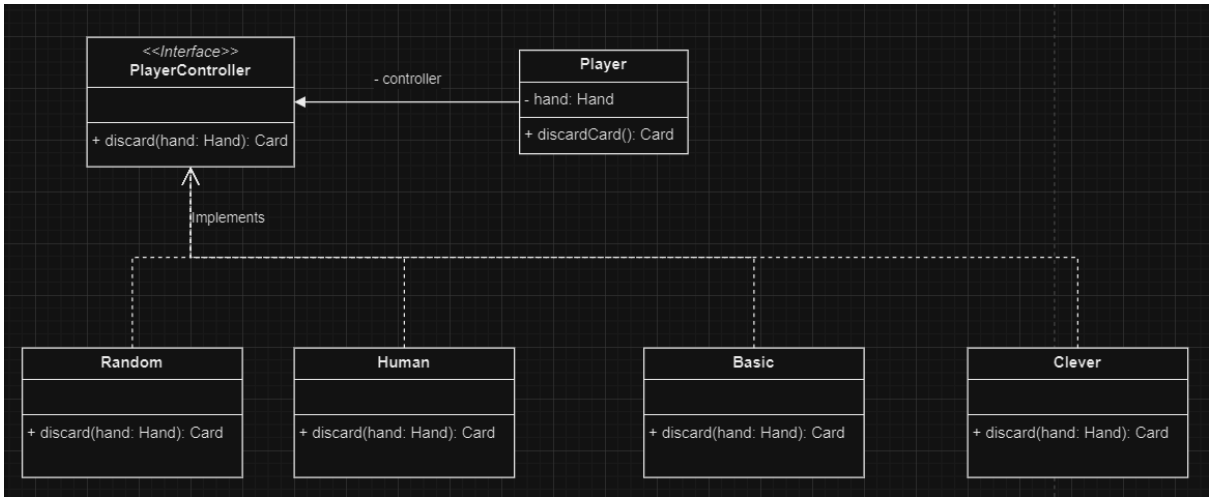


Figure 2 – Design class diagram of the player package

2.3 LOGGER

The skeleton code provides four functions for logging behaviour: “addCardPlayedToLog”, “addRoundInfoToLog”, “addEndOfRoundToLog”, “addEndOfGameToLog”. It was noticed that these four functions can clearly be divided into states of the game, namely, start of round, end of turn, end of round, and end of game. As such, it made sense to implement the state pattern. When the state of the game changes, the behaviour of the logger should correspond to the change to log what is required as per the functions. However, the key flaw of this behavioural design pattern is the implication on the GRASP principles. In its raw form, a state pattern can result in monstrous conditionals and high coupling to the class that controls the state centrally, in this case, the LuckyThirdteen class. This means we lack Indirection, as there is nothing to mediate between the logger and Lucky Thirteen, resulting in reasonably high coupling and no design benefit. As such, the state behavioural pattern was combined with the Observer pattern such that there is an entity to mediate and be sensitive to state changes.

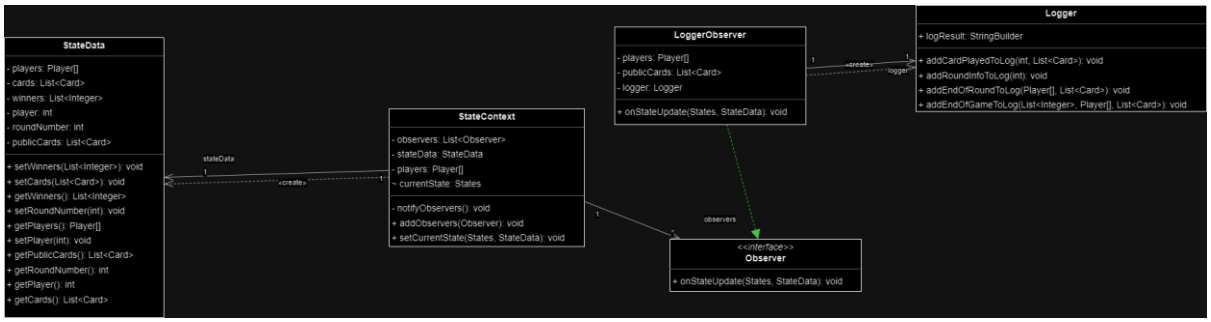


Figure 3 – the state + observer pattern to operate logging.

In the above figure, which illustrates the observer + state pattern, StateData is used to carry data between the different states. StateContext creates the data, and stateContext has stateData. The main program, LuckyThirdteen, simply calls the setCurrentState() and passes an enum corresponding to the state. This notifies all observers through the Observer interface, thus allowing the observers to call the necessary methods of the instance being observed.

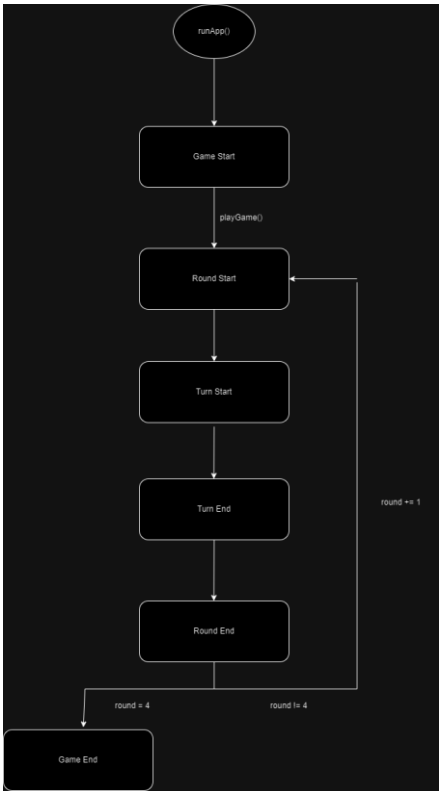


Figure 4 – State machine diagram of the game that provides justification for the implementation of state pattern, as the state iteratively changes, and different methods must occur in response to that

This has some advantages for GRASP principles. Firstly, it means there is far less coupling between objects responsible for behaviours in the game related to state and the LuckyThirdteen class. Now, StateContext simply manages the execution of all state-based methods through A list of observers responding to the change. This is opposed to entities like logger being directly tied to LuckyThirdteen. Additionally, the Observer interface generalises this and prevents classes having to be coupled to StateContext. They are simply all called through notifyObservers. It also promotes cohesion, as by making objects which are tied to state abstracted through stateContext, they are given focus and purpose; all the methods of the class will be cognizant of when it is used. Additionally, protected variation is achieved through the Observer interface, as it provides a stable point from which to make changes and additions to the game.

This has another key advantage: it makes the program incredibly extensible to adding new behaviors that are state based. To implement a new behavior in the game, a developer would simply need to create a new observer that instantiates a class that contains the methods related to the behavior, and using a conditional statement in the interface method, running those functions depending on what the state of the game is. For example, if a developer wanted to add special effects to Lucky Thirteen depending on the state of the game like fireworks shooting at the end of the round, they would simply have a class containing the methods for executing the special effects, SpecialEffects(), and then in the interface method onStateUpdate, make SpecialEffects.shootFireworks() on the condition that state is equal to end of round. The developer is completed abstracted from the LuckyThirdteen class.

It was thought that the UI could follow a similar pattern, as UI changes are directly tied to the state also, however, because actors in the source package JCardGame must be directly

tied to the instance of the game they are defined within to be used it was impossible. Passing the LuckyThirteen instance of JCardGame around would result in a copied reference, and therefore the actors will not display, unless LuckyThirdteen is made static. If a different framework were adopted, this could have been implemented.

2.4 DISCARD PILE

The singleton pattern was used for the discard pile. This was important to do for extensibility, as future player types may also need to be able to see the discard pile, which was defined as visible to all players in the specification. Singleton allows for controlled access to a single instance of a class. The benefit of singleton pattern here is that we have safe global access, as there is still only one instance with a global point of entry; there is controlled and debug friendly use of the instance. Because the singleton pattern ensures that the instance is only being modified through a single point of entry, we know the instance won't ever be overwritten, which allows us to guarantee the discard pile is consistent across all classes. Otherwise, we would end up passing the discard pile around through method calls, which would increase coupling. The singleton discard pile is both purposeful and easy to use for future development.

2.5 DEALER

It was decided that a dealer class should be implemented for single responsibility principal purposes. Dealer adheres to the facade design pattern. It abstracts away the interaction of the complex framework that is the JGameGrid package and its extension JCardGame by ensuring that the Dealer methods are the only means by which we interact with the deck. This ensures future developers don't have to decompile the package to understand how to develop further and can instead rely on the functionality of the Dealer class. All its methods are executed within LuckyThirdteen as it is intrinsically tied to the gameplay, however, despite the coupling the abstraction of these methods increases cohesion dramatically, as the calls of dealer methods such as getCard() can be easily understood. This fulfils the GRASP principle of controller and indirection as the Dealer handles the obtaining and discarding of cards from a player's hand, controlling that behaviour, and it mediates between the main game class and the framework classes of Deck and Hand.

2.6 LUCKYTHIRDTEEN

LuckyThirdteen now exclusively runs the game and renders UI. It is difficult to abstract any further features, and whilst LuckyThirdteen still has some coupling, the cohesion is high now that classes with specific purposes exist. Due to the nature of the framework, some level of coupling will always exist, as entities defined in the game (which is the LuckyThirdteen instance created by driver) need to be modified within the game in which they are declared to be updated reliably. This is seen as an insignificant problem however, as the modification of such instances like player and score are now done through classes like Scorer and PlayerFactory. It is debatable whether it is possible for LuckyThirdteen to have single responsibility principle successfully satisfied, as it must initialise, run the game logic, and draw the UI due to the framework, however, it has as few responsibilities as is possible in our solution.

The initGame() function has been modified to utilise the logger observer to generate logging behaviour, playerFactory to make players, and Dealer to set up hands whilst also still being responsible for drawing the UI. This ensures that whilst it must still initialise these behaviours, they are delegated in a way that is extensible without modification to the main

class. Logging behaviour can be changed, and players can be added without substantial modification to the main game runner, making the responsibility of the LuckyThirdteen class clearer, and the relationships to other classes distinct. Because change in these classes has a low impact on LuckyThirdteen, the coupling is extremely low, and the cohesion is very high. PlayGame() also delegates functionality such as all the logging functions to logging through state declaration. The usage of state currently is underutilised, but it enhances cohesion as it defines the focus of LuckyThirdteen more clearly; to execute functionality through delegation depending on the stage the game is at. Ultimately, whilst LuckyThirdteen is still a sizeable class with many instance variables, our solution delegates much of its functionality and logic to GoF pattern implementations such as the facade of Scorer, making it clearly defined and extensible, as code bloat has been mitigated and responsibilities appropriately delegated to individual classes.

2.7 CONCLUDING BENEFIT ANALYSIS

Our implementation uses extensive yet clearly purposeful GoF software design patterns as discussed. The key benefit is the maintainability and extensibility of the card game. The usage of Facade provides clear ability to add more card game strategies and scoring, the factory allows for additional players and player types, the state allows for additional logging behaviour and implementation of new state dependent behaviours, and singletons provide a safe global point of access. The key feature of all of these is that all issues with code are now extensively diagnosable and the solution provides a clear foundation of which to build. Furthermore, the proven nature of the GoF patterns means that the design class diagram follows clear patterns that are easy to observe, allowing future new developers to quickly understand the solution in a way that a non-GoF implementation would not. Low coupling and high cohesion have thus been successfully achieved by the solution.

3 CLEVER PLAYER IMPLEMENTATION

The clever player takes advantage of all the information available in the game to any human player. The minimax algorithm used in AI, specifically the score maximising component, was the inspiration for our design, however, the algorithm is exclusively evaluation based, as simulating opponent movement was too computationally expensive. The following diagram illustrates the logic flow:

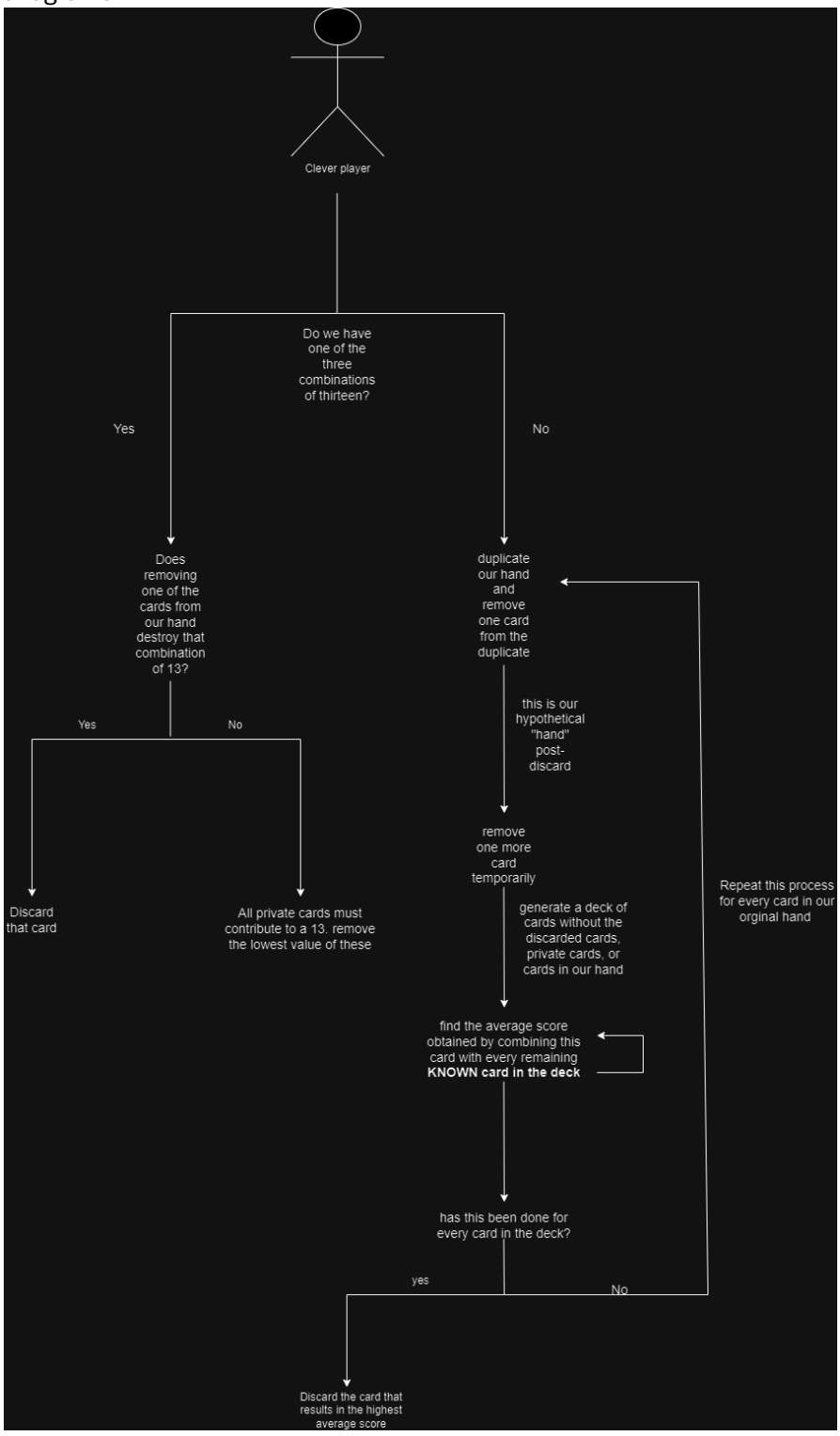


Figure 5 – the flow of logic in the clever player's algorithm.

The design makes use of all visible public cards and all discarded cards, which are visible to all players. The design was done with the aim of optimising the thirteen cases, and the best scoring thirteen case at that. This was done by holding onto our thirteens in the case we check hasThirteen(), which is a static method used to execute all thirteen counting options abstractly without coupling, and it returns true. In such a case, we find what card doesn't contribute to the thirteen combination that exists between our hand and public card by comparing all pairs in our hand of three cards at the start of the round and discard that card. In the case that we don't find a card that doesn't contribute to thirteen, that is, hasThirteen() returns true for every combination, we adopt the basic player logic of removing the lowest scoring card. **This ensures that on average, we always perform better than the basic player logic.**

In the case we don't have thirteen however, clever calculates the average score returned by every card in our hand should it be combined with every card in the deck minus the known discarded cards, known public cards, and known private cards. This ensures we pick a card to discard that has the mostly likely increase in performance.

Additional performance in choosing the correct card to make a high score could come from recursively playing simulated rounds with the depth corresponding to the number of turns left in the game, however, upon implementation of this the computational cost was far too high, resulting in extremely slow games when it came time for the clever player to play. Future implementations could seek to include this in a more optimised way.

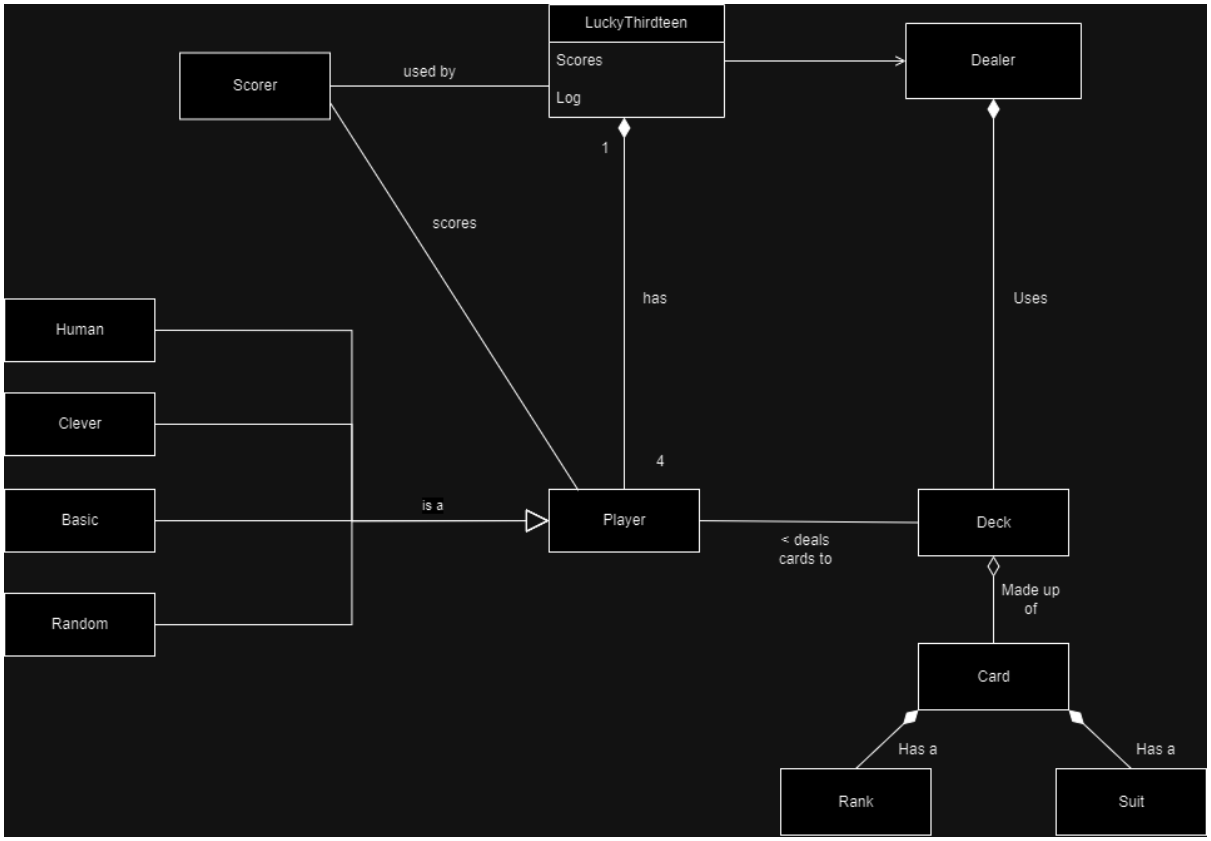
To illustrate its performance, the below table is provided. The seed was set as random, and no automatic moves or starting cards were defined. The winner is in green.

Test Number	Player 0 & score	Player 1 & score	Player 2 & score	Player 3 & score
1	Basic: 17	Random: 33	Random: 28	**Clever: 43**
2	**Basic:40**	Random:0	Random:30	Clever:26
3	Basic:0	Random:36	Random:40	**Clever:46**
4	Basic:0	Random:33	Random:0	**Clever:40**
5	Random:0	Random:14	Basic:0	**Clever:41**
6	**Random:48**	Random:0	Basic:17	Clever:15
7	Random:28	Random:34	**Basic:48**	Clever:31
8	Random:0	Random:0	Basic:0	**Clever:100**

Figure 6 – table of test results in evaluation of Clever player performance. It performs on average better than the other two automated player types. It only lost when Basic and Random got very high scores when others had 13, indicating it was beaten on pure chance

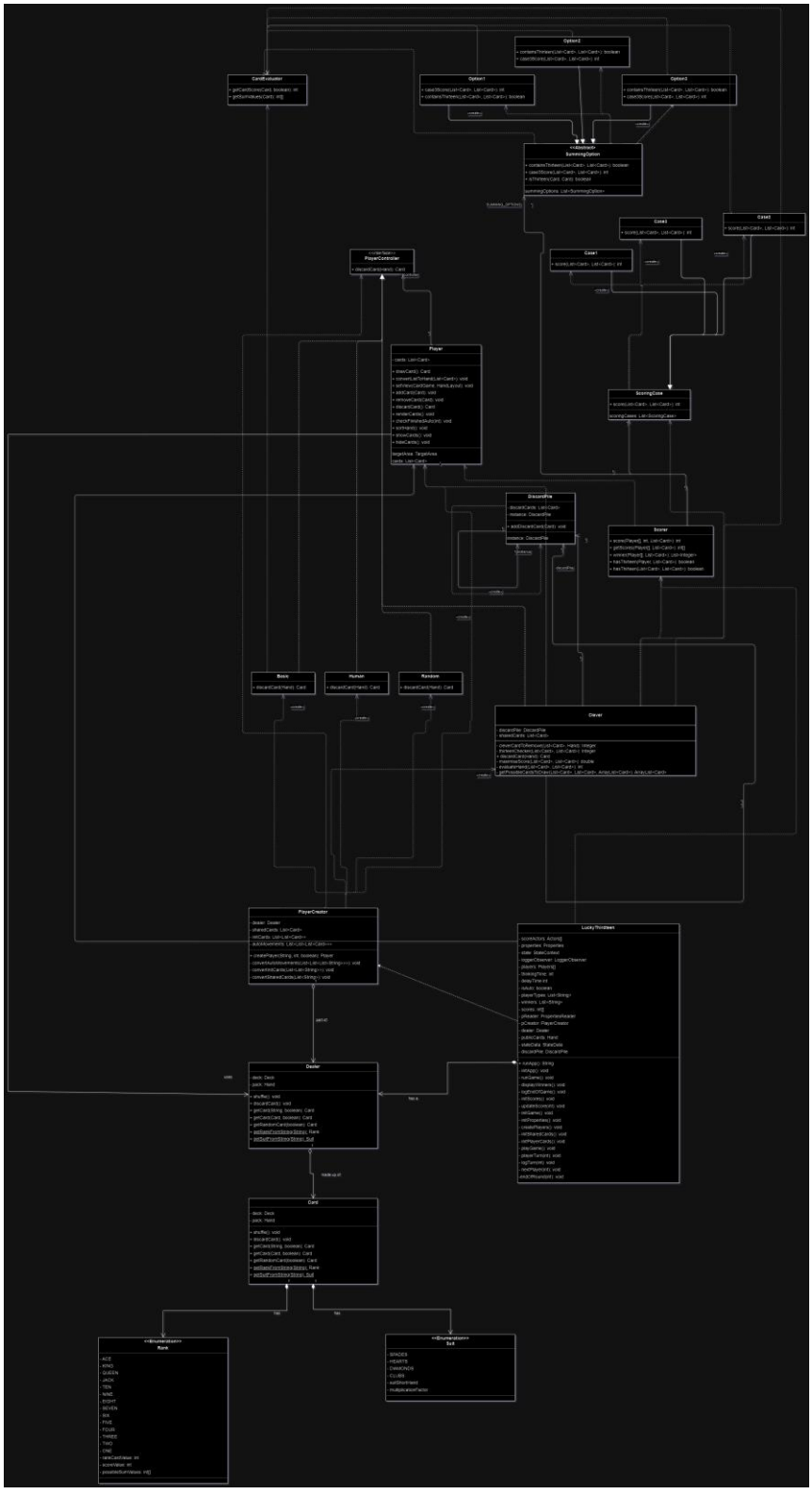
4 SOFTWARE MODELS

4.1 DOMAIN CLASS DIAGRAM



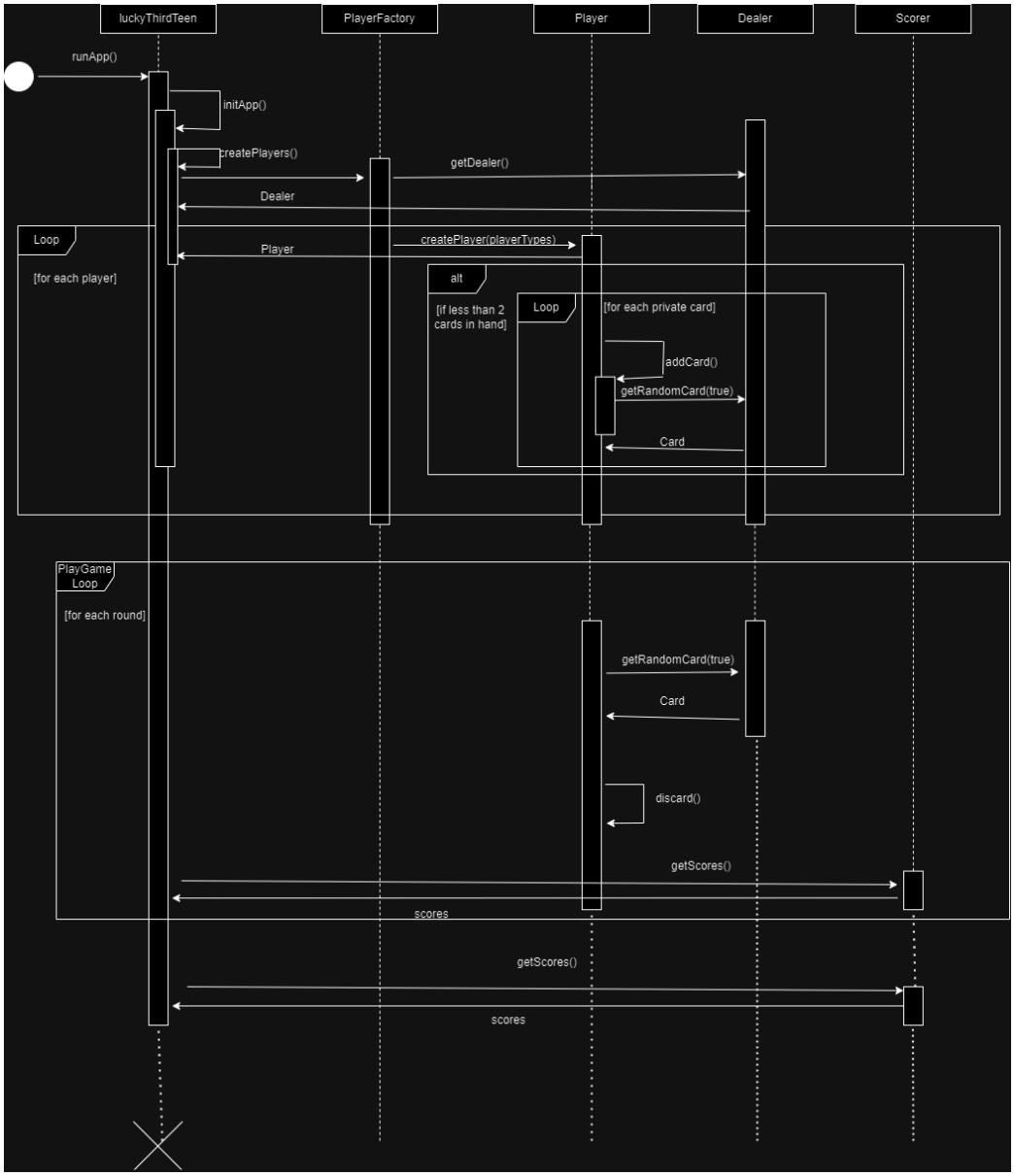
4.2 DESIGN CLASS DIAGRAM

Certain methods and fields have been left out to provide clarity around the player functionality exclusively:



4.3 DESIGN SEQUENCE DIAGRAM

Player functionality is demonstrated through this design sequence diagram, as it is the most pivotal feature within the game.


luckyThirdTeen
PlayerFactory
Player
Dealer
Scorer
runApp()
initApp()
createPlayers()
getDealer()
Dealer
Loop
[for each player]
createPlayer(playerTypes)
Player
alt
[if less than 2 cards in hand]
Loop
[for each private card]
addCard()
getRandomCard(true)
Card
PlayGame Loop
[for each round]
getRandomCard(true)
Card
discard()
getScores()
scores
getScores()
scores