COS20007-Object Oriented Progamming

Custom Program Documentation

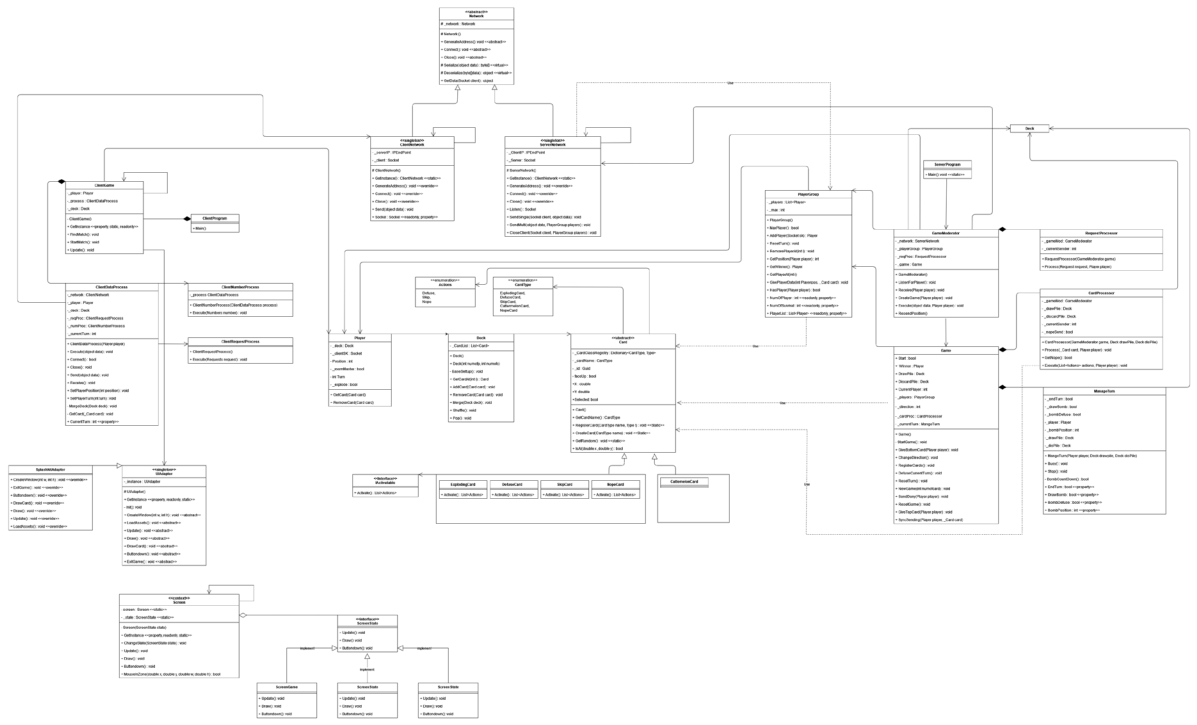
Exploding Kittens

Last update: October 21, 2021

Github Repos: https://github.com/Jiruu246/Exploding_Kittens.git

UML (for the latest update please do check out the files in the Documentation folder)

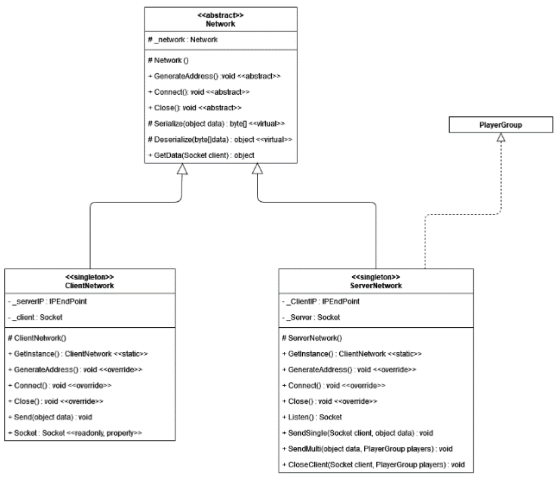
How it works:

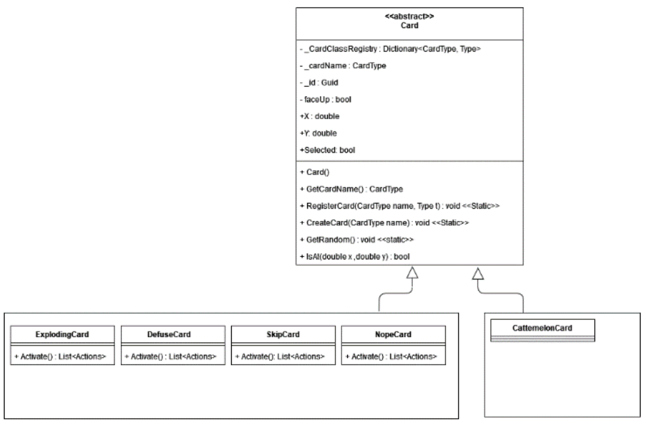


There are two main components: Server and Client, they communicate by sending object data throughout the network. The Server will responsible for creating a player object for each client who connects to it, managing the game based on the game rules, listening for player requests, and telling the player what to do. The Client will be responsible for helping the actual player interact with the game using CLI or GUI. It tells the player what they got or what they need to do and sends the player actions/data to the Server.

This Program is based on a board game called “Exploding Kittens”. The game starts with two or more players each will draw a card from the draw pile, if someone draws an “exploding card” they will be eliminated, the winner is the last one who survives. Other cards that are not “exploding card” will help the player reduce the chance they got “Explode”.

Inheritance and Polymorphism


<<abstract>>
Network
# _network : Network
# Network ()
+ GenerateAddress() :void <<abstract>>
+ Connect(): void <<abstract>>
+ Close(): void <<abstract>>
# Serialize(object data) : byte[] <<virtual>>
# Deserialize(byte[]data) : object <<virtual>>
+ GetData(Socket client) : object
PlayerGroup
<<singleton>>
ClientNetwork
- _serverIP : IPEndPoint
- _client : Socket
# ClientNetwork()
+ GetInstance() : ClientNetwork <<static>>
+ GenerateAddress() : void <<override>>
+ Connect() : void <<override>>
+ Close() : void <<override>>
+ Send(object data) : void
+ Socket : Socket <<readonly, property>>
<<singleton>>
ServerNetwork
- _ClientIP : IPEndPoint
- _Server : Socket
# ServerNetwork()
+ GetInstance() : ClientNetwork <<static>>
+ GenerateAddress() : void <<override>>
+ Connect() : void <<override>>
+ Close() : void <<override>>
+ Listen() : Socket
+ SendSingle(Socket client, object data) : void
+ SendMulti(object data, PlayerGroup players) : void
+ CloseClient(Socket client, PlayerGroup players) : void


<<abstract>>
Card
- _CardClassRegistry : Dictionary<CardType, Type>
- _cardName : CardType
- _id : Guid
- faceUp : bool
+X : double
+Y: double
+Selected: bool
+ Card()
+ GetCardName() : CardType
+ RegisterCard(CardType name, Type t) : void <<Static>>
+ CreateCard(CardType name) : void <<Static>>
+ GetRandom() : void <<static>>
+ IsAt(double x ,double y) : bool
ExplodingCard
+ Activate() : List<Actions>
DefuseCard
+ Activate() : List<Actions>
SkipCard
+ Activate(): List<Actions>
NopeCard
+ Activate() : List<Actions>
CattemelonCard

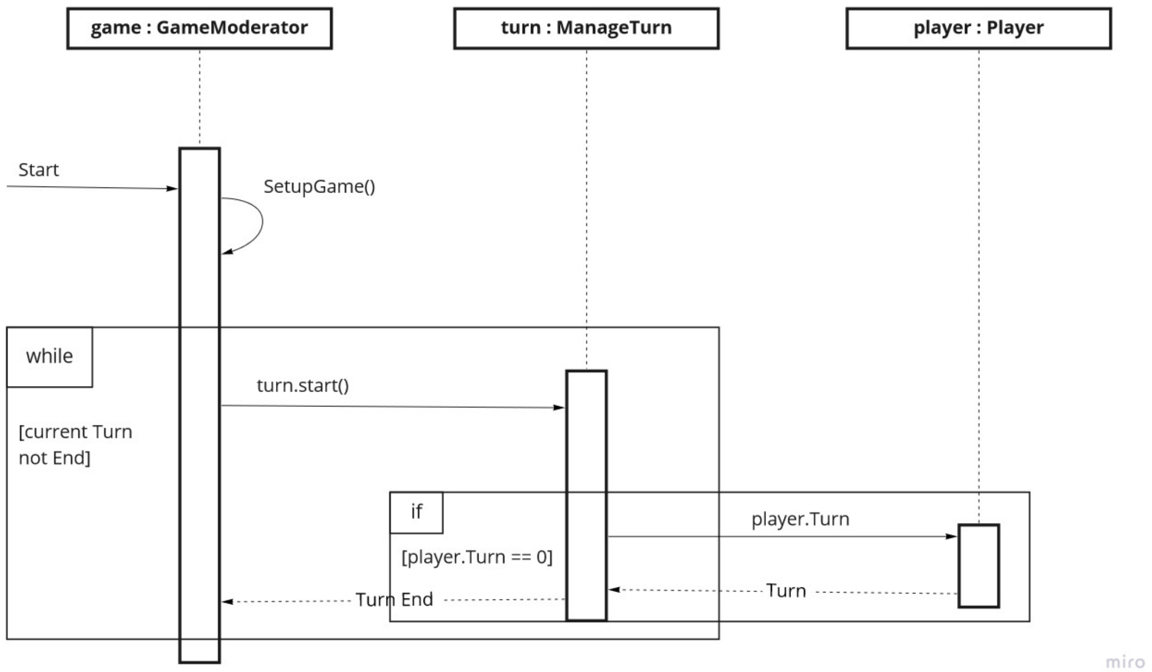
Two significant inheritance is the Network class and Card Class:

+ The Network Class is responsible for transferring data between clients and server. It has 2 children: ClientNetwork and ServerNetwork

+ The Card Class represents a single card in the game. It has many child classes, each is a unique type of card with different actions.

Main procedure

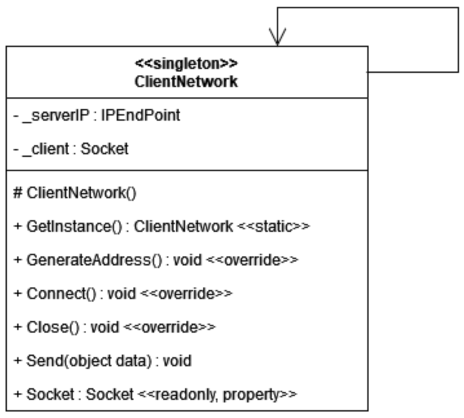
The main class that control the game logic and rules is the GameModerator Class



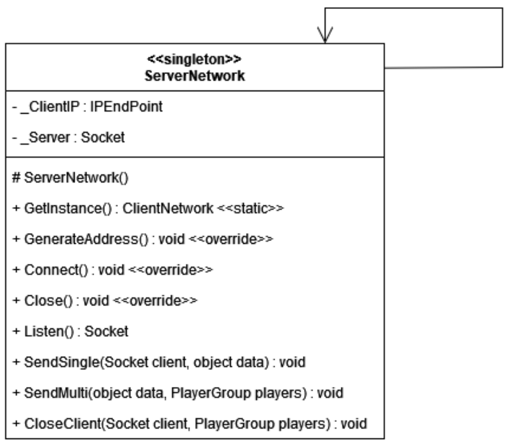
When the player sends a signal to start the game it starts the loop. First, it loops through each player and checks for their turn, while the player's turn does not end it will keep waiting for the player to send the "draw" request. During this time, the player can play any card that they have. After the player sends the draw request, their turn will end and that player will receive a card from the draw pile. If they draw an exploding card, they will be exploded and eliminated from the game.

Design Patterns

1. Singleton pattern
   Networking Classes use the Singleton pattern, as there should only be one network object to manage connection (each belongs to the corresponding project Client and Server).

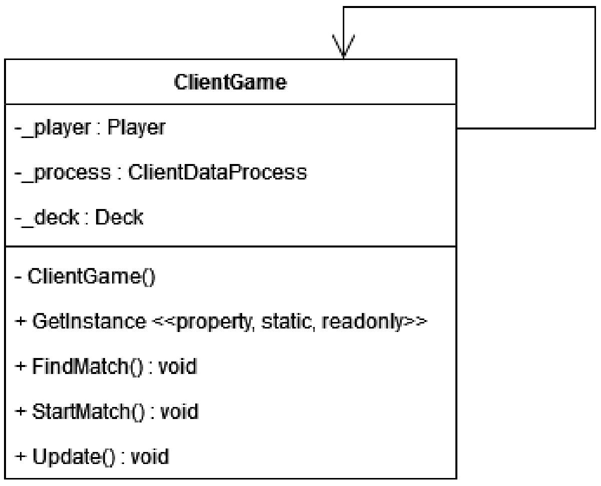


<<singleton>>
ClientNetwork
- _serverIP : IPEndPoint
- _client : Socket
# ClientNetwork()
+ GetInstance() : ClientNetwork <<static>>
+ GenerateAddress() : void <<override>>
+ Connect() : void <<override>>
+ Close() : void <<override>>
+ Send(object data) : void
+ Socket : Socket <<readonly, property>>

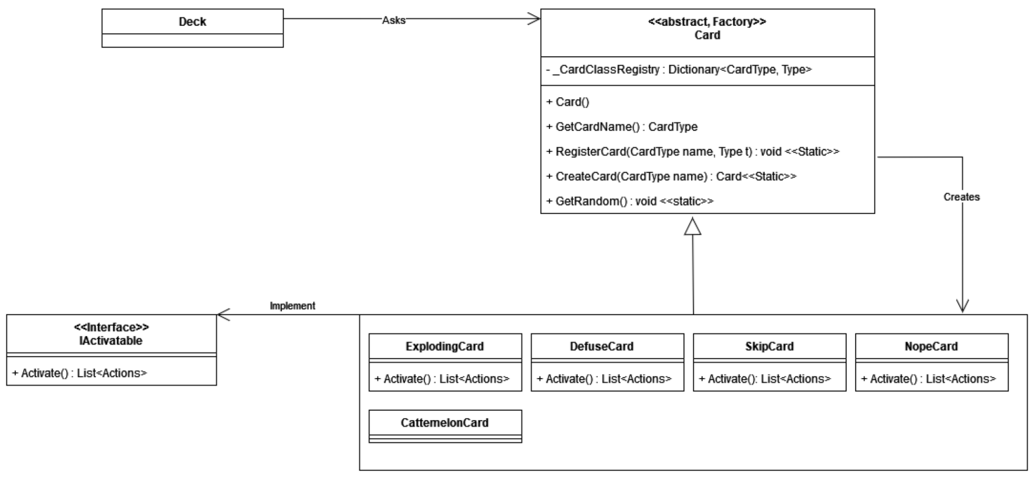


<<singleton>>
ServerNetwork
- _ClientIP : IPEndPoint
- _Server : Socket
# ServerNetwork()
+ GetInstance() : ClientNetwork <<static>>
+ GenerateAddress() : void <<override>>
+ Connect() : void <<override>>
+ Close() : void <<override>>
+ Listen() : Socket
+ SendSingle(Socket client, object data) : void
+ SendMulti(object data, PlayerGroup players) : void
+ CloseClient(Socket client, PlayerGroup players) : void

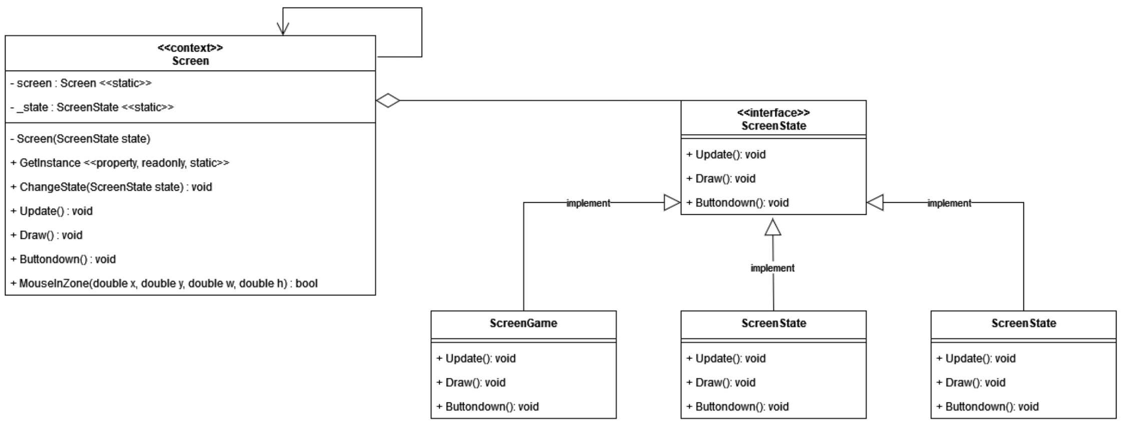
ClientGame uses the Singleton pattern because there is only one game instance when the player runs the Client project.



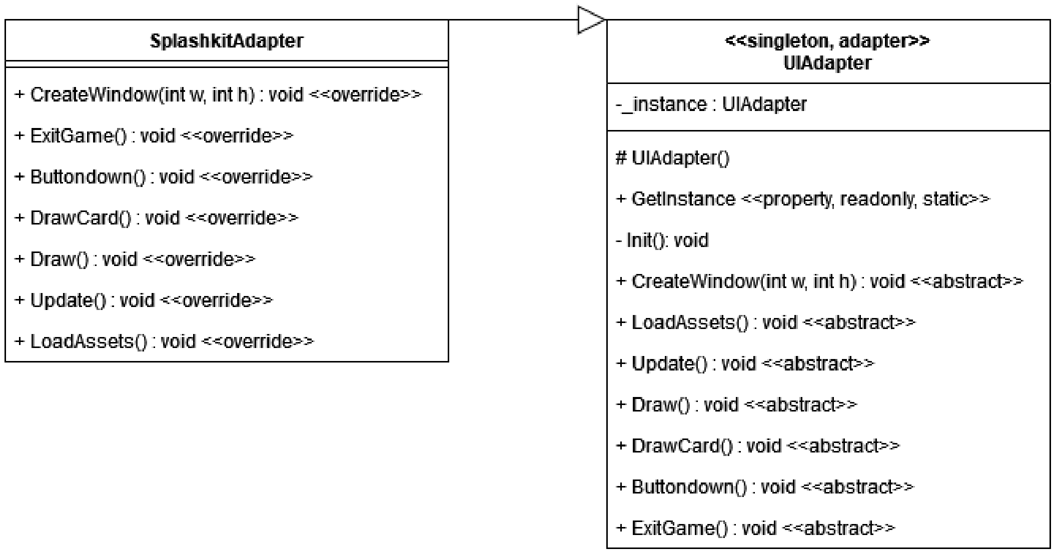
2. Factory pattern
   The Card class is the factory that is responsible for creating a different type of card. Whenever a Deck need to add any kind of card it will ask the Card class to create


Deck
Asks
<<abstract, Factory>>
Card
- _CardClassRegistry : Dictionary<CardType, Type>
+ Card()
+ GetCardName() : CardType
+ RegisterCard(CardType name, Type t) : void <<Static>>
+ CreateCard(CardType name) : Card<<Static>>
+ GetRandom() : void <<static>>
Creates
Implement
<<Interface>>
IActivatable
+ Activate() : List<Actions>
ExplodingCard
+ Activate() : List<Actions>
DefuseCard
+ Activate() : List<Actions>
SkipCard
+ Activate(): List<Actions>
NopeCard
+ Activate() : List<Actions>
CattemelonCard

3. State Pattern
   The Screen will have a different state and when it needs to change its state the ChageState Method will be called.



4. Adapter pattern
   The Client project uses a GUI adapter to access to Splashkit library


SplashkitAdapter
+ CreateWindow(int w, int h) : void <<override>>
+ ExitGame() : void <<override>>
+ Buttondown() : void <<override>>
+ DrawCard() : void <<override>>
+ Draw() : void <<override>>
+ Update() : void <<override>>
+ LoadAssets() : void <<override>>
<<singleton, adapter>>
UIAdapter
-_instance : UIAdapter
# UIAdapter()
+ GetInstance <<property, readonly, static>>
- Init(): void
+ CreateWindow(int w, int h) : void <<abstract>>
+ LoadAssets() : void <<abstract>>
+ Update() : void <<abstract>>
+ Draw() : void <<abstract>>
+ DrawCard() : void <<abstract>>
+ Buttondown() : void <<abstract>>
+ ExitGame() : void <<abstract>>