Question 1:

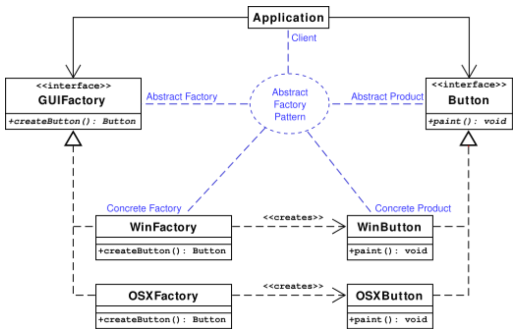
1. **Creational Patterns**
   These patterns are about class creation

   1) Abstract Factory Pattern:

      Definition:
      Abstract factory pattern is a design pattern that provides a way to merge or encapsulate a group of individual units (here factories) that have a common theme without specifying their concrete classes.

      Description:

      Class creation: >> client asks factory object to create
      >> factory will create with preserving abstraction



Here we consider a client and factories (/unites) that having common theme. We can encapsulate these factories and provide abstraction with respect to their concrete classes. So, In Abstract factory pattern, level of abstraction is provided by not directly specifying concrete classes of factory to the client. The factory object provides creation service to entire platform family. Client will not create platform family directly, factory object will be responsible for this creation and client can ask factory object to create platform.

Example: Suppose an application is made to be device independent, we might have to abstract system information to avoid multiple portability issues. In this case if we use factory pattern then we would have encapsulated OS, system driver etc and we can create product effectively through abstracted factory methods.

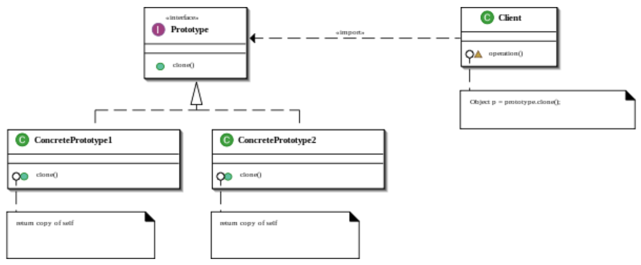
2) Prototype Pattern:

Definition: The prototype pattern is a design pattern which is used when the type of objects to create is determined by a prototypical instance. A prototypical instance which is cloned to produce new objects.

Description:

Class creation: >> object clone/prototype will be stored beforehand
>> client invokes new operator and calls clone method of class to instantiate.

Prototype pattern follows design where an object will have its clone ready and we can easily make new objects with prototype with some basic intelligence. The client that invokes the 'new' operator on a class name, calls the clone method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the clone() method through some mechanism provided by another design pattern.



Here The Factory knows how to find the correct Prototype, and each Product knows how to create new instances of itself.
The pattern implements a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. E.g. an object is to be created after an extensive system operation. We can prototype object and return its new clone.

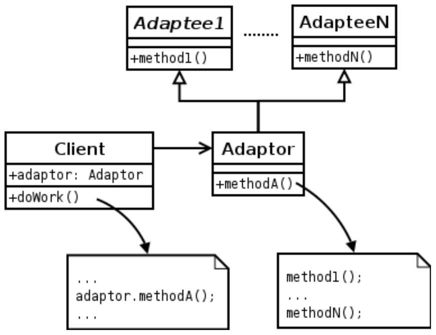
Prototype pattern has advantage over abstract factory pattern as Prototype avoids subclasses of an object creator in client application unlike abstract factory pattern.

2. Structural Patterns

These patterns are for class and object binding

1) Adapter Pattern:
Definition:
Adapter pattern is the pattern which converts the interface of a class into another interface client where adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

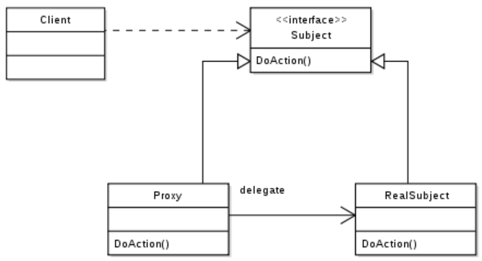
Description:
Structure: >> Wrap existing class with a new interface
>> Adapt it to match an old component to a new system



The Adapter pattern allows incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Adapter changes the interface of an existing object. And adapts it to compatible version. Adapter can also be thought as wrapper.
This pattern involves a single class which is responsible to join functionalities of incompatible interfaces. A real life example can be system driver for wireless mouse. Wireless mouse is compatible with laptop only with system driver. It acts as adapter and make laptop and mouse compatible with one another.

2) Proxy Pattern:
Definition:
Proxy pattern is design pattern which provides a surrogate or placeholder for another object to control access to it.

Description:
A proxy is designed so as to instantiate real objects when a client makes request of proxy. It remembers the identity of real object and forwards instigating request to this real object. All later requests are forwarded directly to encapsulated real object.



A proxy is simply a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.
Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy some extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

3. Behavioral Patterns

These patterns are for Class's objects communication.

1) Chain of responsibility Pattern:

Definition: It is behavioral pattern where more than one objects (chain of objects) are given responsibility to handle the request. Such chain of objects receives request and passes it among objects until an object handles it.

Description:
Behavior: >> chain/ pipeline of objects is present
>> Request is parsed through objects in chain till request is processed.

There is chain/ pipeline of objects already present. Request is given to this chain instead of a single object. Request is passed through this chain/ pipeline until any of the objects of chain executes that request. This is useful for uninterrupted execution of request.
The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked. Not recommended to use when each of the requests is only handled by one handler or when client object knows exactly which object should handle the request.

Example: In real life XML interpreter work with this type of design pattern.

2) Visitor Pattern:

Definition: Visitor pattern is design pattern which represent an abstract operation to be performed on the elements of an object structure and Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Description:
Structure: >> operations are performed without knowledge of traversal
>> lightweight classes are designed

Visitor pattern allows operations to be performed on data structures without knowledge of transversal.
It provides an abstract functionality that can be implemented to an aggregate hierarchy of objects. In this pattern we design lightweight element classes as processing functionality has been removed from list of responsibilities of such objects. New functionalities can be easily added and integrated with original inheritance hierarchy by creating new Visitor subclass.

Example: Consider the scenario where you are using a parser to extract methods calls in a file. You get an Abstract Syntax Tree, but don't want to manually walk entire tree and reason about structure. A Visitor would allow you to get a callback whenever a Method Call node is visited, without knowing about anything else!

As the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; one can create a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

Question 2:

Design patterns have been criticized for being too specific and artificially limited by OO language constraints:

Design patterns are mainly divided into three (sometimes >3) parts namely:

1. **Creational** Patterns: Used for instantiation of objects.
2. **Structural** Patterns: Used for class and object binding
3. **Behavioral** Patterns: Used for class-objects communication
4. Concurrency Patterns: Used for managing threading
5. Data Access Patterns: Used for fetching data
6. Enterprise Patterns: Used for J2EE/ .NET framework
7. Real-time Patterns: Used for real time and scalable systems

These patterns have been evolved in recent years and consequently as OO languages got more familiar, patterns are criticized to be OO specific. Procedural language lack the features of polymorphism, inheritance, data binding and concept of class. So these listed patterns hardly match with procedural kind of programming.

For software development in recent days, OO languages are preferred. E.g. Java, C#, RonR etc. These softwares can be constructed using above patterns easily.

Suppose a **problem statement**:
Build a software for **online exam**. There will be 2 types of users: Professor and students. Professor would set a MCQ questionnaire and student will answer them.

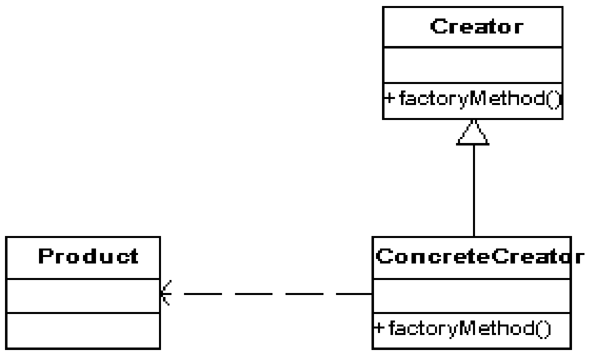
The 3 patterns I select for Software Development are:

1. Factory Method:

This type of pattern falls under creational pattern i.e. it is used for creation of objects.
Factory method defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses
In normal scenario, sometimes it is impossible at compile-time to determine which objects need instantiated. The framework knows when an object is in need, but not what sort of hence we will use Factory method design pattern. Here we have a family of objects that are all inter-related, but each entity in the family performs a different task and we cannot determine at compile-time which object to instantiate, and the object that's to be created might change at run-time.

Common terms in factory method are:

Product: Defines the interface of objects the factory method creates
Concrete Product: Implements the product interface, defines a concrete type
Creator: Declares the factory method, which returns an object of type Product
Concrete Creator-Provides a concrete factory method that returns an instance of a Concrete Product.



e.g. Suppose we are designing an application for job portal that has functionality to upload resume and these resume may have different file formats like .pdf, .jpeg, .docx etc.
At some level of granularity, we ought to create different objects to handle these different types of documents, it could be at a high-level, defining a different Doc class for each type of file, or at a low-level, defining a different DocumentReader class for each file
We might not know at compile-time what sorts of files are going to be open, and the types of files supported by the system may add.

In our scenario, for Online exam, we will use factory method as follows:
Suppose for class Student we will have interfaces as:
Product: Student Profile
Creator: New Student Creation
Concrete Creator: New Student Creator

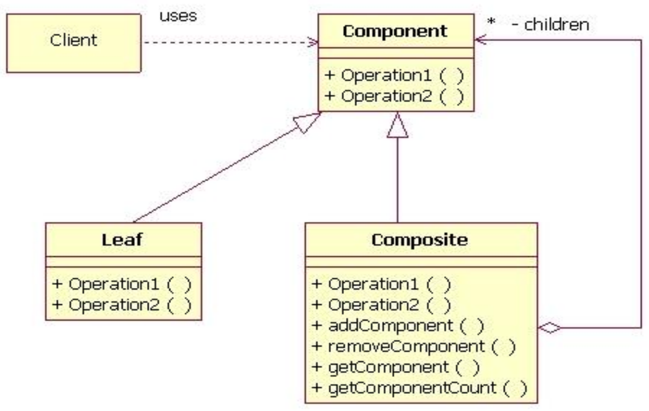
2. Composite:

It composes objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
In normal scenario, if we want to build more complex components out of simpler components, it is difficult to combine them together at run-

time to create complex objects and we don't want to treat the objects and their containers differently, their manipulation should be uniform across all related objects.
We can use composite when we want clients to be able to ignore the difference between compositions of objects and individual objects. Clients would treat all fellow objects in the composite structure uniformly

Common terms in composite are:

Component: It declares the interface for objects in the composition and it implements default behavior for the interface common to all classes, as appropriate
Leaf: it represents leaf objects in the composition and defines primitives in the system
Composite: it defines behavior for components having children and stores children components



Advantages: Defines class hierarchies consisting of primitives and compositions uniformly; wherever a client expects a primitive object, it can also take a composition. It makes the client simple, as the client can treat composite structures and single objects uniformly. And also it makes it easy to add new kinds of components.

In our software, aspects with this pattern will be :
Component: Students Group / Course Students
Leaf: Students
Composite: add student to group/ delete from group / add a group / delete a group.

3. Iterator:

This design pattern falls under behavioral pattern. It is used to sequentially access the elements of a collection.
It provides a means to access the elements of an aggregate type without exposing its underlying implementation.
It uses an iterator to access an aggregate object's contents without exposing its underlying implementation and it supports multiple traversals of aggregate objects. It also decouples an aggregate object from the algorithms that act upon that object.

Common terms in composite are:
Iterator: it defines an interface for accessing and traversing elements
ConcreteIterator: Particular iterator implementation
Aggregate: it defines an interface for creating an iterator object
ConcreteAggregate: creates concrete iterators.

In our Software these terms will be:
Iterator: Exam Start event (It will call objects students till some students take the exam)
Concrete Iterator: Particular exam i.e. time constraint added likewise.
Concrete Aggregate: an event to create various exams.

In addition to these there are various patterns that were designed for languages including procedural languages e.g.

a. Handles

The best way to perform encapsulation is procedural programming. Handle is a pointer or integer that is passed to the functions and the information about data is completely hidden.

b. Contexts

Context is a struct that contains the state in terms of data stored of some system, just like the members of an instance or object. In procedural programming we usually specify function (address of the Context, argument). The internal procedures use the context structure directly, while public functions take a handle only and the implementation resolves it to the actual context structure. This ensures encapsulation and abstractness.

c. Callbacks

Sometimes called as function pointer. The implementer of the procedure passes the reference/ address of procedure to add custom behavior to a system. This allows polymorphism in procedural pattern.
So these are some procedural design pattern that may help in building software.

Question 3:

Github API:

Application program interface (API) is a set of routines, protocols, and tools for building software applications. An API specifies how software components should interact and APIs are used when programming graphical user interface (GUI) components.

Github had announced its API one month after the site had launched. In earlier days, Github API was RESTful API and hypermedia driven but recently Github made big change and now Github API is available through GarphQL. GraphQL is, at its core, a specification for a data querying language.
Added advantage by GraphQl is mainly Scalability.

Earlier Github API had REST for over 60% requests to database tier. Hypermedia navigation requires client to frequently and repeatedly communicate with the server so that it can fetch the information as and when it is needed. Responses in this communication were bloated and filled with all kinds of *_url hints in JSON responses to help customers continue to navigate through the API to get what they needed. Still customers reported that REST API was not that flexible as it sometimes required two to three calls to assemble a complete view of source. So it was inferred that REST API sent too much data and did not include the data customers actually needed.

Github wanted assurances of type-safety for user mentioned parameters and also to generate documentation from our code. This is when Github shifted to GraphQL mainly for scalability reason.

GraphQL vs REST:

GraphQL is data querying language developed by Facebook. In GraphQL one can construct her own request by defining resources she wants. One can send this via POST to server and response from server matches with her request through POST.

Through GraphQL one can fetch a significant amount of data through one call and get exact the things that were requested through request. This type of design enables clients where smaller payload sizes are essential. For example, a mobile app could simplify its requests by only asking for the data it needs. This enables new possibilities and workflows that are freed from the limitations of downloading and parsing massive JSON blobs.

GraphQL is more of a specification than an architectural style like REST. In REST, server returns whole JSON data for particular request without any processing on it where as in GraphQL one can explicitly specify exacts of data to be fetched through Query language.

This make GraphQL more inclined towards optimization and also the conciseness towards response from server.
e.g.

So here we can see that we are specifying exact what all to fetch from server side in a request and client responses with exact things asked in request.

Request

```
{
  viewer {
    login
    starredRepositories {
      totalCount
    }
    repositories(first: 3) {
      edges {
        node {
          name
          stargazers {
            totalCount
          }
          issues(states:[OPEN]) {
            totalCount
          }
        }
      }
    }
  }
}
```

Response

```
{
  "data":{
    "viewer":{
      "login": "octocat",
      "starredRepositories": {
        "totalCount": 131
      },
      "repositories":{
        "edges":[
          {
            "node":{
              "name":"octokit.rb",
              "stargazers":{
                "totalCount": 17
              },
              "forks":{
                "totalCount": 3
              },
            }
          },
          {
            "node":{
              "name":"octokit.objc",
              "stargazers":{
                "totalCount": 2
              },
              "issues": {
                "totalCount": 10
              }
            }
          }
        ]
      }
    }
  }
}
```

Question 4:

Pokemon Go !!

Pokemon Go is Augmented reality game designed basically for mobile users by Niantic. It is compatible with android, apple devices. It is based on popular fiction tele-series Pokemon.

Pokemon Go design basically revolves around the player and his collected pokemon. It mainly uses location service to track the location of player and provide services of pokestops/pokemon sightings accordingly.

Primary tasks for player would be:

1. Player will login with respective credentials to play the game
2. Search for pokemons by walking to nearby places where one can see hints of pokemon.
3. There will be several pokestops where player can get pokeballs, Potions and eggs. A PokéStop will change its shape when you walk close enough. Touch it to interact with it, and spin the Photo Disc to get items.
4. Once pokemon is sighted, catch it by throwing pokeball to the pokemon. One can switch on camera/ give permission to game to access camera to augment reality of pokemon being nearby.
5. Player can make a team with peers for battle.
6. Once sufficient pokemons are captured, Player can battle at Gyms around the map. At rival Gyms, Player can battle other teams' Pokémon for a chance at claiming the Gym.
   Each rival Pokémon Player defeat reduces the Gym's Prestige and potentially lowers the Gym's level. Reduce the Gym's Prestige to zero to capture the Gym for Player's team.
7. Player can continue acquiring pokemons and winning battles at Gym thereby capturing the Gym.

Common Terms in Game:

-Combat Power: A Pokémon's attack strength is measured in units of Combat Power. A Pokémon's CP determines how well it will perform in battle.

-Defender Bonus: You can receive a daily reward of Stardust and PokéCoins for defending a Gym. The Defender bonus can be claimed at the shop

-Eggs: Pokémon Eggs are items that can be found at PokéStops. Once you place an Egg in an incubator and walk a specific distance, the Egg will hatch into a Pokémon.

-Evolution: Evolution is the process of using Candy to change a Pokémon into an individual of an evolved species of Pokémon.

-Experience Points (XP): Your advancement is measured in Experience.

-Points (XP). Increase your XP to advance to higher Trainer levels.

-Gyms: Gyms are locations where you can battle the Pokémon of rival teams, or train your Pokémon by battling against the Pokémon assigned there by other members of your team.

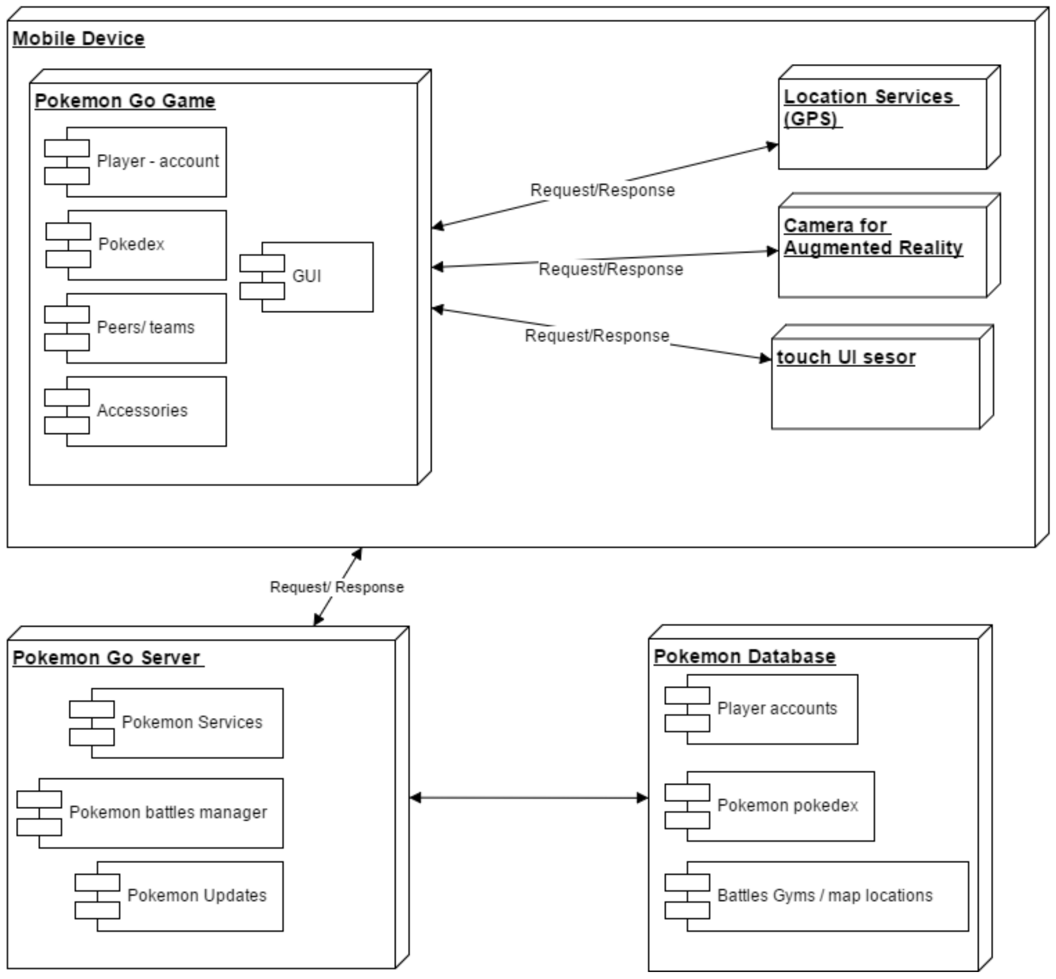
-Hit Points (HP): A Pokémon's health is measured in Hit Points (HP).

-Pokédex: Your Pokédex is where you'll find info about all the Pokémon species you've caught or encountered.

-PokéStops: PokéStops are locations where you can gather items such as Poké Balls, Potions, and Eggs
-Potions,Stardust,Razzberry etc: Other accessories for pokemon healths.

Architecture:

Component architecture:



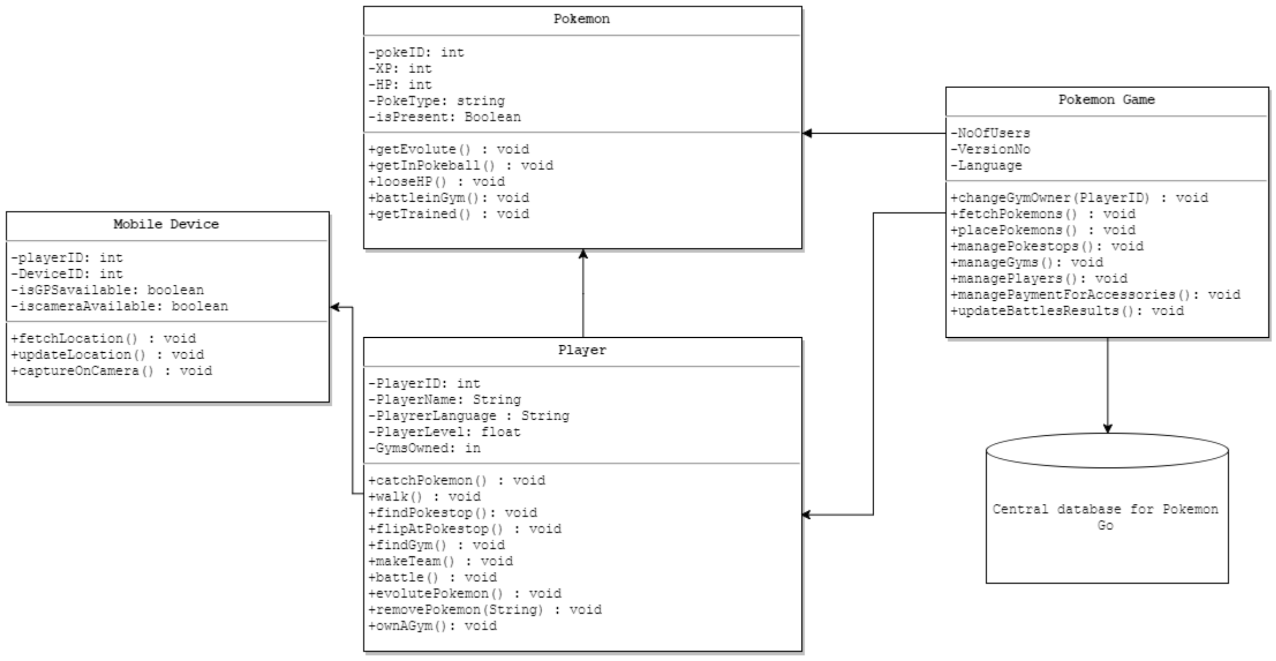
Basic components considered are Player, Pokemon Go Server and Database. Pokemon Go server will be responsible for Gaming services including providing GUI, events such as battles at gym, notifications etc. Pokemon go server monitors battles at gym and accordingly update ownership of gyms.
Player will be having pokemon Go game installed on her mobile device. Pokemon Go mainly uses location services from GPS so that to provide augmented reality to players. Pokemon Go would also use camera to provide augmentation when a pokemon is spotted. In pokemon Go, one must have an account and this account is

accessible from any device if exact authentication details are provided. There will be separate pokedex i.e. database of captured pokemons is maintained for every player.

Accessories are also maintained along with captured pokemons thi may include pokeballs, potions etc.
A central database must be maintained for locations of pokestops, gyms, predicted pokemon spots and also the directory of users playing game, the characteristics of pokemon in terms of XP, their count etc.

For architectural design, one might select Call and Return Architecture with Object Oriented Architecture. As the pokemon Go game has multiple functionalities with respect to specific object i.e. User has pokemons, User has location which is tracked, Pokemon have their XPs etc. So Object oriented architecture can suffice these needs of functionalities.

More detailed architecture:
Class Diagram:



Explanation:
Four basic classes are considered namely

1. Player: the person who logs in to game and play
   Player class might store data such as player ID,playerName, GymsOwned etc and can have procedures like catchPokemon, walk, findpokemon, makeTeam, revolutePokemon, ownGym, battle and likewise.
2. Mobile Device: The device used for playing game. Device must be location service enable so as to implement augmented reality. Mibile Device class will

store information such as playerLoggedIn, isGPSon like data along with procedures namely fetchLocation, updatelocation while Player walks, turn on camera when player spots a pokemon and likewise

3. Pokemon: Pokemon class will have dtata types namely pokemon name, its XP, HP and functions such as revolute, battle in Gym, get in pokeball and likewise.
4. Pokemon Game: This will be class having data types and services to provide GUI, gaming experience, battle services and networking among peers for team battles.
5. Central Database: This repository may store all the data of pokemon, user registry, pokestops location, location of gyms and likewise.

So this is overall architecture with OO pattern being considered for Pokemon Go!

References used:

1. *Lecture slides for CSC 510 by Dr. Pranin.*
2. *Wikipedia*
3. *https://sourcemaking.com/design_patterns*
4. *Pokemon Go homepage*