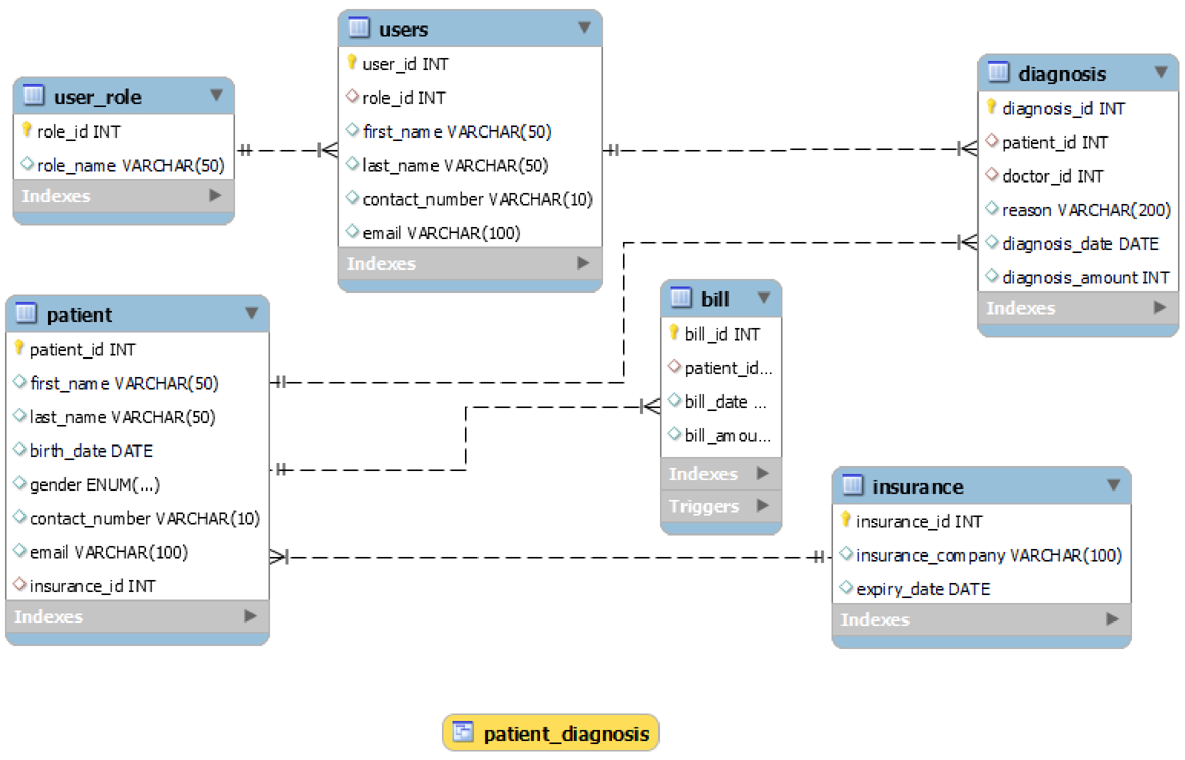
Hospital management system

Q1. Hospital management system database design and queries.

Schema diagram:



Queries:

```
#database creation
create database if not exists hospital_management;

#using databse
use hospital_management;

#creating user_roles and users
create table user_role(
 role_id int primary key,
 role_name varchar(50) unique);

create table users(user_id int primary key ,
role_id int,
foreign key(role_id) references user_role(role_id),
first_name varchar(50),
last_name varchar(50),
contact_number varchar(10)
check(length(contact_number) = 10),
email varchar(100));

# creating patient table
create table patient(
patient_id int primary key ,
first_name varchar(50),
last_name varchar(50),
birth_date date,
gender enum('Male', 'Female', 'Other'),
contact_number varchar(10)
check(length(contact_number) = 10),
email varchar(100),insurance_id int,
foreign key(insurance_id) references
insurance(insurance_id));

 #creating diagnosis table
 create table diagnosis(
 diagnosis_id int primary key ,
 patient_id int,
 doctor_id int,
 foreign key(patient_id) references patient(patient_id),
 foreign key(doctor_id) references users(user_id),
 reason varchar(200),
 diagnosis_date date,
 diagnosis_amount int);

 #creating insurance table
```

```
create table insurance(
insurance_id int primary key,
insurance_company varchar(100),
expiry_date date);

#creating bill table
create table bill (
bill_id int primary key  auto_increment,
patient_id int,foreign key(patient_id) references patient(patient_id),
bill_date date,
bill_amount int);

# Q1. Write necessary queries to register new user roles and personas
insert into user_role values (1,'Doctor');
insert into users values (10,1, 'Aniket', 'shinde', '8594933221', 'aniketshinde@gmail.com');
insert into patient values (100, 'Aditya', 'patil',  '2024-1-13', 'Male', '9484736493','adityapatil@gmail.com', 1000);

# Q2. Write necessary queries to add to the list of diagnosis of the patient tagged by date.
```

insert into diagnosis values(10000,100,10,'kidney stone', '2024-1-15',50000);

# Q3. Write necessary queries to fetch required details of a particular patient.

select p.*,d.doctor_id,d.reason,d.diagnosis_date, i.insurance_company, i.expiry_date from patient p join diagnosis d on p.patient_id =

d.patient_id join insurance i on i.insurance_id = p.insurance_id where p.patient_id = 100;

# Q4. Write necessary queries to prepare bill for the patient at the end of checkout.

insert into bill(patient_id,bill_date,bill_amount) values (100,'2024-1-20', 50000);

#Q5. Write necessary queries to fetch and show data from various related tables (Joins)

select distinct concat(p.first_name," ",p.last_name) as Patient_name, concat(u.first_name," ",u.last_name) as Doctors_name, d.reason,d.diagnosis_date, i.insurance_company, i.expiry_date,b.bill_date,b.bill_amount from patient p join

diagnosis d on p.patient_id = d.patient_id join insurance i on i.insurance_id = p.insurance_id join

bill b on b.patient_id = p.patient_id join users u on u.user_id = d.doctor_id where p.patient_id = 100;

# Q6. Optimize repeated read operations using views/materialized views.

create view patient_diagnosis as select concat(p.first_name," ",p.last_name) as 'Patients name', d.*, concat(u.first_name," ",u.last_name) as 'Doctors name' from diagnosis

d join patient p on p.patient_id = d.patient_id join users u on u.user_id = d.doctor_id where p.patient_id = 100;

select * from patient_diagnosis;

#Q7. Optimize read operations using indexing wherever required. (Create index on at least 1 table)

create index idx_patient_id on diagnosis(patient_id);

#Q8. Try optimizing bill generation using stored procedures.

delimiter //

create procedure bill_generation(in patient_id int)

begin

declare total_bill int;

select d.diagnosis_amount into total_bill from diagnosis d where d.patient_id = patient_id;

insert into bill(patient_id,bill_date,bill_amount) values (patient_id,current_date(),total_bill);

```
 End //
 delimiter ;
```

#Q9. Add necessary triggers to indicate when patients medical insurance limit has expired.

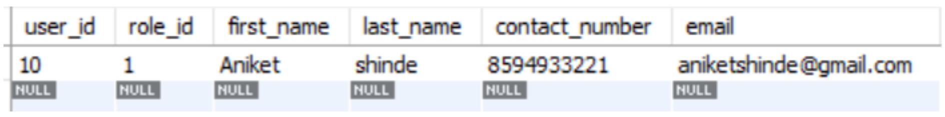
```
delimiter //
create trigger check_insurance_expiry before insert on bill
for each row
begin
        declare exp_date date;
    select expiry_date into exp_date from insurance where
insurance.insurance_id = (select insurance_id from patient
where patient_id =
    NEW.patient_id);
    if current_date() > exp_date then
     SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Billing cannot be done for a
patient with expired insurance.';
    end if;
    end; //
delimiter ;
```

```
#using trigger
 insert into insurance values (1001, 'LIC', '2026-10-21');
 update insurance set expiry_date = '2022-5-3' where
insurance_id = 1001;
 insert into patient values (101, 'Amit', 'pawar',  '2024-8-
13', 'Male', '9484768893','amitpawar@gmail.com', 1001);
 insert into diagnosis values(10001,101,10,'Fracture',
'2024-3-18',30000);
 call bill_generation(101);
```

Outputs and results:

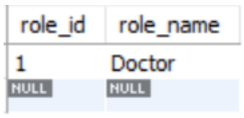
- All the tables are in 1NF as all have an atomic data, 2NF as all non-prime attributes fully dependent on the prime attribute and 3NF as there is no transitive dependency. While normalization up to 3NF is a standard practice, it's essential to strike a balance. Over-normalization can lead to complex query structures so that tables are normalized up to 3NF.
- **Tables:**

1. User table



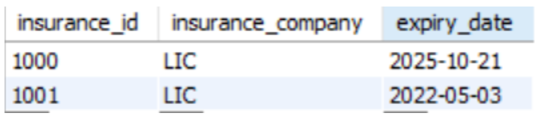
user_id	role_id	first_name	last_name	contact_number	email
10	1	Aniket	shinde	8594933221	aniketshinde@gmail.com
NULL	NULL	NULL	NULL	NULL	NULL

2. User role table



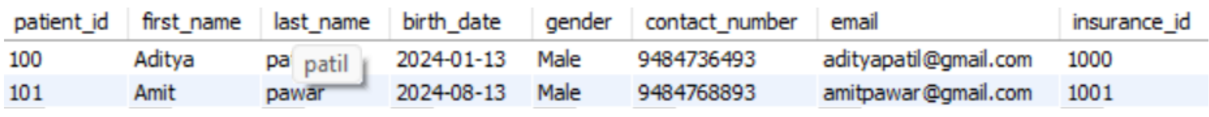
role_id	role_name
1	Doctor
NULL	NULL

3. Insurance table



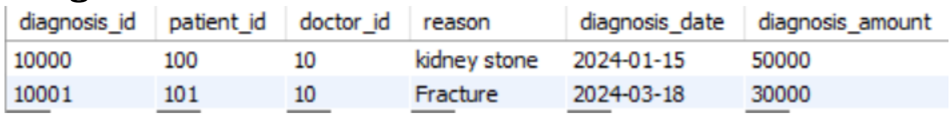
insurance_id	insurance_company	expiry_date
1000	LIC	2025-10-21
1001	LIC	2022-05-03

4. Patient table



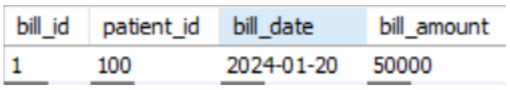
patient_id	first_name	last_name	birth_date	gender	contact_number	email	insurance_id
100	Aditya	patil	2024-01-13	Male	9484736493	adityapatil@gmail.com	1000
101	Amit	pawar	2024-08-13	Male	9484768893	amitpawar@gmail.com	1001

5. Diagnosis table



diagnosis_id	patient_id	doctor_id	reason	diagnosis_date	diagnosis_amount
10000	100	10	kidney stone	2024-01-15	50000
10001	101	10	Fracture	2024-03-18	30000

6. **Bill table**



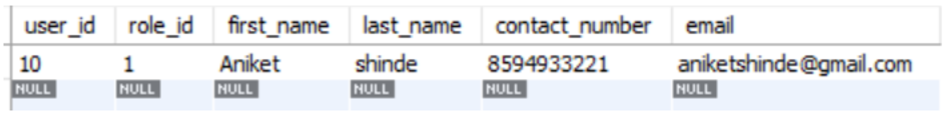
bill_id	patient_id	bill_date	bill_amount
1	100	2024-01-20	50000

- **Questions**

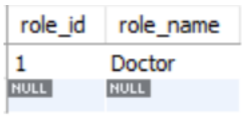
1. Write necessary queries to register new user roles and personas. Here user roles can be different like doctor, nurse, cashier even different kinds of doctors like dentist, surgeon etc.

   Query:
   insert into user_role values (1,'Doctor');
    insert into users values (10,1, 'Aniket', 'shinde', '8594933221', 'aniketshinde@gmail.com');

   Result:



user_id	role_id	first_name	last_name	contact_number	email
10	1	Aniket	shinde	8594933221	aniketshinde@gmail.com
NULL	NULL	NULL	NULL	NULL	NULL

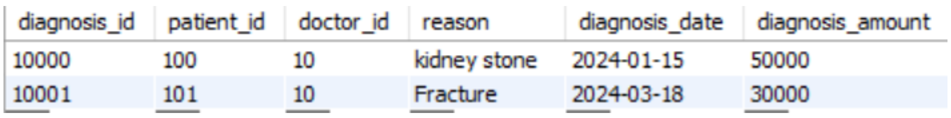


role_id	role_name
1	Doctor
NULL	NULL

2. Write necessary queries to add to the list of diagnosis of the patient tagged by date.

   Query:
   insert into diagnosis values(10000,100,10,'kidney stone', '2024-1-15',50000);
   insert into diagnosis values(10001,101,10,'Fracture', '2024-3-18',30000);

   Result:



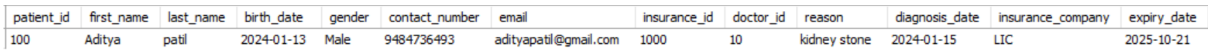
diagnosis_id	patient_id	doctor_id	reason	diagnosis_date	diagnosis_amount
10000	100	10	kidney stone	2024-01-15	50000
10001	101	10	Fracture	2024-03-18	30000

3. Write necessary queries to fetch required details of a particular patient. Here patient data from patient, diagnosis and insurance tables is shown together.

   Query:

select p.*,d.doctor_id,d.reason,d.diagnosis_date,
i.insurance_company, i.expiry_date from patient p join diagnosis d on p.patient_id =
d.patient_id join insurance i on i.insurance_id = p.insurance_id where p.patient_id = 100;

Result:

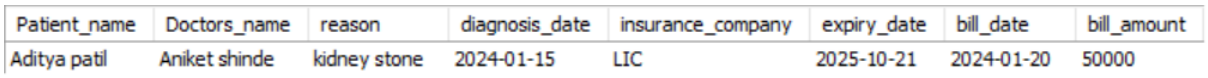


patient_id	first_name	last_name	birth_date	gender	contact_number	email	insurance_id	doctor_id	reason	diagnosis_date	insurance_company	expiry_date
100	Aditya	patil	2024-01-13	Male	9484736493	adityapatil@gmail.com	1000	10	kidney stone	2024-01-15	LIC	2025-10-21

4. Write necessary queries to prepare bill for the patient at the end of checkout. We can fill the bill data for same patient id, same patient can come to the hospital multiple times.
   Query: insert into bill(patient_id,bill_date,bill_amount) values (100,'2024-1-20', 50000);

5. Write necessary queries to fetch and show data from various related tables (Joins). It gives data from patients name, doctors name, insurance and bill details;
   Query:
   select concat(p.first_name,p.last_name) as Patient_name,d.doctor_id,d.reason,d.diagnosis_date, i.insurance_company, i.expiry_date,b.bill_date,b.bill_amount from patient p join diagnosis d on p.patient_id = d.patient_id join insurance i on i.insurance_id = p.insurance_id join bill b on b.patient_id = p.patient_id where p.patient_id = 100;

   Result:



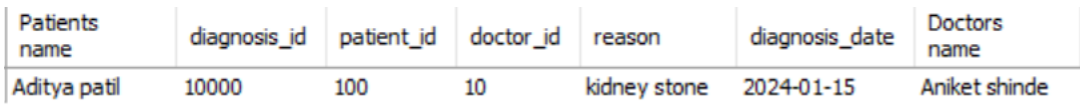
Patient_name	Doctors_name	reason	diagnosis_date	insurance_company	expiry_date	bill_date	bill_amount
Aditya patil	Aniket shinde	kidney stone	2024-01-15	LIC	2025-10-21	2024-01-20	50000

6. Optimize repeated read operations using views/materialized views. This view provides patients and diagnosis details.
   Query:
   create view patient_diagnosis as select concat(p.first_name,'' '',p.last_name) as 'Patients name', d.*, concat(u.first_name,'' '',u.last_name) as 'Doctors name' from diagnosis d join patient p on p.patient_id = d.patient_id join users u on u.user_id = d.doctor_id where p.patient_id = 100;

#To view data in view

```
select * from patient_diagnosis;
```

Result:



Patients name	diagnosis_id	patient_id	doctor_id	reason	diagnosis_date	Doctors name
Aditya patil	10000	100	10	kidney stone	2024-01-15	Aniket shinde

7. Optimize read operations using indexing wherever required. (Create index on at least 1 table)
   Query: create index idx_patient_id on diagnosis(patient_id);

8. Try optimizing bill generation using stored procedures.
   Query:

```
delimiter //

   create procedure bill_generation(in patient_id int)

begin
 declare total_bill int;
 select d.diagnosis_amount into total_bill from diagnosis d where
d.patient_id = patient_id;
 insert into bill(patient_id,bill_date,bill_amount) values
(patient_id,current_date(),total_bill);
 End //
 delimiter ;
```

#To call procedure

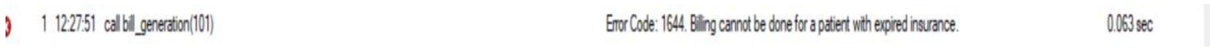
```
Call bill_generation(100);
```

9. Add necessary triggers to indicate when patients medical insurance limit has expired.
   Query:

```
delimiter //
create trigger check_insurance_expiry before insert on bill for each
row
begin
        declare exp_date date;
   select expiry_date into exp_date from insurance where
insurance.insurance_id = (select insurance_id from patient where
patient_id =
   NEW.patient_id);
   if NEW.bill_date > exp_date then
```

```
     SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Patients insurance is expired bill not
possible!';
    end if;
    end; //
delimiter ;
```

Result: If insurance is not expired, new record is inserted into bill table else following message is shown when we call procedure or insert into bill table:



```
1  12:27:51  call bill_generation(101)          Error Code: 1644. Billing cannot be done for a patient with expired insurance.     0.063 sec
```

Q2. Rendering and Design Patterns Report

Pattern:

Pattern is a toolkit of solutions to common problems in software design. They define a common language that helps your team to communicate more efficiently.

Design Pattern:

Design pattern is a description or a template of how to solve a problem that can be used in very different situations. They generally used to solve the problems of object generation and integration. Following are the types of design pattern -

- ## Creational Pattern:

  This type deals with object creation and initialization. This pattern gives the program more flexibility in deciding which objects need to be created for a given case.
  Example: Singleton, Factory, Abstract Factory etc.

  1. ### Factory Method:
     Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
     Problem: Problem here is when you have one class and later you decided to add another class into your software then you must change entire codebase.
     Solution: The Factory Method pattern suggests that you replace direct object construction calls with calls to a special *factory* method.
     There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. We can implement interfaces to a new subclass whenever we need to add a class.
     Applicability: Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.
     Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.

2. Abstract Factory Method:

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problem: Suppose you have different variety of products and variety of families like chair and sofa of modern and art deco type. You need to create individual objects that they match other objects of the same family. Also, you don't want to change existing code when adding new products or families of products to the program.

Solution: Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family. We must create abstract classes of both products as well as factory. We must create concrete classes using these abstract classes.

Applicability: Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand, or you simply want to allow for future extensibility.

3. Builder:

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
Problem: Imagine we have a complex object which requires step-by-step initialization of many fields and nested objects. Such objects require large objects with lots of parameters. Let's suppose the object is house there are various combinations of house, we can add subclass with lots of combinations, but we will end up with large number of subclasses or even with constructors with large parameters constructor call becomes ugly as most of the parameters are unused in some cases.

Solution: The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*. The pattern organizes object construction into a set of steps. To create an object, you execute a series of these steps on a builder object. You can call only those steps that are necessary for producing a particular configuration of an object. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

Applicability: Use the Builder pattern to get rid of a "telescoping constructor" (Constructors with large number of parameters). Use the Builder pattern when you want your code to be able to create different representations of some product .

4. **Prototype:**

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Problem: When we want to make a copy of an object it is not easy every time as some of the fields of an objects may be private also, they have their dependencies to some other concrete classes.

Solution: The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

Applicability: Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy. Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

5. **Singleton:**   **Singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem: This pattern solves two main problems one is it ensures class has just a single instance and another is it provides a global access point to that instance.

Solution: Make the default constructor private, to prevent other objects from using the new operator with the Singleton class. Create a static creation method that acts as a constructor. This method calls the private constructor to create an object and saves it in a static field.

Applicability: Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program. Use the Singleton pattern when you need stricter control over global variables.

- ## Structural pattern:

  Structural patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

  Example: Proxy, Bridge, Flyweight etc.

1. ### Adapter:

   Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. **Problem:** Suppose your application supports XML format for data but later updating it adding some features to it if we decide to use JSON file for data, it makes it difficult to change existing code to adapt with JSON and what if we do not have access to library.
   Solution: You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it. Here how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Applicability: Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code. Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

2. **Bridge:** **Bridge** is a structural design pattern that lets you split a large **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies abstraction and implementation which can be developed independently of each other.

Problem: Suppose we have class shape and two subclasses circle and triangle. Later we decide to add new dimension as colours, suppose, blue and red. It creates four class combinations. Adding more colours increases number of combinations exponentially.

Solution: This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by colour. That's a very common issue with class inheritance. The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. Abstraction means GUI where implementation means operating systems API. We have several combinations of GUI and OS API. After selecting GUI, we can go to API where reference field pointing.

Applicability: Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality. Use the pattern when you need to extend a class in several orthogonal (independent) dimensions. Use the Bridge if you need to be able to switch implementations at runtime.

3. Composite:

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.
Problem: Suppose we have two class's products and boxes. Box contains multiple products as well as other smaller boxes. Determining price of such order is difficult.
Solution: The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.
Applicability: Use the Composite pattern when you must implement a tree-like object structure. Use the pattern when you want the client code to treat both simple and complex elements uniformly.

4. Decorator:

Decorator is a structural design pattern that lets you attach new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

Problem: suppose there is a notifier app that notifies about important events. But different users prioritize different kind of notifications, which would bloat the code immensely.

Solution: A wrapper is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives.

Applicability: Use the Decorator pattern when you need to be able to assign extra behaviours to objects at runtime without breaking the code that uses these objects. Use the pattern when it's awkward or not possible to extend an object's behaviour using inheritance.

5. Façade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Problem: Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework, making it hard to maintain.

Solution: A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Applicability: Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem. Use the Facade when you want to structure a subsystem into layers.

6. Flyweight:

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object.

Problem: Many times, application works on one machine and no other due to lack of RAM. When there are large number of objects and while loading RAM gets full which results into crashing of an app.

Solution: The constant data of an object is called as intrinsic state. The object altered from outside by other object is extrinsic state. The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object. An object that only stores the intrinsic state is called a flyweight. Extrinsic state stores in a container object. We need to create the class that stores extrinsic state along with reference to the flyweight object.

Applicability: Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.

7. **Proxy:**
   Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
   Problem: Suppose you have massive objects that uses vast amount of system resources. If we create the object when needed causes lot of code duplication.
   Solution: The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.
   Applicability: When you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it. This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large. This is when you want to keep a history of requests to the service object. This is when the service object is located on a remote server.

- **Behavioural pattern:**

  Behavioural patterns take care of effective communication and the assignment of responsibilities between objects.

  Example: Command, Iterator, State etc.

1. **Chain of responsibility:**
   Chain of Responsibility is a behavioural design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
   Problem: During authentication each time when we add new check into the system code becomes mess and may affect other checks.
   Solution: The **Chain of Responsibility** relies on transforming behaviour into stand-alone objects called *handlers*. The pattern suggests that you link these handlers into a chain. Each linked handler

has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain.
Applicability: Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand. Use the pattern when it's essential to execute several handlers in a particular order. Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.

2. **Command:**
   Command is a behavioural design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request's execution, and support undoable operations.
   Problem: Suppose you have buttons that looks same, but all do different things. For all these buttons we can add many subclasses which creates a mess of code. When there are alternatives for these button operations, we must copy the code from respective button which leads to duplicate code.
   Solution: The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all the request details, such as the object being called. All the commands must have same interface and one execute method in it.
   Applicability: Use the Command pattern when you want to parametrize objects with operations. Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely. Use the Command pattern when you want to implement reversible operations.
3. **Iterator:**
   Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation.
   Problem: Data stored in collections like list, tree, and stack. When we want to traverse the elements in collection, it is easy for simple collection like list but for complex collection like tree there are several approaches like breadth first and depth first which we required to change at any time.

Solution: The main idea of the Iterator pattern is to extract the traversal behaviour of a collection into a separate object called an *iterator*. In addition to implementing the algorithm itself, an iterator object encapsulates all the traversal details, such as the current position and how many elements are left till the end.
Applicability: Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients. Use the pattern to reduce duplication of the traversal code across your app. Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.

4. **Mediator:**

   Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
   Problem: Suppose you have different components like text field, checkbox, buttons etc. If one or two components are coupled with each other, we cannot use these components into another form.
   Solution: The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components.
   Applicability: Use the Mediator pattern when it’s hard to change some of the classes because they are tightly coupled to a bunch of other classes. Use the pattern when you can’t reuse a component in a different program because it’s too dependent on other components. Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

5. **Memento:**

   Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
   Problem: Suppose you want to undo the operations carried out on a text in a text editor. For that we must take snapshots that is we need to store state of an object. But objects are not public, there are many private fields which restricts the access. To allow other

objects to write and read data to and from a snapshot, you'd probably need to make its fields public.
Solution: The pattern suggests storing the copy of the object's state in a special object called *memento*. The contents of the memento aren't accessible to any other object except the one that produced it. Other objects must communicate with mementos using a limited interface which may allow fetching the snapshot's metadata, but not the original object's state contained in the snapshot.
Applicability: Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object. Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

6. **Observer:**
   Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
   Problem: Suppose some customers are interested in a new product yet to come. We can notify all the customers whenever that product becomes available via email. But the problem here is all the customers not interested in new product get upset.
   Solution: Object which is been observed known as publisher and observing objects called as subscriber. The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.
   Applicability: Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically. Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.
7. **State:**
   State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
   Problem: Suppose our objects have different states depending on internal fields. It can change its state whenever respective condition met. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code like this is very difficult to maintain because any change to the

transition logic may require changing state conditionals in every method.
Solution: The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes. We can replace state objects when respective condition met.
Applicability: Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently. Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields. Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

8. **Strategy:**

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
Problem: Suppose we created a navigation app which is used by one car users. Later we decided to other transport mediums, but every time we add new medium result in increasing size of our class which makes it difficult to maintain the class.
Solution: The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*. Like google map first we select destination after that we can select transport options like car, bike etc.
Applicability: Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime. Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior. Use the pattern when your class has a massive conditional statement that switches between different variants of the same algorithm.

9. **Template Method:**

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
Problem: Suppose our data mining app only supports doc files then in update whenever we add support to other files with new

classes, only the code dealing with data extraction changes, the basic algorithm remains same for all. Also, on client side it involves lot of conditionals to choose respective class.
Solution: The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method.* To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed.
Applicability: Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure. Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

10. **Visitor:**

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.
Problem: Suppose you have colossal graph of different locations in city in your map.
If you decided to export all these nodes into XML you need to add export method to each node, if it results in bug that breaks entire code. And, to support other format we must change that class again and again.
Solution: The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes. The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.
Applicability: Use the Visitor when you need to perform an operation on all elements of a complex object structure. Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

Rendering Patterns:

1. **Client-side rendering:**

In Client-Side Rendering (CSR) only the barebones HTML container for a page is rendered by the server. The logic, data fetching,

templating, and routing required to display content on the page is handled by JavaScript code that executes in the browser/client. As complexity of page increases size of JavaScript bundle also increases.
Pros: All the UI generated on client and entire web application loaded on first request which allows to have a Single-Page Application that supports navigation without page refresh and provides a great user experience.
Cons: If large payload and network requests are there which may result in rendering delay. At the first-time user need to wait for JavaScript to load. Some elements of code may be repeated across client and server. Data fetching is event driven and may require load time depending on size of the data.
How to improve: We can minimize the JavaScript bundle size to reduce load time. We can preload the JavaScript. We can do lazy loading that is only load must need things. We can also load dynamically by code splitting.

2. **Incremental static generation:**
   In SSG there is problem while loading the page with dynamic data. This iSSG helps to solve that dynamic data problem by updating existing pages and adding new one by prerendering. The lazy loading concept is used to include new pages on the website after the build. This means that the new page is generated immediately on the first request. To re-render an existing page, a suitable timeout is defined for the page. This will ensure that the page is revalidated whenever the defined timeout period has elapsed.
   Advantages: Dynamic data, Speed, Availability, Consistent.
3. **Progressive hydration:**
   In a server rendered applications server generates HTML, CSS, JSON. Then it sends the data to the client so that client quickly display it on the screen. But only components are loaded, JavaScript bundles are not loaded at the time which result in non-interactive layout. Hydration means attaching the JavaScript bundle to the components on screen. In progressive hydration instead of hydrating entire page at once it starts with root of the DOM, and progressively hydrates other nodes.
   Pros: Promotes code splitting, allows on-demand loading for infrequently used parts of the page, **Reduces bundle size.**

Cons: progressive hydration may not be suitable for dynamic apps where every element on the screen is available to the user and needs to be made interactive on load.

4. **Selective hydration:**

   In SSR first entire tree is generated it is sent to the client. Then it starts hydrating a tree. It has some problems like before the server-rendered HTML tree can get sent to the client, all components need to be ready. This means that components that may rely on an external API call or any process that could cause some delays, might end up blocking smaller components from being rendered quickly.[redacted] Another issue is the fact that React only hydrates the tree once.[redacted]Instead of using the **renderToString** method we are using here **pipeToNodeStream** method on the server.[redacted]This method, in combination with the **createRoot** method and **Suspense**, makes it possible to start streaming HTML without having to wait for the larger components to be ready, which makes possible lazy load of the components.[redacted]Components can be hydrated as soon as they're streamed to the client, since we no longer must wait for all JavaScript to load to start hydrating and can start interacting with the app before all components have been hydrated.

5. **Server-side rendering:**

   Server-side rendering (SSR) is one of the oldest methods of rendering web content. SSR generates the full HTML for the page content to be rendered in response to a user request. The connect and fetch operations are handled on the server. HTML required to format the content is also generated on the server. As such, rendering code is not required on the client and the JavaScript corresponding to this need not be sent to the client. With SSR every request is treated independently and will be processed as a new request by the server. Even if the output of two consecutive requests is not very different.
   Pros: Lesser JavaScript leads to quicker FCP and TTI
   Provides additional budget for client-side JavaScript.
   Cons: Full page reloads required for some interactions.

6. **Static rendering:**

   Static rendering or static generation (SSG) delivers pre-rendered HTML content to the client that was generated when the site was built. A static HTML file is generated ahead of time corresponding to each route that the user can access.[redacted]Static content like that in 'About us' or 'Contact us' pages may be rendered as-is without getting data from a

data-store. However, for content like individual blog pages or product pages, the data from a data-store must be merged with a specific template and then rendered to HTML at build time.
Key Considerations: A large number of HTML files in the data store. SSG site to be super-fast and respond quickly, the hosting platform used to store and serve the HTML files should also be good. n SSG site needs to be built and re-deployed every time the content changes. This makes SSG unsuitable for highly dynamic content.

7. **Streaming server-side rendering:**
   We can reduce the Time to Interactive while still server rendering our application by *streaming server rendering* the contents of our application. Instead of generating one large HTML file containing the necessary markup for the current navigation, we can split it up into smaller chunks! Node streams allow us to stream data into the response object, which means that we can continuously send data down to the client. The moment the client receives the chunks of data, it can start rendering the contents. streaming implies chunks of HTML are streamed from the node server to the client as they are generated. If the network is clogged and not able to transfer any more bytes, the renderer gets a signal and stops streaming till the network is cleared up. Thus, the server uses less memory and is more responsive to I/O conditions.
8. **Island Architecture:**
   Islands are a component-based architecture that suggests a compartmentalized view of the page with static and dynamic islands. The static regions of the page are pure non-interactive HTML and do not need hydration. The dynamic regions are a combination of HTML and scripts capable of rehydrating themselves after rendering. The Islands architecture facilitates server-side rendering of pages with all their static content. However, the rendered HTML will include placeholders for dynamic content.
   Pros: Reduces the amount of JavaScript code shipped to the client. The code sent only consists of the script required for interactive components.
   Prioritizes important content, Accessibility, The architecture offers all advantages of component-based architecture, such as reusability and maintainability.
   Cons: The only options available to developers to implement Islands are to use one of the few frameworks available or develop the

architecture yourself. The architecture is not suitable for highly interactive pages like social media apps which would probably require thousands of islands.

9. **Animating view API:**
   Animating view API offers a simple way to transition any visual DOM change from one state to the next. This might include small changes such as toggling some content, or broader changes such as navigating from one page to the next.