Q. Write a report on your understanding of Rendering and Design Patterns. Mention and elaborate where a particular Rendering pattern is applicable and is well suited for which use case.

→ Ans.

1. Rendering Patttern

Following are the types of Rendering pattern

✨ CSR : Client Side Rendering

✨ SSR : Server Side Rendering

✨ SSG : Static Site Generation

✨ ISR : Incremental Static Regeneration

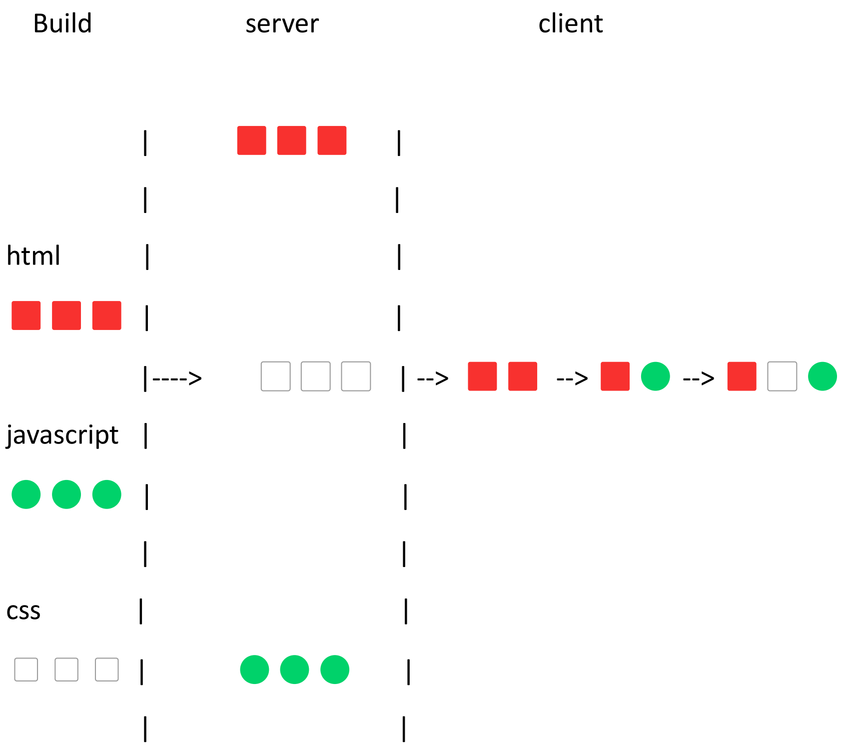
What is Build process ?

source code --> build --> server --> client (brower)

first the source code is passed to build process, that it the code is stored on to the server. and then sent on to the brower.

e.g. npm run build / dev etc.

1) CSR : client side Rendering



- the build phase is something where you write the code

- on the sever all your code is kept seperate and all the html, css and js are kept seperate.

- and now on client side first we send html, and then javascript is sent and as required the js is sent and css is also added.

- this is the core react approach, where everthing will happen through javascript and we sent javascript to the client.

- since empty html page is thrown on client side, it is difficult for search engines. There is no content there, but it is actually created when client visites the page.

- here the web page is rendered/created on client side

<u>Where it is used?</u>

- Single Page Applications (SPAs):

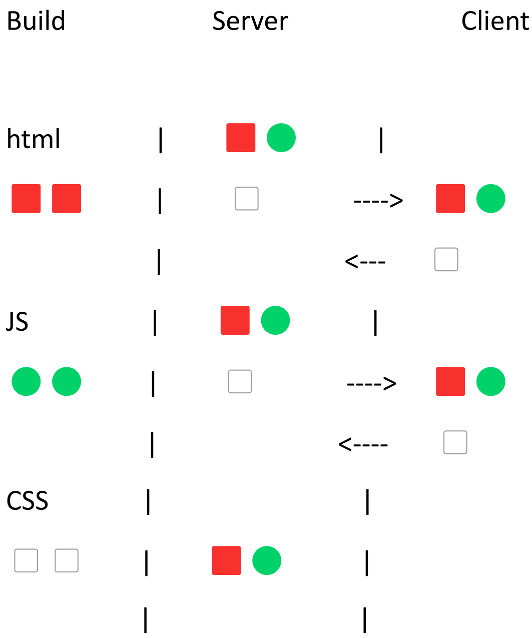
  SPAs load a single HTML page and dynamically update the content as the user interacts with the application. The rendering is done on the client side using JavaScript frameworks like React, Angular, or Vue.js. This provides a smoother and more interactive user experience.

- Offline Capabilities:

  With client-side rendering, it is possible to cache resources and enable offline access to certain parts of the application. This is beneficial for users in environments with limited or intermittent connectivity.

---

2) Server-side Rendering



- here all the html, css and js are loaded on server itself.

- and makes request to the server and on each request new web page is loaded.

- server has more power than the normal browser cause user uses his mobile phone, or laptop which have very limited ram and processign power.

- so, sending everying on server is beneficial as it is expandable too.

<u>Where it is used?</u>

- Search Engine Optimization (SEO):

  SSR is beneficial for SEO because search engine crawlers can easily index the content that is rendered on the server. This is important for websites that rely on search engine visibility, as client-side rendering may face challenges in terms of search engine indexing.

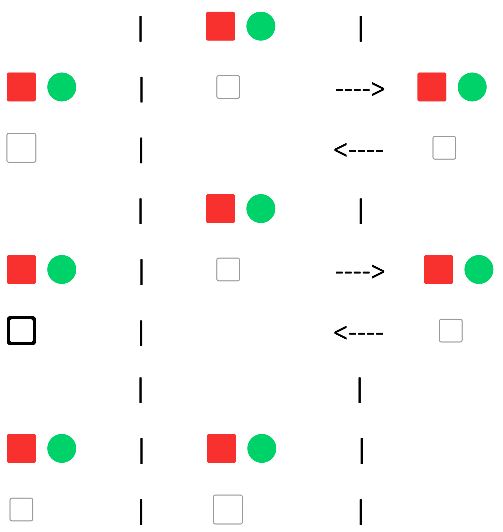
- Improved Initial Page Load Time:

For websites that prioritize fast initial page load times, server-side rendering can provide a better experience. The server sends a fully rendered HTML page to the client, reducing the time it takes for the user to see meaningful content, especially on the first visit.

---

3) SSG : Static Site Generation

Build Server Client



here, the html css and javascript are combined together at the time of build only.

hence it takes a lot of time but it is the most efficient way.

creating web pages at build time.

<u>Where it is used?</u>

- Content-Based Websites:

  Static site generation is well-suited for content-based websites, such as blogs, documentation sites, and news portals, where the content doesn't change frequently

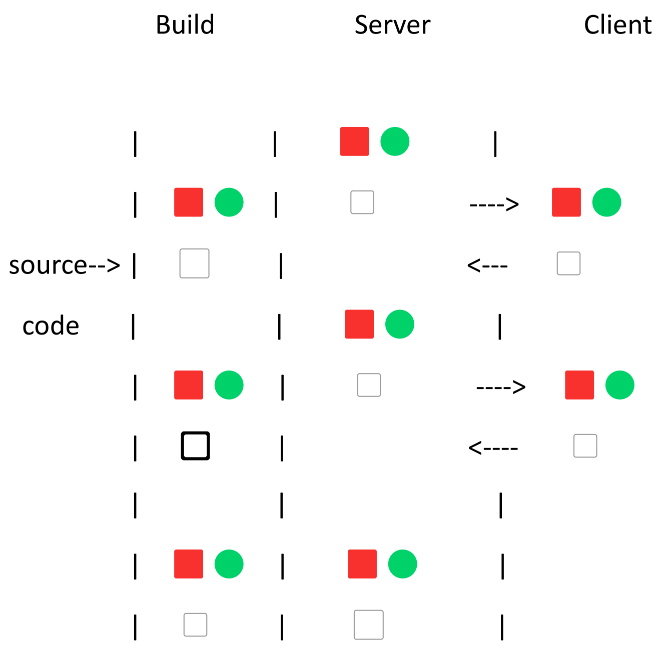
- more performance and speed

  Static sites are fast because they don't require server-side processing for each request.

- hosting is also cost effective

---

4) ISR : Incremental Static Regeneration



- in SSR we were unable to get the updates quickly, whenever the next phase is build then the updates are pushed.

e.g. let say you want to change some content after every 2 days, and you don't know when the next build phase is, so it is not convinient.

- but, in ISR after every some time the build phase will happen, so you can get the updated content in real time.

<u>Where it is used ?</u>

➢ Large Sites with Frequently Changing Content:

For large websites with a considerable amount of content, regenerating the entire site can be time-consuming and resource-intensive. Incremental static site generation allows developers to update only the parts of the site that have changed, reducing build times.

➢ Frequent Content Updates:

Websites that require frequent content updates, such as news sites, blogs, or e-commerce platforms, can benefit from ISG. Instead of regenerating the entire site for every new piece of content, only the affected pages are regenerated, improving efficiency.

2. Desing Patterns

❖ Creational Patterns

- it deal with the process of object creation. They provide solutions to the problem of how to instantiate objects in a flexible and efficient way, without specifying their exact classes this focuses on 2 thigns.

- abstraction

- hiding

there are 5 Categories

👉 Abstract factory DP

- it provides an interface for creating related objects without specifying their concrete class.

- multiple look and feel

👉 Builder DP

- Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

- builder : defines the interface

concreteBuilder : implements the interface

director : constructs an object by using interface provided by builder

product : reprsents the object under constructions

code:

```
class Product

{

private:

   string partA_, partB_, partC_;

public:

   // Methods to set different parts of the product

   // only defining the interface

   void setPartA(const string& partA) {

      partA_ = partA;

   }

   void setPartB(const string& partB) {

      partB_ = partB;

   }

   void setPartC(const std::string& partC) {
```

```
        partC_ = partC;
    }
    void show() {
        std::cout << "Part A: " << partA_ << "\n";
        std::cout << "Part B: " << partB_ << "\n";
        std::cout << "Part C: " << partC_ << "\n";
    }
};
class Builder {
public:
    virtual void buildPartA() = 0;
    virtual void buildPartB() = 0;
    virtual void buildPartC() = 0;
    virtual Product* getResult() = 0;
}
class ConcreteBuilder : public Builder {
private:
    Product* product;
public:
    // construction of methods
    ConcreteBuilder() : product(new Product()) {}

    void buildPartA() override {
        product->setPartA("A");
    }
    void buildPartB() override {
        product->setPartB("B");
    }
    void buildPartC() override {
        product->setPartC("C");
    }
    Product* getResult() override {
        return product;
    }
};
```

----

3. Factory DP

- it defines an inerface for creating an object, but the subclasses decides which class to instantiate.

- all the implementation details are stored in one class, and whenever a particular class want's to access it, he can use it.

e.g. creating desktop application where two abstract classes are there

> application

> documentation

e.g. creating a drawing application

- drawingApplication
- drawingDocumentation

structure:

- again the same participants are there,

> product

> concreteProduct()

Code

```
class Product {
public:
   virtual void create() = 0;
};
class ConcreteProductA : public Product {
public:
   void create() override {
      // Implementation for creating Product A
   }
};

class ConcreteProductB : public Product {
public:
   void create() override {
      // Implementation for creating Product B
   }
};
```

4. Prototpye dp

- prototype is a template for any object before the actual object is constructed.
- you will clone the interface and then can add the functionalities to it.

structure:

- again there will be participants
  > client (calls prototype())
  > prototype (defines the interface)
  > concretePrototype1 (constructs the product)
  > concretePrototype2 (constructs the product)

```
class Prototype {
public:
   virtual Prototype* clone() = 0;
   virtual void use() = 0;
};

class ConcretePrototype : public Prototype {
public:
   Prototype* clone() override {
      return new ConcretePrototype(*this);
   }

   void use() override {
      // Implementation for using the cloned object
   }
};
```

----

5. Singletone

- it ensures that a class has single instance only and provides a global point of access to it meaning that, can be accessed from outside of the class as well.
- sometimes we want only one window manager or just one factory for a product, any other functionalities you don't want

strecture:

Singletone

----

static instance()   <---------- return uniqueinstance

SingletoneOperation()

code:

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {} // Private constructor to prevent instantiation.

public:
    static Singleton* getInstance() {
        if (!instance) {
            instance = new Singleton();
        }
        return instance;
    }
};

// Usage:
Singleton* obj = Singleton::getInstance();
```

2. Desing Patterns

#️⃣ Creational Patterns

- it deal with the process of object creation. They provide solutions to the problem of how to instantiate objects in a flexible and efficient way, without specifying their exact classes this focuses on 2 thigns.

    - abstraction
    - hiding

there are 5 Categories

👉 Abstract factory DP

- it provides an interface for creating related objects without specifying their concrete class.
- multiple look and feel

👉 Builder DP

- Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

- builder : defines the interface
  concreteBuilder : implements the interface
  director : constructs an object by using interface provided by builder
  product : reprsents the object under construction

code:

```
class Product
{
private:
    string partA_, partB_, partC_;

public:
    // Methods to set different parts of the product
    // only defining the interface
    void setPartA(const string& partA) {
        partA_ = partA;
    }

    void setPartB(const string& partB) {
        partB_ = partB;
    }

    void setPartC(const std::string& partC) {
        partC_ = partC;
    }

    void show() {
        std::cout << "Part A: " << partA_ << "\n";
        std::cout << "Part B: " << partB_ << "\n";
        std::cout << "Part C: " << partC_ << "\n";
    }
};

class Builder {
public:
```

```
        virtual void buildPartA() = 0;
        virtual void buildPartB() = 0;
        virtual void buildPartC() = 0;
        virtual Product* getResult() = 0;
    };

    class ConcreteBuilder : public Builder {
    private:
        Product* product;

    public:
        // construction of methods
        ConcreteBuilder() : product(new Product()) {}

        void buildPartA() override {
            product->setPartA("A");
        }

        void buildPartB() override {
            product->setPartB("B");
        }

        void buildPartC() override {
            product->setPartC("C");
        }

        Product* getResult() override {
            return product;
        }
    };
```

----

3. Factory DP

- it defines an inerface for creating an object, but the subclasses decides which class to instantiate.
- all the implementation details are stored in one class, and whenever a particular class want's to access it, he can use it.

e.g. creating desktop application where two abstract classes are there

> application

> documentation

e.g. creating a drawing application

- drawingApplication
- drawingDocumentation

structure:

- again the same participants are there,
  - > product
  - > concreteProduct()

#️⃣ Code

```
    class Product {
    public:
        virtual void create() = 0;
    };

    class ConcreteProductA : public Product {
```

```
        public:
            void create() override {
                // Implementation for creating Product A
            }
        };

        class ConcreteProductB : public Product {
        public:
            void create() override {
                // Implementation for creating Product B
            }
        };
```

----

4. Prototpye dp

- prototype is a template for any object before the actual object is constructed.
- you will clone the interface and then can add the functionalities to it.

structure:

- again there will be participants
    > client (calls prototype())
    > prototype (defines the interface)
    > concretePrototype1 (constructs the product)
    > concretePrototype2 (constructs the product)

```
        class Prototype {
        public:
            virtual Prototype* clone() = 0;
            virtual void use() = 0;
        };

        class ConcretePrototype : public Prototype {
        public:
            Prototype* clone() override {
                return new ConcretePrototype(*this);
            }

            void use() override {
                // Implementation for using the cloned object
            }
        };
```

----

5. Singletone

- it ensures that a class has single instance only and provides a global point of access to it meaning that, can be accessed from outside of the class as well.
- sometimes we want only one window manager or just one factory for a product, any other functionalities you don't want

structure:

```
        Singletone
        -----------------
        static instance()       <---------- return uniqueinstance
        SingletoneOperation()
```

code:

```cpp
        class Singleton {
        private:
            static Singleton* instance;
            Singleton() {} // Private constructor to prevent instantiation.

        public:
            static Singleton* getInstance() {
                if (!instance) {
                    instance = new Singleton();
                }
                return instance;
            }
        };

        // Usage:
        Singleton* obj = Singleton::getInstance();
```

#️⃣ Behavioral Patterns

- they are concerned with assignment of responsibility between objects.
- they also descriebes the communication between them.

here we have 11 patterns.

2 are class patterns which uses inheritance
9 are object

1. chain of responsibility
    it avoids the coupling of a senders
2. command
3. interpreter
4. iterator
5. mediator
6. memento
7. observer
8. state
9. strategy
10. teplate method
11. visotor

Behavioral design patterns are a category of design patterns that focus on how objects interact, communicate, and collaborate with each other. These patterns define the ways in which objects distribute responsibilities and encapsulate behavior. Here are some commonly used behavioral design patterns:

1. Chain of Responsibility Pattern:
   - Purpose: Passes a request 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.
   - Use case: When you want to avoid coupling the sender of a request to its receiver.

    🃏 --> 🃏 --> 🃏 --> 🃏
    ❎   ✅
    if one fails then send to the next

    - during the run time you need to specify the set of objects, that will handle the request.

----

2. Command Pattern:

-Intent: Here you will wrap the request into an object in the form of command and this will be passed to invoker object.
Invoker object will search for that particular object which can handle this command.

- Use case: When you want to parameterize objects with operations, queue requests, or log the operations.

- support undo operation.

----

3. Interpreter Pattern:
- Purpose: Defines a grammar for interpreting the sentences in a language and provides an interpreter to evaluate these sentences.

- Use case: When you need to interpret sentences of a language and represent them as a set of abstract syntax trees.

- e.g. searching a string that match a pattern.
interpreter provides a set of algorithms that are pre defined.

----

4. Iterator Pattern:
- Purpose: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Use case: When you want to traverse a collection of objects without exposing its internal structure.
e.g. array, list. hence we are accessing elements of this sequentially.

----

5. Mediator Pattern:

- Purpose: Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly.
- Use case: When you want to reduce direct connections between multiple classes or systems.

----

6. Memento Pattern:
- Purpose: Captures and externalizes an object's internal state so that the object can be restored to this state later.
- Use case: When you need to capture and restore an object's state, such as implementing undo mechanisms.
-

----

7. Observer Pattern:
- Purpose: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Use case: When you need to establish a communication mechanism between objects where one object changes affect others.

----

8. State Pattern:
   - Purpose: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
   - Use case: When an object's behavior depends on its state and changes in state should be reflected in its behavior.

----

9. Strategy Pattern:
   - Purpose: Defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
   - Use case: When you want to define a family of algorithms, encapsulate each one, and make them interchangeable.

----

10. Visitor Pattern:
    - Purpose: Represents an operation to be performed on the elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates.
    - Use case: When you want to define new operations on a structure of objects without changing their classes.

These patterns provide solutions to various communication and collaboration challenges in software design, allowing for more flexible and maintainable systems. Choosing the appropriate behavioral pattern depends on the specific requirements and dynamics of your application.

#️⃣ Structural Desing Patterns

- they are more concentrated on how the classes and objects are componsed to form larger structure.

- we have class and object Structural patterns.

classes will use inheritance and defines implementation
and objects are defining the way to cretate or compose them.

- Adaptor will only be the class and others are objects.

ABCDFFP
Adaptor
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

1. Adaptor Pattern

- converts the interface of a class into another interface that client wants.
- it wraps existing class into new interface.
- when you want to use existing class but the inerface is not matching with the client's expectations, then you will make use of inheritance.

----

2. Bridge Pattern

- it seperates the implementation and abstraction so that both can vary independently.

e.g. let say you have two process schedulers and two OS.

2 Process schedulers --> (a, b)
2 OS --> (x, y)

now you want to os to execute hte process, so what combinations can be made are as follows
    - (a, x)
    - (a, y)
    - (b, x)
    - (b, y)

hence, everything is implented independently.

----

3. Composite pattern
- compose objects into tree structures

----

4. Decorator Pattern:

Purpose: Attaches additional responsibilities to an object dynamically.
Use case: When you want to extend the functionality of objects without altering their structure.

- applying inheritance

- participants
    > component (defines interface)
    > concrete component (defines object to which you want to add additional responsibilities)
    > Decorator
    > concreteDecorator1
    > concreteDecorator2
----

5. Facade Pattern:

Purpose: Provides a simplified interface to a set of interfaces in a subsystem.
Use case: When you want to define a higher-level interface that makes the subsystem easier to use.

- no. of dependencies are reduced for recuding complexities

```
                        Facade
                      /    |    \
                    /      |     \
    Facade is receiving no. of requests, and they are distributed accordingly.
```

----

6. Flyweight Pattern:

Purpose: Uses sharing to support large numbers of fine-grained objects efficiently.
Use case: When you need to support a vast number of similar objects in an efficient manner.

- it reduces the no. of objects.
- if there are similar kind of objects then it reduces to single one, other wise not.

e.g. in your game there are 10,000 soldiers and all are similiar then you won't create 10,000 objects, but use the same object for all

----

7. Proxy Pattern:

Proxy means in place of.

Purpose: Provides a surrogate or placeholder for another object to control access to it.
Use case: When you want to add a level of control over the access to an object.
