**Creational Pattern**

* Consider applying creational patterns when:
* A system should be independent of how its objects and products are created.
* A set of related objects is designed to be used together.
* Hiding the implementations of a class library or product, revealing only their interfaces.
* Constructing different representation of independent complex objects.
* A class wants its subclass to implement the object it creates.
* The class instantiations are specified at run-time.
* There must be a single instance and client can access this instance at all times.
* Instance should be extensible without being modified.

**Abstract factory**: so that the

The Abstract Factory design pattern solves problems like:

* How can an application be independent of how its objects are created?
* How can a class be independent of how the objects it requires are created?
* How can families of related or dependent objects be created?

**Builder**: The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

The Builder design pattern solves problems like:[2]

* How can a class (the same construction process) create different representations of a complex object?
* How can a class that includes creating a complex object be simplified?

eg : Pizza

**Factory:** How can an object be created so that subclasses can redefine which class to instantiate? let the subclasses define the implementation

The Factory Method design pattern solves problems like: [2]

* How can an object be created so that subclasses can redefine which class to instantiate?
* How can a class defer instantiation to subclasses?

eg : MagicRoom

**Prototype:** It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

The Prototype design pattern solves problems like:

* How can objects be created so that which objects to create can be specified at run-time?
* How can dynamically loaded classes be instantiated?

**Adapter**: Define a separate Adapter class that converts the (incompatible) interface of a class (Adaptee) into another interface (Target) clients require.  
Work through an Adapter to work with (reuse) classes that do not have the required interface.

The Adapter design pattern solves problems like:

* How can a class be reused that does not have an interface that a client requires?
* How can classes that have incompatible interfaces work together?
* How can an alternative interface be provided for a class?

eg: Indian to US pin converter

**Bridge:** Decouple an abstraction from its implementation allowing the two to vary independently.

imagine if you had a few implementations of a data store: one is efficient in space, the other is efficient in raw performance... and you have a business case for offering both in your app or framework.

What problems can the Bridge design pattern solve?

* An abstraction and its implementation should be defined and extended independently from each other.
* A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

**Decorator**: Dynamically adds responsibility to the interface by wrapping the original code

What problems can the Decorator design pattern solve?

* Responsibilities should be added to (and removed from) an object dynamically at run-time.
* A flexible alternative to subclassing for extending functionality should be provided.

eg Pizza

**Delegation**: Support "composition over inheritance"

What problems can the Facade design pattern solve? [[2]](https://en.wikipedia.org/wiki/Facade_pattern#cite_note-2)

* To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
* The dependencies on a subsystem should be minimized.

**Flyweight:** Use sharing to support large numbers of similar objects efficiently.

What problems can the Flyweight design pattern solve? [[2]](https://en.wikipedia.org/wiki/Flyweight_pattern#cite_note-2)

* Large numbers of objects should be supported efficiently.
* Creating large numbers of objects should be avoided.

Eg : car dealer selling series 1 series 2 car

**Proxy:** Provide a surrogate or placeholder for another object to control access to it.

What problems can the Proxy design pattern solve? [[2]](https://en.wikipedia.org/wiki/Proxy_pattern#cite_note-2)[[edit](https://en.wikipedia.org/w/index.php?title=Proxy_pattern&action=edit&section=2)]

* The access to an object should be controlled .
* Additional functionality should be provided when accessing an object.

**Chain of Responsibility:**

What problems can the Chain of Responsibility design pattern solve? [[3]](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern#cite_note-3)

* Coupling the sender of a request to its receiver should be avoided.
* It should be possible that more than one receiver can handle a request.

Eg: purchase order

**Command Pattern:**

What problems can the Command design pattern solve? [[2]](https://en.wikipedia.org/wiki/Command_pattern#cite_note-2)

* Coupling the invoker of a request to a particular request should be avoided. That is, hard-wired requests should be avoided.
* It should be possible to configure an object (that invokes a request) with a request.

Uses

It is possible to send whole command objects across the network to be executed on the other machines, for example player actions in computer games.

Suppose a program has a sequence of commands that it executes in order. If each command object has a getEstimatedDuration() method, the program can easily estimate the total duration. It can show a progress bar that meaningfully reflects how close the program is to completing all the tasks.

The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object

**Interpreter Pattern:**

the **interpreter pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that specifies how to evaluate sentences in a language.

What problems can the Interpreter design pattern solve? [[3]](https://en.wikipedia.org/wiki/Interpreter_pattern#cite_note-3)

* A [grammar](https://en.wikipedia.org/wiki/Backus-Naur_form) for a simple language should be defined
* so that sentences in the language can be interpreted.

Uses:

* Specialized database query languages such as [SQL](https://en.wikipedia.org/wiki/SQL).
* Specialized computer languages that are often used to describe communication protocols.
* Most general-purpose computer languages actually incorporate several specialized languages.

**Mediator Pattern:**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the **mediator pattern** defines an object that encapsulates how a set of objects interact. This pattern is considered to be a [behavioral pattern](https://en.wikipedia.org/wiki/Behavioral_pattern) due to the way it can alter the program's running behavior.

With the **mediator pattern**, communication between objects is encapsulated within a **mediator** object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing [coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming)).

What problems can the Mediator design pattern solve? [[2]](https://en.wikipedia.org/wiki/Mediator_pattern#cite_note-2)

* Tight coupling between a set of interacting objects should be avoided.
* It should be possible to change the interaction between a set of objects independently.

**Creational Design Pattern:**

* Prototype: fully initialized instance to be copied/cloned eg chess game, they have a initial setup, creating this setup every-time would involve efforts, we can store the initial setup and return this copy when asked for a instance, instead of creating initial setup again and again, we just use the prototype
* Builder: Separates object construction from its representation, used when there is complex object structure. Eg Multi course dinner: drink, starter, main course and desert. Providing easy interface for building. Now a person says I want this this…
* Singleton
* Factory:

**Behavioral Design Pattern:**

Behavioral design patterns are concerned with **the interaction and responsibility of objects.**

In these design patterns,**the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

How objects interact

* Chain of Responsibility Pattern:
* Iterator
* State Pattern: Alter state object behavior, fan wall controller, we mention after this state what should be the next state
* Strategy Pattern: Encapsulates algo in a class, based on run time we select which approach to take, eg sorting based on what object is passed
* Observer Pattern: Change in one object should be notified to other, eg online bid, sachin century notification to fans
* Visitor Pattern: adding new functionality without change, no is-a relation, eg travelling in cab, user is no longer in controller in cab
* Template Method Pattern: Defer the exact stages of algorithm to sub classes, concrete classes would be implementing the class, eg : how we build house, there is template to house plan, plumbing and etc is already done, if there is need for customization we can extent it and do it.
* Command Pattern: how things work in restaurant, order given-> order noted by waiter-> waiter takes order to chef and he knows how to execute all this, eg runnable interface and you call start method.
* Momento Pattern: restoring objects previous stage, undo redo, goes to previous state, by storing states at different points.
* Mediator Pattern: Defines simplified communication between classes, eg Air traffic controller 100 flight flying, each flight decides when to land, the controller is one who is orchestrating this.

**Structural Design Pattern:**

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

How objects are composed

* Proxy: Object representing another object, eg debit card, where is the actual transaction happening, in the bank so this debit card is a proxy, object acting like another object, it hides complexity
* Decorator Pattern: Add Responsibility to object Dynamically, pizza example
* Façade Pattern: Single class representing entire subsystem, event manager. Organize event, book place, food etc.. doing everything our self we use event manager, we need to tell I need these things. Reduce Network cost, Reduced Coupling (web layer only needs to know about sub systems), Helps in establishing transaction boundary. Web application frequently used
* Adapter pattern:
* Flyweight: Store as much as possible, and we would try and reuse that object