Real Life Examples of Patterns :

1. **Builder Patterns :**

* EX1 : Building Pizza or Restaurant order based on Pizza & Menu Options

Pizza pizza = new Pizza.Builder(Size.medium).onion(true).olives(true).build();

Pizza(Size size, Boolean onion, Boolean cheese, Boolean olives, Boolean tomato, Boolean corn, Boolean mushroom, Sauce sauceType);

* **Ex2 : Building an XML document with HTML** elements (<html>,<h1>,<h2>, <body>,<p> and etc) 🡺 Builder Patterns
* Ex3 : **Building a smartphone object with attributes like RAM, size, resolution, OS, waterproof and so on.** 🡺 Builder Patterns

Factory Methods :

A super class has multiple sub-classes and based on input, need to return one of the sub-class.

**Usage examples:**

1. **Logger Framework** where the log messages maybe written into the log file (represented by **FileLogger** class) or displayed in the console (represented by **ConsoleLogger** class). Email/Kafka/JDBCLOgger etc
2. Java JDK is widely using the Factory pattern, for example valueOf() method in wrapper classes like String, Boolean and etc.
3. Each time when we have family of different kind of objects that created according to some logic:

* different databases maybe supported: Oracle, SQLServer, H2
* different kind of employees: developer, tester, manager

# Abstract Factory

This pattern captures how **to create families of related product objects,** **without instantiating classes directly**.

**It`s a super-factory which creates other factories.**

## Consider using this when

there is a **family of factories and you need a super factory for related factories**

**Ex1. .** DeviceProviderFactory

## Usage examples:

Application may have the Abstract DeviceProviderFactory which will be able to detect if the device is local or remote (from Amazon Device Farm), and return the corresponding factory accordingly: LocalDeviceProviderFactory or RemoteDeviceProviderFactory . Each such factory knows how to create device provider per OS type of device: AndroidDeviceProvider, iOSDeviceProvider, WindowsPhoneDeviceProvider

# 4. Prototype

* When using prototype pattern, **the objects are cloned instead of creating the new one**s with constructor, which improves the performance.
* In addition, prototype pattern helps **to minimize complexity in object creation.**

## Consider using this when

* **Creation of the object is very time consuming**

Ex1 1. **Build & Clone Cover letters per Organization** — no need to create the Cover letter for each organization from scratch. Instead, one cover letter will be created in most appealing format and for others only a copy will be created with personalized organization name.

* 2. **Clone Board Game & filling next Move in Chess Game** Chess game — may be used for chess board creation, which may be time consuming. Using Prototype pattern, the board may be cloned, from the already existing board object.

# 5. Flyweight

The Flyweight pattern **defines a structure for sharing objects.** Objects are shared for **at least two reasons: efficiency and consistency**. Flyweight focuses on sharing for **space efficiency**. But objects can be shared only if they don’t define context-dependent state. Flyweight objects have no such state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, Flyweight objects may be shared freely.

## Consider using this when

1. The **number of Objects to be created by application should be huge.**
2. The **object creation is heavy on memory and it can be time consuming too.**
3. **The object should be immutable**

## Usage examples:

1. **Store & Create KeyBoards Characters** : May be used to represent the keyboard characters: one object for ‘a’, one for ‘b’ and so on.
2. **Drawing Different Shapes With Different** colors: When drawing a lot of shapes with different colors: one object for red circle, one object for blue circle and so on. In **case red circle was already created once**, **there is no need to create new such object, since the same object may be reused.**



# 6. Proxy

A proxy can be used in many ways. It can act as a local **representative for an object in a remote address space**. It can represent a large object that **should be loaded on demand and avoids duplication of the same object**. Without the concept of proxies, an **application could be slow, and appear non-responsive**. The proxy might **protect access to a sensitive object.**

## Usage examples:

1. **Image viewer** program that **lists and displays high resolution photos**. The program has to show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.
2. The same for **document editor** that can **embed graphical objects in a document**. **It isn’t necessary to load all pictures when the document is opened,** because not all of these objects will be visible at the same time.
3. The protective **proxy acts as an authorization layer** to **verify if the actual user has access to appropriate content**. An example can be thought about the proxy server which provides restrictive internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
4. Maybe **used also for adding a thread-safe** feature to an **existing class without changing the existing class’s code.**

# 7. Decorator

Decorator pattern **allows behaviour to be added to an individual object**, either statically or dynamically, without affecting the behavior of other objects from the same **class**.

Decorator require the interface of Decorator object to be identical to the decorated object.



Photo by [Licor Beirão](https://unsplash.com/@licorbeirao?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText" \t "_blank) on [Unsplash](https://unsplash.com/search/photos/add-cream?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText" \t "_blank)

## Usage examples:

1. Very useful when need **to measure the time it takes to handle some web service request.** The class **TimerDecorator** with **timer capabilities should be created**, which will start the timer when request received and will stop the timer when response sent back.
2. When need to **add to** **some shape component a border or a shadow functionality.**
3. When need **to add zooming, or scrolling functionalities to a page.**

# 8. Template Method

This pattern may be useful when **we need to define an algorithm step by step**. The **implementation of some of the steps may be different, but the general flow and the order of the steps in the flow must remain unchanged**.

For example, if we have **different types of phone devices** (iOS and Android) which may be connected to the PC with USB, and we have an **application** that **when device is connected it should perform the following steps**: get **device info**, **install** an Agent on device and at the end report that device connected. We know that this flow is constant and want to be sure that it will stay the same also for the **new device types in case will be supported in the future (for exp. Windows Phone).** Suppose we have Abstract class — **DeviceConnector** **and two sub-classes IosDeviceConnector and AndroidDeviceConnector**. We have the **methods: getDeviceInfo(), installAgent() and reportDeviceConnected().** The first to methods will be implemented in the two different ways for Android and iOS, because we are using device specific libraries in order to do that. So, in Abstract Device class those two methods will be abstract and the sub-classes will be forced to add their specific implementation. The last method should have the same implementation for all device types, so it may be implemented in the Abstract Device class itself. But the main question here is, **how we will be sure that all device types (also added by other developers in the future) will always implement those 3 methods in this specific order?** The answer is simple; we will add a method to our Abstract class that will represent the steps of our flow (will include our three methods and in correct order). It`s also important to define this method as **final** so that it could not be overridden and changed by its sub classes. In our example, the Abstract class will look like this:

public abstract class DeviceConnector { public final void connectDevice() { getDeviceInfo(); installAgent(); reportDeviceConnected(); } protected abstract void getDeviceInfo(); protected abstract void installAgent(); public void reportDeviceConnected(){ //Add implementation here }}

## Usage examples:

1. When implementing some **general Parser**, which **loads the data from different sources** (like **CSV file, SQL Server database, some 3rd party tool),** **parses** the data (data from different sources will be parsed in different way and then it may be saved to some location. We will have template methods for load(), parse() and save() methods.
2. When implementing credit cards validator. For different **kind of credit cards** (Visa, MasterCard and etc) the validation algorithm is the same: need to check **expiration** date, **length** of the credit card number, account **status** and etc. But the **actual implementation for each credit card type may be different**.

# 9. Observer (also may be called Publish/Subscribe pattern)

The Observer pattern defines and maintains a dependency between objects. The classic example of Observer is **Model/View/Controller**, where all views are Observers of the model, which is called Observable. All views are notified whenever the model’s state changes. The main idea of the Observer patter is that the Observable class will hold a list of Observers and whenever it wants to broadcast something, it just calls the method on all the observers, one after the other.

## Usage examples:

1. **Board Notifications to Students :**

**Students and Board, students should be notified when new message appear on the board**. **Students are Observers and the board is Observable.**

1. **Notify Players if there is any change in Board Status after move**

The chess game: the players are observers and they get notified when there is any change on the board.

1. Pub-Sub messaging.

MSG Producer 🡺 Channel 🡪 MSG Consumer

When messages were published on some channel — the clients that are listening on this channel will be notified that there is a new message.

# 10. Strategy

Multi Algorithms -🡪 Selected & Picked at runtime

Strategy pattern helps to define a family of algorithms, to encapsulate each one of them and **make them interchangeable and independent from the clients that use them**. With this approach, our system become much more flexible and even the **algorithm may be changed at runtime**. The idea is to use encapsulated family of algorithms as composition within the client’s class instead of inheritance.

## Usage examples:

1. **Select Sorting Algo by sortType :**

Application which should be able to choose a sorting algorithm at runtime (Bubble sort, QuickSort and so on).

1. **Choose Payment Mode/Strategy after selecting Shopping Carts**:

When implementing the shopping site: the user adds items to the basket and by the end on checkout, the user can choose the payment strategy in runtime: PayPal, Credit Card and so on.

1. Game Weapons based on Game Character Types :

In a game where we can **have different characters** and each character **can have multiple weapons** to attack but at a time can use only one weapon. The method attack() will have different implementation depends on which weapon is being used.

1. Useful when a client may need to apply a different compression algorithms

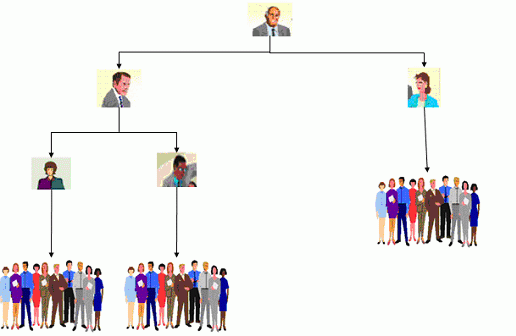
# Conclusion

In this article I described 10 Design Patterns, that in my opinion are very handy in day to day coding tasks. There are much more useful Design Patterns worth checking out. To dive deeper, I suggest you the following books:

1. **Design Patterns: Elements Of Reusable Object-Oriented Software**by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gang of Four)
2. **Head First Design Patterns**:**A Brain-Friendly Guide** by Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson
3. **Thinking in Patterns with Java**by Bruce Eckel
4. **Non-software examples of software design patterns** by Michael Duell, John Goodsen, Linda Rising

## Composite pattern

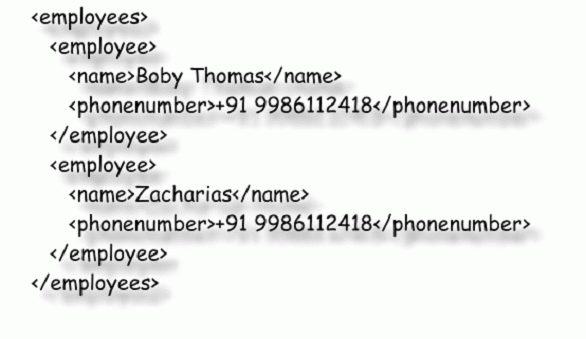
In Composite pattern, you don’t differentiate between a leaf and a composite. The client treats a leaf or a composite the same way. You can correlate this pattern with an organization. An organization consists of many departments. Each department consists of many projects. Each project consists of many project members. A typical organization structure is as shown below.



### **Recursion**

What makes the Composite pattern one of the most beautiful is the power of recursion. I can explain this with the same organization example. You want to find the total salary paid to all employees of the organization. It is nothing but the salary of CEO + the salary paid to all the departments. What is the salary of a department? It is the salary of the department head + the salary of all projects. What is the total salary of a project? It is the salary of the project manager + the salary of all the project members. In a nutshell, the salary of anything is the salary of self + the salary of all its sub groups.

A typical example for this in the software scenario is an XML DOM (Document Object Model) structure. A simple XML structure is shown below.



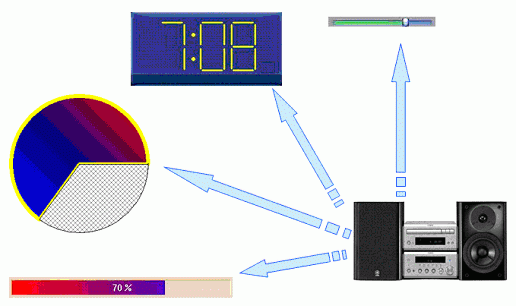
As you can see here, “employees” is a node. “employee” is another node which is a child of employees. What about name in employee? That is also a node. It is nothing but a collection of node objects. Each node can have n-number of child-nodes. If I draw a class diagram of the same, it will be like shown below.

## Observer pattern

A typical example for the Observer pattern is the conventional mail delivery system. Whenever you have a mail, the postman will come and knock at your door with the mail. Just imagine if you have to go to the post office every day and check whether there are any mails. It would have been a really inefficient system as each individual needs to go to the post office periodically. This is eliminated in real life by introducing a mail delivery mechanism.



It is the same in a GUI system as shown below. Your business logic, in this case, the cassette player, will update all the GUI components periodically, just like a post man who comes to your house with mail. It would have been inefficient for the total system to implement a periodic check by each widget to get the progress.



There are two typical models for Observer based on the way data is passed. It can be a push model or a pull model. In the push model, data is pushed along with the notification. An example for this is mail delivery, magazine subscriptions etc. In the pull model, the client will be notified about what is happening. But, it is always the responsibility of the client to check whether the relevant data is there or not. An example for this is RSS feeds. Whenever there is a change in the website, and if you have subscribed for the feed, you will get a notification. But, it is up to you to go back and see if you are interested.

## Factory Pattern and Abstract Factory pattern

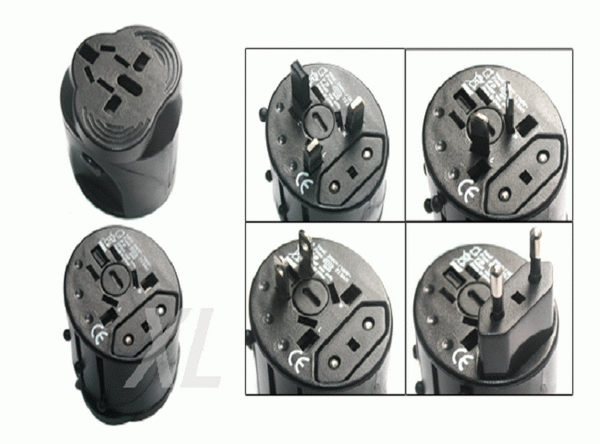
Imagine you are constructing a house and you approach a carpenter for a door. You give the measurement for the door and your requirements, and he will construct a door for you. In this case, the carpenter is a factory of doors. Your specifications are inputs for the factory, and the door is the output or product from the factory.

Now, consider the same example of the door. You can go to a carpenter, or you can go to a plastic door shop or a PVC shop. All of them are door factories. Based on the situation, you decide what kind of factory you need to approach. This is like an Abstract Factory.



## Adapter

Ex1 : Power Adaptor I think this is a self explanatory pattern. You can find many real life examples for this pattern, and the simplest, which is called by the same name, is a power adapter. You have an AC power supply and the responsibility of the adapter is to provide different types of sockets adapted to the one needed.



Ex2 : Language Translator : Another example for this is a translator who is translating languages spoken by one person for another person.

## Façade

**Ex1 : Marriage Event Managers acts as an Façade for arranging all marrrige events & activities including music, dinners, foods, chairs etc.**

Imagine you want to organize a marriage reception. You need to **decorate the hall** where the event is planned. You **need a music band to play** some music. You want to **organize dinner** for say 100 people, and **also snack in the evening** for around 50 people. If you wanted to organize such an event some 10 years back, you wanted to go to a caterer for food, a musical troop for music, and organize flowers and people to decorate them, and so and so. **Some twenty years back, we needed to go to the fish market, the meat shop, the vegetable market etc**., **for the raw materials for the food and** then organize a cook so that you have everything ready for the event. Now, life is really simpler with event managers. **You just need to walk to an event manager and tell him that you need a hall decorated with flowers, dinner for 100 people, snacks for 50 people, and need a good music troop to play music** while the reception is going on. **You don’t need to worry about anything**, **and they will take care of the rest.** **This is a typical example for Façade.**

**You don’t need to interact with many different objects for each and every activity. You will be provided a single interface which will act as a façade for many activities.** This makes the life of clients much simpler.

**Ex2 : Support Desk to pay kinds of Bills & utility**

Other examples for Façade are **one stop bill payment shops**, **a support desk** in a big organization which takes all kinds of support requests etc.

## Decorator

**A decorator decorates something**.

Ex1 : **SungGlasses** :

An example for Decorator is a sunglass. **It takes light as input, removes all harmful parts in the light, and passes it on**. If you give apple as an input to a system, and if it gives orange as output, that is **not** a Decorator.

Ex2 : Another example for this is **music filtering.**

A software example for Decorator is JAVA I/O. Initially, I wondered why we need to create so many objects just to do a file operation. I realized how flexible and wonderful it is after some time.

## Template

Ex1 : A typical example for the Template pattern is the **Computer Engineering degree curriculum**. It covers all the basic stuff required for a typical software engineer. **He can’t work as it is in an industry as he needs to fill in many gaps** which are **required in the specific domain**. But, **he will have the necessary knowledge to move to any software company after completion of the course.**

## Reactor

Recently, I went to a restaurant in Bangalore. There was a female standing outside, receiving all the people entering the restaurant. She received us with a pleasant smile, then took us inside, and helped us find a table. She assigned a waiter to serve us, and then went back to the reception and waited for new clients. All she did was receive a new visitor. She is just **like a web server serving multiple clients**. In a typical web server**, there is a thread simply waiting for requests from clients.** Whenever there is a client request, it creates a separate thread and assigns that thread to handle the request. The thread will die down after serving the requests.

## Conclusion

You can find many more examples like this in our day to day life for real life design patterns. If we correlate these, we will appreciate the patterns better. Most of these real life patterns were evolved over a long period of time by brilliant people to have efficient systems in the society. If we follow this in software systems, we can create efficient software systems just like other efficient systems around you.

### Creational Patterns

Creational patterns **deal with object creation**. They try to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by controlling the object creation and providing a flexible and efficient way to create objects.

Some common examples of creational patterns are:

* **Singleton Pattern**: Ensures that **only** **one instance** **of a class is created**, providing a **global access point to that instance**.
* **Factory Pattern**: Creates **objects without specifying the exact class to create.**
* **Prototype Pattern**: Creates **new objects by copying existing objects**.

### Structural Patterns

Structural patterns **deal with object composition, creating relationships** between objects to form larger structures. These patterns focus on creating relationships between objects to increase the flexibility and reuse of the code.

Some common examples of structural patterns are:

* **Adapter Pattern**: Allows **incompatible classes to work together** by converting the interface of one class into another.
* **Bridge Pattern**: **Decouples an abstraction from its implementation**, allowing the **two to vary independently**.
* **Composite Pattern**: Composes **objects into tree structures** to represent **part-whole hierarchies**.

### Behavioral Patterns

Behavioral patterns **focus on communication between objects and how they operate together**. These patterns use **inheritance** to provide new behavior between objects. By using behavioral design patterns, you can **create object interactions** that are more dynamic and adaptable, allowing your software to **respond to changing requirements and conditions**.

Some common examples of behavioral patterns are:

* **Observer Pattern**: Defines a one-to-many dependency between objects, such that when **one object changes state, all its dependents are notified and updated automatically.**
* **Strategy Pattern**: Allows you to **choose different implementations of the same functionality at runtime,** without overloading methods or writing duplicate code.
* **Template Method Pattern**: Defines the **skeleton of an algorithm** as an **abstract class, allowing subclasses to provide specific implementation details.**

By understanding and applying these different types of design patterns in your work, you can become a more skilled and effective developer and solve complex problems with ease.

Command Patterns : Table Order By Waiter in Restaurant

One example of the command pattern being executed in the real world is the idea of a **table order at a restaurant**: the waiter takes the order, which is a command from the customer. This order is then queued for the kitchen staff.

### The Command Pattern

The Command pattern is known as a **behavioural**pattern,as it's used to manage algorithms, relationships and responsibilities between objects.

**ncapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations**

**Command** declares an interface for all commands, providing a simple **execute()**method which asks the **Receiver**of the command to carry out an operation. The **Receiver**has the knowledge of what to do to carry out the request.  The **Invoker**holds a command and can get the **Command**to execute a request by calling the execute method. The **Client** creates **ConcreteCommand**s and sets a **Receiver**for the command. The **ConcreteCommand**defines a binding between the action and the receiver. When the **Invoker**calls execute the ConcreteCommand will run one or more actions on the Receiver.

### When Would I Use Coomand Pattern?

The Command Pattern is useful when:

* **A history of requests is needed**
* You need **callback** functionality
* **Requests need to be handled at variant times or in variant orders**
* The **invoker should be decoupled from the object handling the invocation.**
* You'll see command being used a lot when you **need to have multiple undo operation**s, where a stack of the recently executed commands are maintained. To implement the undo, all you need to do is get the last Command in the stack and execute it's undo() method.
* You'll also find Command useful **for wizards, progress bars, GUI buttons and menu actions, and other transactional behaviour**.