# **Encapsulation**

## **What is it?**

Encapsulation is a concept that binds together the data and functions that manipulate the data, and keeps both safe from outside interference or misuse. It's a way of structuring data where the data (attributes) and the operations on the data (methods) are bundled together as a single unit - a class.

When an object of the class is created, you can think of encapsulation as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the class.

Access to the data is provided via public methods, often referred to as getters (which allow reading of data attributes) and setters (which allow modifying them), while the data attributes themselves are made private or protected. This way, encapsulation provides control over the data and allows for validation or specific rules to be applied when accessing or modifying it.

## **When do we use it?**

Encapsulation is a fundamental concept in OOP that's used throughout the entire life cycle of an OOP-based project. Here's when it's typically used:

**When defining a class:** Encapsulation is used to hide and protect class data. It involves bundling the data (attributes) and the methods that operate on the data into a single unit, which is called a class.

**When creating objects:** When an object is created from a class, encapsulation allows the object's data to be accessed only through the object's methods, thus preserving the integrity of the data.

**When you want to control data access:** Encapsulation is used when you want to make certain data 'read only' or 'write only'. This is achieved in OOP through getter and setter methods.

**When modifying existing code:** Encapsulation makes it safer and easier to modify your code, because you can change one part of the code without affecting other parts. This is because the internal state of objects is hidden from the outside, and interaction is done through methods, so the internal workings can be changed without affecting external code that uses the object.

## **How is it achieved?**

Encapsulation is achieved by using access modifiers to restrict access to the variables and methods of a class. The three access are:

1. **Private:** A private variable or method can only be accessed within the same class. It cannot be accessed by any other class, even if that class is in the same package.
2. **Protected:** A protected variable or method can be accessed within the same class, as well as by any subclass of that class or any other class in the same package.
3. **Public:** A public variable or method can be accessed by any class in any package.

By using access modifiers, you can control which parts of your class are visible to the outside world. This helps to prevent other parts of the program from accidentally modifying the internal state of an object, which can lead to bugs and other unexpected behavior.

Encapsulation also provides a way to hide the implementation details of a class. Instead of exposing the inner workings of a class, we can provide a public interface that other parts of the program can use to interact with the class. This interface can include methods for getting and setting the values of the class's variables, as well as other operations that the class supports.

Overall, encapsulation is an important principle in OOP that helps to ensure that programs are modular, maintainable, and robust.

# 

# 

# **Inheritance**

## **What is it?**

In object-oriented programming, inheritance is a mechanism that allows a new class called a "subclass" to be based on an existing class called a "superclass", inheriting its attributes and behaviors. The new subclass can then add or override the attributes and behaviors of the superclass as needed.

This allows developers to create classes that build on existing code, rather than starting from scratch each time. Inheritance also helps promote code reusability and maintainability, as changes made to the superclass will automatically propagate to any subclasses that inherit from it.

## **When do we use it?**

Inheritance is used in many real-world scenarios. For example, if you have a class for a general car, you can create a subclass for a sports car that inherits all the properties and methods of the general car but adds new properties and methods specific to the sports car. Similarly, you can create a subclass for a SUV that inherits from the general car class but adds new properties and methods specific to SUVs.

## **How is it achieved?**

Inheritance is a mechanism in object-oriented programming that allows a new class, called the child class, to be based on an existing class, called the parent class.

To achieve inheritance, the child class is defined with the keyword "extends" followed by the name of the parent class. This means that the child class will inherit all the properties and methods of the parent class, and can also add new properties and methods or override existing ones.

# **Polymorphism**

## **What is it?**

Polymorphism is the concept of using a single interface or abstract class to represent multiple types of objects. In other words, it allows objects of different classes to be treated as if they were objects of a common superclass.

Polymorphism can take on several forms, including method overriding and method overloading. Method overriding is when a subclass provides a specific implementation of a method that is already provided by its parent class, while method overloading is when a class provides multiple methods with the same name but different parameter types.

Polymorphism helps to make code more modular, flexible, and reusable by allowing objects to be manipulated in a more generic way without knowing their specific type or implementation details.

## **When do we use it?**

Polymorphism is used when you have different classes with the same method names, but different implementations of those methods. This allows you to write more flexible and reusable code, as you can use a single method to perform different actions based on the type of object that it's called on. Polymorphism is also useful for creating generic code that can work with a variety of different objects, without having to know the specific details of each one.

**Example:**

**Game Development:** In a game, you might have various types of characters like a soldier, an alien, a robot, etc., but they all might inherit from a common 'Character' class. Each character could have a 'move' method, but the way a soldier moves might be different from how an alien or a robot moves. Using polymorphism, the game can simply tell each character to 'move', without needing to know the specific type of character.

## **How is it achieved?**

Polymorphism is typically achieved through inheritance and method overriding. Inheritance is the process of creating a new class that inherits properties and methods from an existing class, known as the base class or parent class. The new class is called the derived class or child class.

Method overriding is the process of providing a new implementation for a method in a derived class that was already defined in the parent class. The method signature (name and parameters) must match that of the parent class method, but the implementation can be different.

Polymorphism allows you to write code that can work with objects of different classes, as long as they implement the same methods with the same signatures. This is achieved through the use of interfaces or abstract classes, which define a common set of methods that must be implemented by any class that wants to use them.

# **Abstraction**

## **What is it?**

Abstraction is a design principle that aims to reduce complexity by hiding unnecessary details from the user. It allows programmers to represent real-world entities in the programming environment in a simplified manner. Abstraction helps to focus on what an object does instead of how it does it.

**For example, consider a car**. A car has various parts like an engine, gearbox, wheels, etc. If you're just driving the car, you don't need to know how all these parts are working together. You only need to know how to interact with the car's controls like the steering wheel, pedals, and buttons. This is similar to how abstraction works in OOP.

In the context of programming, classes often provide abstraction. A class defines the properties (attributes) and methods (behaviors) of an object, but when you create an instance of the class (an object), you don't see the inner workings of these methods; you just see their inputs and outputs. This abstraction makes it easier to work with complex systems, as you don't need to understand every single detail of how everything works - just the high-level interface.

In addition, abstraction also enables better code maintainability and reusability. The internals of a class can be changed without affecting any code that uses the class, as long as the interface stays the same. And classes can be reused in different parts of a program or in different programs, reducing the amount of code that needs to be written.

## **When do we use it?**

It is used extensively throughout the entire design and development process. Here are a few situations where abstraction is used:

**Designing Classes:** Abstraction is used when we define a class in OOP. A class is an abstraction of a real-world entity. For example, a 'Car' class represents the concept of a car, with attributes like color, make, model, and methods like start(), stop(), accelerate(), etc., abstracting the complex inner workings of a real car.

**Creating Interfaces and Abstract Classes:** Interfaces and abstract classes are used to define abstract types that can be used to generalize behavior across related classes. For example, you might define an 'Animal' interface with an abstract method 'makeSound()'. This method is implemented differently by each class that implements the 'Animal' interface, such as 'Dog' or 'Cat'.

**Encapsulation:** Abstraction is employed in the principle of encapsulation, where the internal workings of a class are hidden from the outside world. Only the necessary details are exposed via public methods, and the complexity is abstracted away. This makes the class easier to use and reduces the chance of errors.

**Inheritance and Polymorphism:** Abstraction is also used in inheritance and polymorphism. A superclass or parent class represents a general form (an abstraction), and subclasses or child classes are more specific forms of this abstraction. Polymorphism allows us to treat objects of different subclasses as instances of the parent class, abstracting away the specifics of each subclass.

**Design Patterns:** Many design patterns use abstraction to solve common software design problems. For example, the Factory pattern abstracts away the process of creating objects, and the Strategy pattern abstracts away specific behaviors or algorithms.

Remember, the main goal of abstraction in OOP is to reduce complexity by breaking down complex systems into simpler, more manageable parts. This makes the software easier to understand, develop, test, and maintain.

## **How is it achieved?**

Here are several ways abstraction is achieved in OOP:

**Classes:** In OOP, a class is the blueprint from which individual objects are created. Each class is an abstraction that encapsulates data (attributes) and functions (methods) that operate on the data. The data represents the state of the object, and the methods represent its behavior.

**Interfaces and Abstract Classes:** An interface is a completely "abstract class," which can only contain abstract methods (methods without bodies). An abstract class is a class that cannot be instantiated and is often used as a base class for other classes. Abstract classes and interfaces are ways to achieve abstraction in languages like Java. They define a contract, or set of methods, that classes inheriting from them must implement, but they don't provide implementations of these methods.

**Encapsulation:** Encapsulation is another important aspect of abstraction. It refers to the bundling of data and methods that act on that data within one unit, and restricting access to some of the object's components. This means that the internal state of an object is hidden from the outside world, and can only be manipulated in ways defined by the object. This helps to reduce complexity and increase maintainability.

**Inheritance:** In OOP, classes can inherit properties and methods from other classes. The class being inherited from is the parent class, and the class inheriting from the parent class is the child class. Inheritance allows programmers to create a general class first then later extend it to more specialized classes. This is a form of abstraction, as the parent class often represents a higher-level concept, and the child classes are more specific.

**Polymorphism:** This is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. This allows abstracting functionality in base classes, which can be overridden in derived classes to provide specific implementations.

These mechanisms help developers manage complexity by letting them focus on the relevant details at a given level of abstraction, ignoring the rest.

# **Observer Pattern**

## **What is it?**

The Observer pattern is a software design pattern that establishes a one-to-many relationship between objects so that when one object changes state, all of its dependents (observers) are notified and updated automatically.

The Observer pattern consists of two main player types:

**Subject**: This is the object having methods to attach and detach observers to a client object. It is responsible for maintaining a list of observers and calling their update method whenever its state changes.

**Observer**: This is an interface defining the update method for objects that need to be notified of a Subject's changes of state.

In the context of the Observer pattern, the Subject is usually referred to as the "Observable" and the Observers are often referred to as "Subscribers".

A common use case for the observer pattern is in event-driven programming. For example, GUI elements in a desktop app might change based on changes to application data: if the data changes, the view updates automatically. Another typical use is in distributed systems to notify interested parties when a certain action has taken place.

Overall, the Observer pattern promotes a loose coupling between objects which is a key principle in object-oriented design, making systems easier to extend, modify, and debug.

## **How does it work?**

The subject is an object that has some state or data that can change/update over time. The observers are other objects that are interested in the state of the subject and want to be notified when it changes.

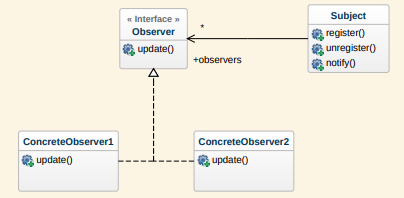
## **When to use it?**

The Observer pattern is useful in situations where you have a one-to-many relationship between objects and need to maintain loose coupling between them.

**Benefits:** The subject does not need to know anything about the observers and while the observers are interested in the subject's state, they do not need to know how the subject's state is changing or why.

**Negative:** The subject may sometimes send updates that do not matter to the observers.

By using this pattern we can make our program more modular, flexible, and easier to maintain over time.



# **Factory Pattern**

## **What is it?**

The Factory Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass but lets subclasses decide which specific object to instantiate.

The Factory Pattern is particularly useful when a system needs to be independent of how its objects are created, composed, and represented. It encapsulates the process of creating and delivering objects, and puts it into a "factory" object. This factory object has a method for every type of object it is capable of creating. The factory object decides which object to create based on the arguments given to it.

This pattern promotes loose coupling and scalability, and is commonly used in situations where a class cannot anticipate the type of objects it needs to create, or when a class wants its subclasses to specify the objects it creates. It also allows the code to defer instantiation to subclasses.

## **How does it work?**

Let’s say we want to use the factory method for creating animals.

This pattern consists of

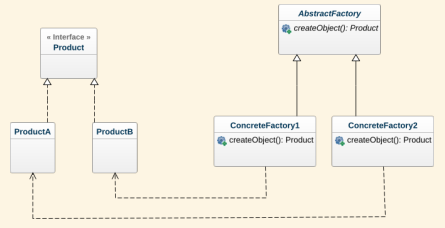
-a **Product** class, which is the animal object, -a **Creator/Factory** class, which is the superclass that creates products (creates animals), it might be abstract since we want to use Polymorphism -**Concrete Product** classes that implement the Product class, they are different products that share similar properties (for example different animals such as dogs, cats or ducks),

-**Concrete Creator/Factory** classes, which are the subclasses that implement the Creator class to create objects of specific types (creates specific animals).

The factory receives information about a Product (animal) the client wants to create, it uses its createObject method to create a product (an animal) and lets the concrete creator class decide the concrete product (cat, dog or duck) that in the end will be instantiated and returned to the user.

## **When to use it?**

The factory pattern is used when we need to create objects that share the same or similar properties, it is used when all potential classes share the same superclass/interface and it is used when we want to encapsulate object creation so that whenever we want to modify something, there is only one place we need to go.



# **Singleton Pattern**

## **What is it?**

The Singleton Pattern is used when we want to eliminate the option of instantiating more than one object from a class, meaning when a class is created, there is only ever going to be one object created from that class.

## **How does it work?**

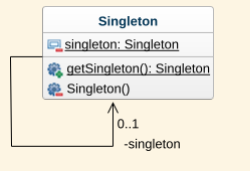
When we define a class as a Singleton, we make sure that it has a private constructor, which means that the class cannot be instantiated directly from outside itself. Instead, we provide a static method that returns and creates an instance of the class.

First, we call the static method to check if an instance of the class already exists. If it does, we simply return the existing instance. If not, the method will create a new instance and return it. The class also includes a private static field that stores the single instance of the class.

One important consideration when implementing the Singleton Design Pattern is thread safety. If multiple threads are accessing the Singleton class, there is a risk that two threads could create separate instances of the class at the same time. To prevent this, the static method should use the synchronize method. By using synchronize, only one thread can access the method at a time, preventing multiple threads from creating multiple instances of the Singleton class simultaneously.

## **When to use it?**

The pattern is useful when we want to ensure that only one instance of a particular class exists in the application, and that can be easily accessed and used by any part of the application. It is often used in situations where we need to ensure consistency and reliability, and where multiple instances of a class could cause issues with data consistency or performance.



# **Iterator Pattern**

## **What is it?**

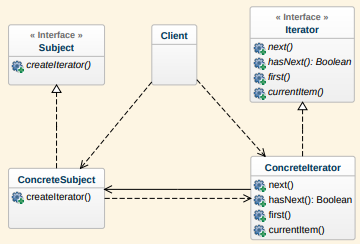
The iterator pattern is a behavioral pattern that provides an interface that allows access to different collections of objects and treats them as if they were the same.

## **How does it work?**

In Java, the Iterator Design Pattern is implemented through the java.util.Iterator interface. This interface provides methods for iterating over a collection, such as hasNext() to check if there are more elements, and next() to get the next element in the collection.

## **When to use it?**

We use the Iterator Design Pattern when we need to access the elements of a collection in a generic way, without relying on the specific implementation of the collection. If we get an Array, Arraylist and Hashtable of Objects, we can implement the iterator interface for each of them and treat them as if they were the same. If the implementation of each of the collections changes, the code that uses the iterator will not need to be modified.



# **Strategy Pattern**

## **What is it?**

The Strategy pattern is a design principle that enables an application to dynamically choose an algorithm or behavior at runtime.

The core concept of the Strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from the clients that use it. Essentially, it allows you to decouple the behavior of an object from its class.

This pattern is particularly useful when a class has a behavior that can be implemented in multiple ways, or when these behaviors can change dynamically. Instead of implementing all possible behaviors, the class can have a reference to a Strategy object, which is used to perform a behavior. The strategy object will be an instance of a subclass of an abstract Strategy class, where each subclass provides a different implementation of a behavior.

Overall, the Strategy pattern offers a method to reduce the complexity of your code, improve maintainability by allowing you to manage behaviors and algorithms separately, and provides a more flexible and scalable solution to changing requirements or business rules.

## **How does it work?**

The strategy pattern is like deciding which route to take to reach a destination. You know you want to get from point A to point B, but you can take a car, a bike, or go by foot. Each method (strategy) will get you to your destination, but the time and effort involved may be different.

In the context of programming, the strategy pattern involves selecting an algorithm to solve a problem at runtime.

**Here's how it works:**

**Strategy:** This is the common interface that the strategies will implement. This is the plan or the approach you're going to take. In programming, this would be an interface that represents the strategy or plan. It defines what actions must be implemented by any strategy.

**Concrete Strategies**: These are the classes that implement the strategy interface. These are the specific plans. In programming, these are classes that implement the Strategy interface and provide a specific way to achieve the strategy. For example, if your strategy is sorting a list, one concrete strategy might be to use quicksort, another could be mergesort, and so on.

**Context:** This is the class that contains a strategy. This is the situation you're dealing with. In programming, the context uses the strategy. It maintains a reference to a Strategy object and uses that to perform some work. It can switch strategies based on its needs.

So, using the travel example, the context could be you needing to reach a destination. The strategy would be the method of transportation, and the concrete strategies would be going by car, bike, or foot. You (the context) can choose any of these strategies based on factors like distance, time, and weather.

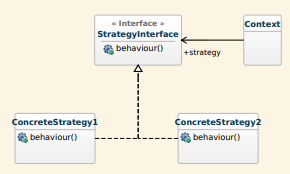
In summary, using the Strategy Pattern, you can easily switch between different algorithms at runtime, without changing the code that uses them. Additionally, you can add new algorithms or modify existing ones without affecting the code that uses them, promoting flexibility and maintainability.

## **When to use it?**

You should use the Strategy pattern when you have a set of related algorithms or behaviors that need to be used interchangeably in a particular context. This pattern can be used to avoid the need for conditional statements or switch statements to choose between different algorithm implementations.

The Strategy pattern is particularly useful when you have a complex algorithm or behavior that needs to be divided into smaller, more manageable pieces. Each piece can be implemented as a separate strategy object, and the context object can choose the appropriate strategy object based on the situation.

Overall, the Strategy pattern is a good choice when you need to provide multiple implementations of an algorithm or behavior and want to allow the client code to choose the appropriate implementation dynamically.



# **Prototype Pattern**

## **What is it?**

The Prototype pattern is a creational design pattern in object-oriented programming. The main idea behind this pattern is to create objects based on a prototype instance, which is cloned to create new objects instead of creating them from scratch.

In this pattern, a prototype object is created and then multiple copies of the prototype are created, with each copy being an exact copy of the prototype. These copies are then modified as required, resulting in new objects with the same structure as the prototype.

## **How does it work?**

The Prototype pattern involves creating a prototype object and then creating multiple copies of that prototype. The copies are created by cloning the prototype and then modifying the cloned copies as needed. Here are the steps to use the Prototype pattern:

**1.** Create a prototype object: Create an object that serves as the prototype for all the other objects that you want to create. This prototype object should contain all the properties and methods that you want to use in the new objects.

**2.** Clone the prototype: Use the clone method to create a new object that is a copy of the prototype. This copy should be a shallow or deep copy, depending on your requirements.

**3.** Customize the copy: Modify the properties and methods of the cloned copy as needed to create a new object with the desired characteristics.

**4.** Repeat as needed: Create multiple copies of the prototype by repeating the clone and customization steps as many times as needed.

Using the Prototype pattern can help you save time and resources by creating new objects more efficiently. By using a prototype object as the basis for all the new objects, you can avoid duplicating code and simplify the creation process. Additionally, the Prototype pattern allows you to customize each new object as needed, giving you more flexibility in your design.

## **When to use it?**

Here are some situations where using the Prototype pattern can be beneficial:

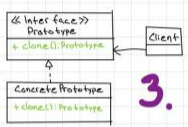
**1.** Creating objects is expensive: If creating new objects is a resource-intensive process, such as when the creation involves heavy computation or data retrieval from a database or network, using the Prototype pattern can help you optimize your application's performance. By creating a prototype object and then cloning it to create new objects, you can save time and resources.

**2.** Creating objects dynamically: If you need to create new objects dynamically at runtime, the Prototype pattern can be helpful. Instead of creating new objects from scratch, you can clone a prototype object and then modify its properties and methods as needed to create the new object.

**3.** Need for object variations: If you need to create objects that vary in their properties or behavior, the Prototype pattern can be useful. By using a prototype object as a template, you can create multiple copies of the object with different properties or behaviors, thus avoiding duplication of code.

**4.** Avoiding subclassing: If you need to create new objects that differ in only a few properties or methods from existing objects, subclassing can be an option. However, using the Prototype pattern can be a more efficient solution as it avoids the need for creating a new subclass for each variation. Instead, you can create a prototype object and then clone it to create new objects with the desired variations.

Overall, the Prototype pattern is useful when you need to create objects efficiently, dynamically, and with variations in their properties and behavior.



# **Memento Pattern**

## **What is it?**

The Memento pattern is a design pattern that allows the capturing and externalizing of an object's internal state without violating its encapsulation. This pattern is used to provide the ability to restore an object to its previous state. The Memento pattern is composed of three main components: the Originator, the Memento, and the Caretaker.

The **Originator** is the object whose state needs to be saved and restored.

The **Memento** is an object that stores the internal state of the Originator.

The **Caretaker** is an object that is responsible for managing the Mementos, and for restoring the Originator's state from a Memento.

The Memento pattern is commonly used in applications where it is necessary to provide an "undo" functionality or to save and restore the state of an object for later use. By using the Memento pattern, an object can save its internal state without exposing it to the outside world, providing a clean and elegant way of implementing state management.

## **How does it work?**

The Memento pattern is used to capture and externalize an object's internal state without violating its encapsulation, and to provide the ability to restore an object to its previous state. To use the Memento pattern, you need to follow these general steps:

**1.** Create the Originator class: This is the class whose state needs to be saved and restored. The Originator class should have a method to create a Memento object that stores its internal state and another method to restore its state from a Memento object.

**2.** Create the Memento class: This is the class that stores the state of the Originator. The Memento class should have methods to get and set the state of the Originator.

**3.** Create the Caretaker class: This is the class that is responsible for managing the Mementos. The Caretaker class should have methods to store and retrieve Mementos.

**4.** Use the Memento pattern: To use the Memento pattern, you first create an instance of the Originator class. You can then manipulate the state of the Originator as needed. When you want to save the state of the Originator, you create a Memento object using the Originator's createMemento() method and store it in the Caretaker. When you want to restore the state of the Originator, you retrieve the Memento from the Caretaker and use the Originator's setMemento() method to restore its state.

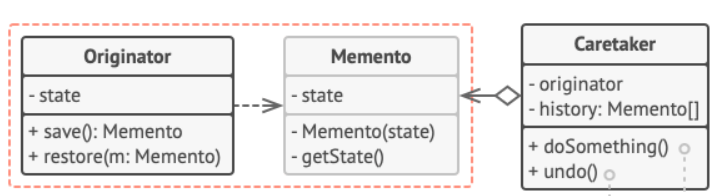
By using the Memento pattern, you can easily implement features such as undo/redo functionality or the ability to save and restore an object's state. The pattern helps to keep your code organized and maintainable by separating the responsibilities of each class involved.

## **When to use it?**

The Memento pattern is useful in situations where you need to provide the ability to undo or revert an action or to save and restore the state of an object. Here are some specific scenarios when the Memento pattern might be a good fit:

* Undo/Redo functionality: The Memento pattern is commonly used to implement undo/redo functionality in applications, allowing users to undo or redo their actions and restore the previous state of an object.
* State management: The Memento pattern can be used to save and restore the state of an object, which is useful in applications where you need to maintain the state of an object across different stages of execution.
* Database transactions: The Memento pattern can be used to implement database transactions, where you need to roll back a transaction if it fails to complete successfully.
* Configuration management: The Memento pattern can be used to manage the configuration of an application, allowing you to save and restore the configuration settings when needed.

By using the Memento pattern, you can improve the flexibility and maintainability of your code, and make it easier to implement features such as undo/redo functionality or the ability to save and restore an object's state.



# **Composite Pattern**

## **What is it?**

The Composite Pattern allows you to treat individual objects and compositions of objects uniformly. It provides a way to organize objects into tree-like structures to represent part-whole hierarchies. The pattern allows clients to work with the individual objects and the groups of objects in a transparent and consistent manner.

In the composite pattern, each component in the tree can be treated as a composite or a leaf node. A composite node can have child nodes, which can be either composite or leaf nodes. This recursive structure enables the creation of complex structures, where any part of the structure can be treated as a whole. The pattern is widely used in designing user interfaces, file systems, and other hierarchical structures.

## **How does it work?**

In software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies.

## **When to use it?**

You can use the Composite Pattern when you want to represent part-whole hierarchies of objects, and you want clients to treat individual objects and groups of objects uniformly.

Here are some situations where the Composite Pattern can be useful:

* When you need to represent a hierarchy of objects, and you want to be able to treat each object in the hierarchy uniformly, regardless of whether it's a composite or a leaf node.
* When you need to add or remove objects dynamically from the hierarchy.
* When you want to perform operations on the objects at different levels of the hierarchy.
* When you want to create complex structures that can be treated as a single object.
* When you want to simplify client code by using the same interface to interact with both the composite and the leaf nodes.

Some common examples where the Composite Pattern can be used are file systems, user interfaces, and graphics systems.

