Design Patterns

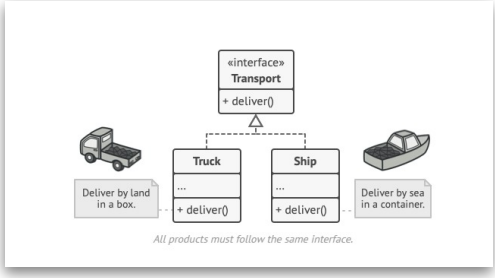
Typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize and implement it in various ways to solve a particular design problem in your code.

1. Creational Design Patterns:

Provide various object creation mechanisms, which increase flexibility and reuse of existing code. Helps make a system independent of how its objects are created, composed and represented

A. Factory method:

Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created (let subclasses decide which specific class or object to instantiate).



Example: in a logistic app initially designed for truck transport this pattern allows you to add sea transportation by creating a common Transport() interface that both truck and ship implement. This way the client code interacts with the Transport interface making it flexible and extensible to support new transport types without significant changes to the codebase.

Purpose: the factory method pattern abstracts the process of objects creation, promoting loose coupling between the creator and the created object

When to use it:

1. Use when you don't know beforehand the exact types and dependencies of the objects your code should work with
2. When you want to provide users of your library or framework with a way to extend its internal components
3. When you want to save system resources by reusing existing objects instead of rebuilding them each time
1. When you want to encapsulate object creation logic
2. When you need to support multiple products variations
3. When object initialization is somewhat expensive

Components:

1. Creator: an abstract class or interface that declares the factory method
2. Concrete creators: subclasses that implement the factory method
3. Product: the common interface for the objects created
4. Concrete products: actual instances of objects created by the factory method

Benefits:

1. Decouples client code from concrete classes
2. Supports extensibility and flexibility
3. Can move the product creation code into one place in the program, making the code easier to support
4. Introduce new types of products into the program without breaking existing client code
5. Simplifies complex object creation

Impacts and consideration:

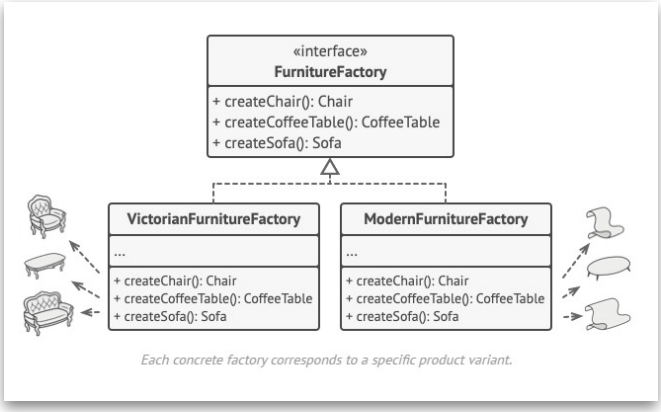
1. Scalability: easily create corresponding factories without modifying existing code
2. Maintenance
3. Code reusability
4. Powerful tool for managing complexity in software design

Real-Life Scenario:

Notification System: the notification mode can vary based on the user preferences or conditions

B. Abstract Factory:
organizes the creation of related or dependent object without specifying their concrete classes.

Example: The Abstract Factory pattern is used to create families of related products, such as Chair, Sofa, and CoffeeTable, in various styles like Modern, Victorian, and ArtDeco, ensuring that products match within a family. It defines interfaces for each product type, allowing different variants to implement these interfaces. An Abstract Factory interface includes creation methods for each product type, which concrete factory classes implement to produce specific variants, like ModernFurnitureFactory creating ModernChair, ModernSofa, and ModernCoffeeTable. Client code interacts with factories and products via their abstract interfaces, enabling the addition of new product families or variants without altering existing code. The specific factory type is typically chosen during application initialization based on configuration or environment settings.



Components:
1. Abstract factory: A high-level blueprint defining rules for creating families of related objects, methods for creating different types of related objects
2. Concrete factories: implementing the rules specified by the abstract factory, provides specific implementations for creating objects within a family
3. Abstract products: representing the family of related objects
4. Concrete products: actual instances of objects created by concrete factories
5. Client: Utilizing the abstract factory to create families of objects without knowing their concrete types

Real-Life Scenarios:
1. Developing a cross-platform GUI library: different platforms require different UI components
2. Handling various database systems: abstract factory can create database related objects(connections , queries) for different databases
3. Maintaining a consistent look and feel across diverse environments. It ensures consistent UI elements across platforms

Advantages:
1. Isolate client code from concrete classes
2. Eases exchanging object families
3. Promotes consistency among objects

Disadvantages:
1. Adds complexity due to multiple layers of abstraction
2. Requires defining new interfaces for each family of objects

Use when:
1. You need to create families of related objects but don't want it to depend on the concrete classes of those products - they might be unknown beforehand or you simply want to allow for future extensibility
2. Want to switch between different types easily while following the same rules
3. Consider implementing the abstract factory method when you have a class with a set of factory methods that blue its primary responsibility (In a well-designed program each class is responsible only for one thing. When a class deals with multiple product types, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown Abstract Factory implementation.)

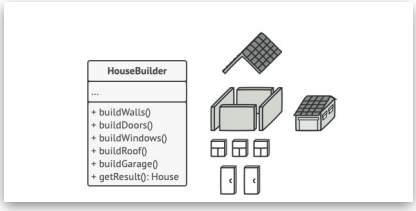
Difference between Factory method and Abstract Factory

Factory Method:
Purpose: Defines an interface for creating an object but lets subclasses decide which class to instantiate.
Responsibility: Delegates instantiation to subclasses, allowing them to choose the concrete implementation.
Example: Consider a class hierarchy for different types of vehicles (e.g., Car, Bike). Each subclass (e.g., CarFactory, BikeFactory) implements the factory method to create the specific vehicle1.
Use Case: Useful when you want to keep the instantiation logic separate from business logic and avoid code duplication.

Abstract Factory:
Purpose: Creates an entire family of related or dependent objects without specifying their concrete classes.
Responsibility: Provides an interface for creating families of objects (e.g., UI elements, furniture) that aren't in the same hierarchy.
Example: Creating different styles of UI elements (buttons, menus) for various platforms (Windows, macOS, Linux) using an abstract factory2.
Use Case: Ideal when you need to create related objects with consistent behavior across different families.

In summary
Factory Method focuses on creating a single object with subclass flexibility.
Abstract Factory deals with creating families of related objects without specifying their concrete classes. It's more versatile but adds complexity.

C. Builder:
Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an objet using the same construction code. It separates the construction of an object from its representation allowing the same construction process to create different presentation



Example: The Builder pattern is a creational design pattern that separates the construction of complex objects from their representation, allowing you to build different types and representations of an object using the same construction process. By extracting the object construction code into separate builder objects, you can construct complex objects step by step, with the flexibility to call only the necessary steps for a particular configuration. Different builder classes can implement the same construction steps (like buildWalls, buildDoor) differently to produce various representations of the product, such as a wooden cabin, a stone castle, or a palace made of gold and diamonds. Additionally, a director class can be introduced to define the order of the building steps, hiding the construction details from the client code and allowing reuse of the construction routines. The client interacts with the builder via a common interface and uses the director to execute the construction process, resulting in a well-constructed product tailored to specific requirements

when to use:
1. When you want your code to be able to create different representations of some product
2. To construct composite trees or other complex objects

Advantages:
1. Construct objects step by step, defer construction steps or run steps recursively
2. You can reuse the same constructions code when building various representations of products
3. Single responsibility principle. You can isolate complex construction code form the business logic of the product

Real-Life scenario:
In web development, the Builder pattern can be used in the creation of complex Response objects in a web server. For instance, consider a web server that needs to construct various types of HTTP responses based on the request it receives. Each response might need to have different headers, body content, and status codes.

Here's how the Builder pattern could be applied:

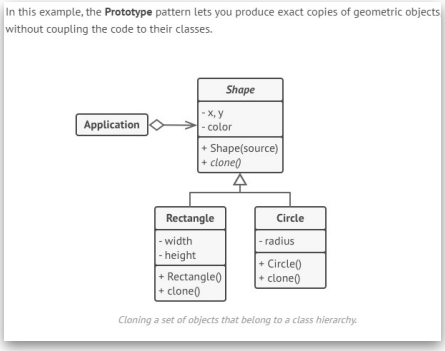
- A ResponseBuilder interface defines methods for adding headers, setting the status code, and building the body.
- Concrete classes like JsonResponseBuilder or HtmlResponseBuilder implement this interface to create different types of responses.
- A ResponseDirector takes a ResponseBuilder and constructs the response by calling the appropriate methods.

This allows for a flexible system where new types of responses can be added with minimal changes to the existing codebase, and the construction process is abstracted away from the main logic of the server

Components:
1. Builder: An interface that defines the steps to build a part of a complex object.
2. Concrete Builder: Implements the Builder interface and provides the implementation for those steps. There can be multiple Concrete Builders, each for different representations of the product.
3. Director: Responsible for managing the construction process using a Builder instance. It calls the building steps to construct the product.
4. Product: The complex object that is being built. The final output of the construction process.

D. Prototype:
Lets you copy existing object without making your code dependent on their classes. Instead of building a new object from scratch , you make a copy of an existing object that serves as a blueprint and can be customized as needed

Example:The Prototype pattern solves the problem of creating exact copies of objects by delegating the cloning process to the objects themselves. It defines a common interface for cloning, typically with a single clone method. This approach allows you to clone an object without needing to know its concrete class, avoiding the issues of external copying and class dependencies. The clone method creates a new instance of the object's class and copies all field values, including private ones, because objects of the same class can access each other's private fields. This pattern is especially useful when objects have numerous fields and complex configurations, as cloning a pre-configured prototype can be more efficient than constructing a new object from scratch. Pre-built prototypes, configured in various ways, can be cloned as needed, providing a flexible and reusable solution.

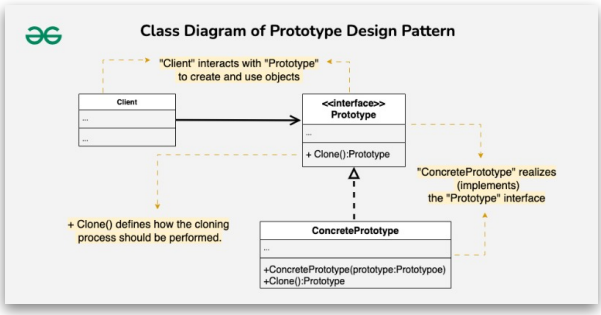


Real-Life scenario:
In web development, the Prototype Design Pattern can be particularly useful when dealing with the creation of complex user interface components. For instance, consider a web application that allows users to create custom dashboards. Each dashboard contains widgets like charts, tables, and forms that are configured by the user.
Instead of creating each widget from scratch every time a user wants to add one to their dashboard, the application could use prototypes of each widget type. When a user selects a widget to add, the application clones the prototype and then applies the user's custom configurations.

When to use:
1. When your code shouldn’t depend on the concrete classes of the object you need to copy
2. When you want to reduce the number of subclasses that only differ in the way they initialize their perspective objects

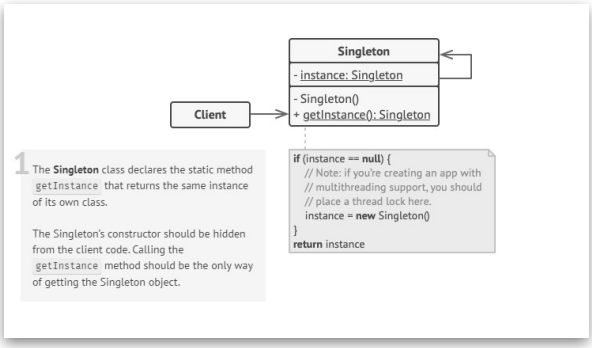
Advantages:
1. Clone objects without coupling to their concrete classes
2. Can get rid of repeated initialization code in favor of cloning pre-built prototypes
3. Can produce complex objects more conveniently
4. Get an alternative to inheritance when dealing with configuration presets for complex objects

Components:
1. prototype interface or abstract class
2. concrete prototypes and the client code
3. clone method specifying cloning behavior.



E. Singleton
Ensure that a class has only one instance, while providing global access to this instance. This is useful when exactly one object is needed to coordinate actions across the system

Example: The Singleton pattern ensures that a class has only one instance and provides a global access point to that instance, addressing two main problems. First, it controls access to shared resources like databases or files by ensuring only one instance of the class exists. This prevents multiple instances from being created and potentially causing conflicts or inconsistencies. Second, it provides a way to access this single instance globally while protecting it from being overwritten, unlike a global variable. The pattern achieves this by making the class constructor private to prevent direct instantiation and providing a static method that returns the single instance, creating it if it doesn't already exist. This static method ensures that every call returns the same object, maintaining a single point of access and control.



Code Perspective & Analysis: The Singleton pattern ensures that only one instance of the class is created, accessed by a static method (getInstance). The constructor is private to prevent direct instantiation.

Examples & Real-Life Scenario:
1. Database Connections: Singleton can manage a connection to a database, ensuring that only one connection is active at any given time.
2. Configuration Files: It can be used to read configuration settings. Once the settings are loaded, the same instance is used throughout the application.
3. Logging: A logging class can be implemented as a Singleton to ensure all logs are coordinated through a single instance.

Impact Analysis:
Singletons can be problematic if not handled carefully. They can:
1. Make unit testing difficult due to their global state.
2. Lead to resource contention in multi-threaded applications if not synchronized properly.
3. Create hidden dependencies, leading to tightly coupled system components.

When to use:
1. When a class in your program should have just a single instance available to all clients; for example a single database object shared by different parts of the program
2. When you need stricter control over global variables

Difference between static and singleton:
A static class is a class that cannot be instantiated and all its members are static, meaning they belong to the class itself rather than any object instance. It's used for grouping related static members, like utility methods, that can be accessed without creating an instance of the class.

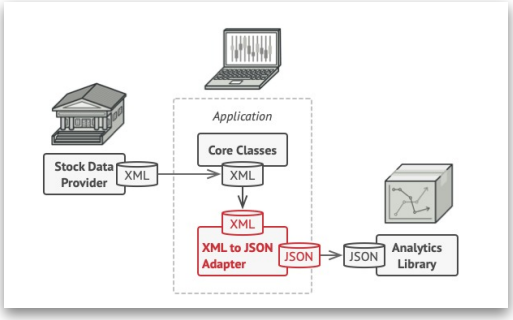
A singleton class, on the other hand, is a class that allows only one instance of itself to be created and provides a global point of access to that instance. It's not inherently static, but it controls object creation through a static method that returns the singleton instance.

Key Differences:
1. Instantiation: Static classes cannot be instantiated at all, while singleton classes can be instantiated once.
2. State: Static classes hold no state; they only have static members. Singleton classes can hold state within their single instance.
3. Inheritance: Static classes cannot be inherited since they cannot be instantiated. Singleton classes can be inherited but maintaining the singleton nature in the subclass can be complex.

2. Structural Design Patterns:
Concerned with how classes and objects are composed to from larger structures. They help ensure that when one part of a system changes, the entire structure does not need to do the same. Explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient

A. Adapter:
Allows object with incompatible interfaces to collaborate

Example:To integrate a third-party analytics library that only works with JSON into your stock market monitoring app, which downloads stock data in XML format, you can use the Adapter pattern. This pattern involves creating an adapter that converts XML data into JSON, allowing the app to communicate with the analytics library without modifying its source code or breaking existing functionality. The adapter acts as an intermediary, translating the XML data into the JSON format required by the library, ensuring seamless integration and enabling the app to leverage the analytics capabilities without any compatibility issues.



Real-Life example:
In software, a real-life scenario for the adapter pattern could be when you have classes that need to work with different types of databases. Each database (like MySQL, PostgreSQL, Oracle) may have different methods for connecting and querying data. An adapter can standardize these methods, allowing your application to interact with any database using a single interface1.

This pattern is also useful when integrating third-party libraries or APIs that have different interfaces from what your application expects.

When to use:
1. When you want to use some existing code, but its interface isn't compatible with the rest of the code
2. When you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass

Advantages:
1. You can separate the interface or data conversion code from the primary business logic of the program
2. Introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface

B. Bridge:
Lets you split a large class or a set of closely related classes into two separate hierarchies - abstraction and implementation - which can be developed independently of each other

Abstraction and implementation:
In the Bridge design pattern, *abstraction* refers to the high-level control layer that the client interacts with. It's an interface that defines operations that can be performed. The *implementation*, on the other hand, is the actual code that carries out these operations.

Abstraction Example:
In our previous JavaScript example, WebPage and its subclass AboutPage represent the abstraction layer. They define a structure and a set of functionalities (like getContent) that clients can use without needing to know about the theme details.

Implementation Example:
The Theme class and its subclasses DarkTheme and LightTheme are the implementation. They provide the underlying functionality that the abstraction calls upon, which in this case is just the color of the theme.

The "bridge" between them is typically represented by a reference or pointer in the abstraction to an object of the implementation. This allows the abstraction to delegate some of its responsibilities to the implementation, which can be changed dynamically at runtime.

This separation allows you to change the implementation without changing the abstraction and vice versa, leading to a more modular and maintainable codebase.

let's consider a remote control as an abstraction and the devices it controls (like a TV, DVD player, or streaming box) as the implementation.

Abstraction (Remote Control):

It provides an interface with buttons to perform operations like turn on/off, volume up/down, and change channel.
It doesn't know the specifics of how these operations are implemented in the TV or DVD player.
Implementation (Devices):

The TV or DVD player has the actual code to turn on/off, adjust volume, and change channels.
Each device has a different way of performing these operations.
The "bridge" is the signal sent from the remote control to the device. You can use the same remote control (abstraction) to operate different devices (implementations) because of this bridge.
If you get a new DVD player, you can program the remote to control it without needing a new remote.

Real-Life scenario:
Abstraction (Database Interface):

It defines a standard set of operations like connect, query, and disconnect.
The web application interacts with this interface, not directly with the database systems.
Implementation (Database Drivers):

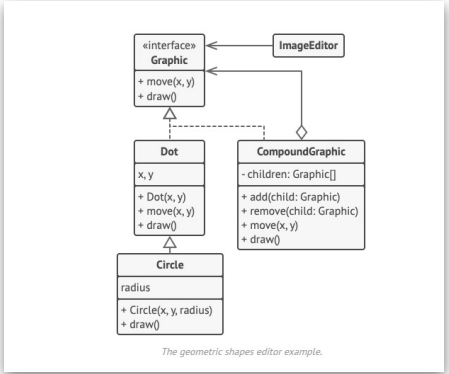
Each database system has its own driver (MySQLDriver, PostgreSQLDriver, MongoDBDriver) that implements the database interface.
These drivers handle the specifics of interacting with their respective databases.
The “bridge” here is the use of the database interface in the web application code. This allows you to switch between different databases without changing the high-level database interaction code in your web application.

In software design, a graphics application might need to draw shapes like circles and rectangles on different operating systems (Windows, Linux, macOS).

When to use:
1. You want to separate an abstraction from its implementation so that the two can vary independently.
2. You expect that the abstraction’s interface as well as its implementation may need to vary or extend in separate ways.
3. You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when they must be selected or switched at runtime.
4. You have a proliferation of classes resulting from a coupled interface and numerous implementations.
5. You need to share an implementation among multiple objects, and this fact should be hidden from the client.
In summary, use the Bridge pattern when you need to decouple an interface and its implementation so that they can be developed, extended, and operated independently.

C. Composite:
Compose objects into tree structures and then work with these structures as if they were individual objects

Example:The CompoundGraphic class is a container that can comprise any number of sub-shapes, including other compound shapes. A compound shape has the same methods as a simple shape. However, instead of doing something on its own, a compound shape passes the request recursively to all its children and “sums up” the result.



The geometric shapes editor example.

The client code works with all shapes through the single interface common to all shape classes. Thus, the client doesn’t know whether it’s working with a simple shape or a compound one. The client can work with very complex object structures without being coupled to concrete classes that form that structure.

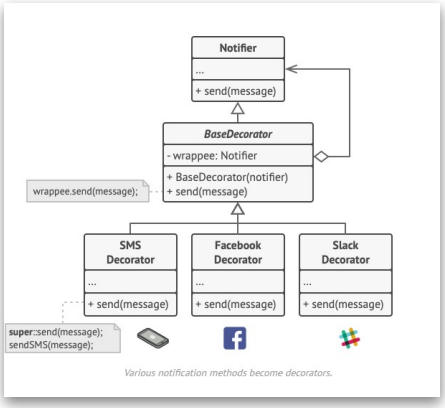
When to use:
1. You have to implement. Tree-like object structure
2. You want client code to treat both simple and complex elements uniformly

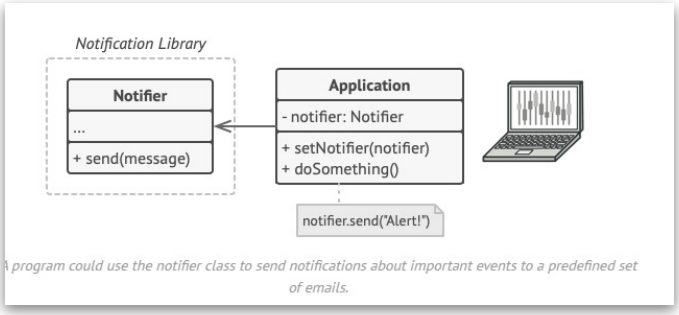
Advantages:
1. Can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage
2. Introduce new element types into the app without breaking the existing code, which now works with the object tree

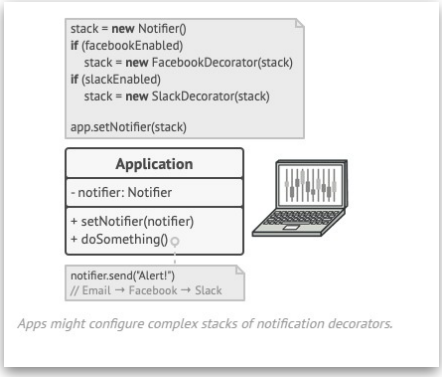
Real-Life scenario:
1. Building a customer's portfolio in banking applications.
2. Creating complex GUIs where individual elements and groups of elements are treated similarly.
3. Representing file systems with files and directories.
4. Simplify manipulation of complex UI components and their individual elements

D. Decorator:
Lets you attach new behaviors to object by placing these objects inside special wrapper objects that contain the behaviors

Example: Initially, the notification library allowed sending email notifications via a Notifier class with a single send method. As users demanded additional notification types like SMS, Facebook, and Slack, subclasses were created for each type. However, the need to send multiple notifications simultaneously led to a combinatorial explosion of subclasses. To resolve this, the Decorator pattern was employed. Instead of extending classes, behaviors were added dynamically using composition. Each notification type was turned into a decorator, wrapping the base Notifier class and allowing multiple decorators to be stacked. This way, the client code could configure a chain of decorators, enhancing the base notifier with additional behaviors while maintaining a clean, manageable structure.







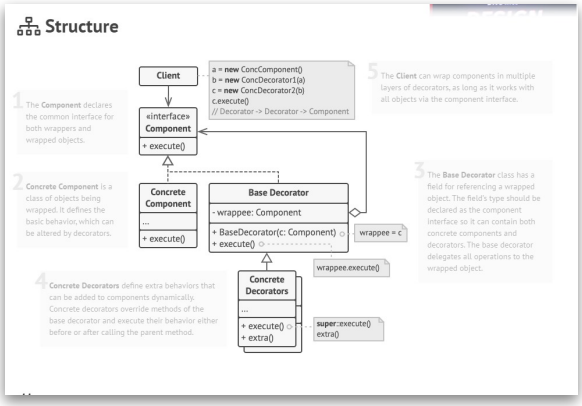
When to use:
1. Need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects
2. Not possible to extend an object's behavior using inheritance. Many programming languages have the final keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

Impact: The Decorator pattern is impactful when you need to add responsibilities to objects at runtime without affecting other objects. It provides a flexible alternative to subclassing for extending functionality

Real-Life scenario:
In web development, a common scenario is enhancing UI components. For instance, you might have a basic TextInput component, and you can use decorators to add features like spell-checking, auto-completion, or styling without modifying the TextInput itself.

Components:
1. Component: An interface or abstract class defining the methods that will be implemented.
2. Concrete Component: A class implementing the Component interface.

3. Decorator: An abstract class that maintains a reference to a Component object and conforms to the Component interface.
4. Concrete Decorator: A class extending the Decorator class, providing additional behavior.



Advantages:
1. Extend an objects behavior without making a new subclass
2. Add or remove responsibilities from an object at runtime
3. Combine several behaviors by wrapping an object into multiple decorators
4. Can divide monolithic class thar implements many possible variants of behavior into several smaller classes

E. Facade:
Provides a simplified interface to a complex system of classes, library, or framework. It hides the complexities of the system and provides an easier way to access its functionalities.

Example: computer might have several operations like startCPU(), loadRAM(), readDisk(), etc. A ComputerFacade class could provide a simple method startComputer() that encapsulates all these operations, making it easier for the user to operate the computer without knowing the underlying complexities.

When to use:
1. When you need to have a limited but straight forward interface to a complex subsystem
2. When you want to structure a subsystems into layers

Real-life example:
1. Simplifying API Usage: A facade can provide a simple method to interact with a complex set of APIs. For example, instead of making multiple API calls to different services, a facade can offer a single method that encapsulates all these calls, streamlining the process for the developer.
2. Database Operations: When saving a user to a database, instead of writing multiple lines of code to handle the connection, query preparation, execution, and error handling, a facade can offer a simple saveUser method that hides all the complexity.

F. Flyweight:
aims to minimize memory usage and computational expenses by sharing as much data as possible with similar objects. It's particularly useful when you need to create a large number of similar objects and want to avoid the overhead of each object's memory and processing footprint. Objects created using the flyweight pattern typically share the same underlying data, making them immutable to facilitate sharing. This approach helps in supporting large numbers of fine-grained objects efficiently
In the Flyweight pattern, flyweight objects must be immutable, initialized only once via constructor parameters, and should not expose setters or public fields. To manage these objects efficiently, a factory method can be used to maintain a pool of flyweights. This method checks if an existing flyweight with the desired intrinsic state exists; if found, it returns that flyweight, otherwise, it creates a new one and adds it to the pool. This factory method can be placed in a flyweight container, a new factory class, or made static within the flyweight class itself.

When to use:
1. When you program must support a huge number of objects which barely fit into available RAM

Real-life scenario:

1. Text Editors: In text editors, the flyweight pattern can be used to manage the formatting of characters. Instead of creating a new object for each character with its formatting, a shared object can be used to represent characters with similar formatting, reducing memory usage.
2. Web Browsers: Modern web browsers use the flyweight pattern to prevent loading the same images twice. When a browser loads a web page, it checks if an image is already in its cache before fetching it from the internet, thus saving bandwidth and speeding up page rendering.
3. String Interning in Java: The java.lang.String class in Java uses the flyweight pattern to reuse instances of strings that are identical, thereby saving memory.

G. Proxy:
The Proxy Design Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. This pattern is particularly useful when you want to add an extra layer of control over the access to an object. The proxy acts as an intermediary and can perform additional tasks such as:
_Controlling the creation and deletion of the object
_Managing access rights
_Providing a simplified interface to a complex system
_Adding security checks before accessing the real object
In essence, the proxy pattern allows you to perform something either before or after the request gets through to the original object, which can help in scenarios where direct access to the object is not ideal or possible

When to use:
1. Virtual proxy: when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time. You can delay the object's initialization to a time when it's really needed
2. Protection proxy: when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications, the proxy can pass the request to the service only if the client's credentials match some criteria
3. Remote proxy: when the service object is located on a remote server; the proxy passes the client request over the network, handling all of the nasty details of working with the network
4. Logging proxy: when you want to keep a history of requests to the service object, the proxy can log each request before passing it to the service
5. Caching proxy: This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.
6. Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it. The proxy can keep track of clients that obtained a reference to the service object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources.

Advantages:
1. Control the service object without the clients knowing about it
2. Manage lifecycle of the service object when clients don't care about it
3. Works even if the service object isn't ready or is not available
4. Introduce new proxies without changing the service or clients

_Virtual Proxy: Acts as a stand-in for expensive-to-create objects and may create the real object only when its functionality is explicitly required.
Use case: Lazy loading of a high-resolution image.
Example: A VirtualImage class that only loads the actual image when it's needed to be displayed.

_Protection Proxy: Controls access to the real object, protecting it from unauthorized access by implementing appropriate permissions.
Use case: Access control for a sensitive object.
Example: A ProtectedFile class that only allows users with the correct password to open the file.

_Remote Proxy: Provides a local representation for an object that resides in a different address space or on a remote server. It's used in distributed systems to hide the complexity of remote calls.
Use case: Interacting with an object in a different network location.
Example: A RemoteDatabase class that represents a database on a server, handling the complexity of network communication.
_Logging Proxy: Adds logging functionality when accessing an object. It can keep track of the number of times an object's method has been called, for example.
Use case: Tracking the number of times an operation is performed.
Example: A LoggingCalculator class that logs every calculation performed by a calculator object

_Caching Proxy is used to improve performance by holding the results of expensive operations and reusing them when the same inputs occur again. It can be particularly useful in web development for caching web pages, API calls, or database queries.
For example, in a web application, a caching proxy could store the results of a database query. When the application requests the same query again, instead of hitting the database, it would retrieve the result from the cache, saving time and resources.

3. Behavioral Design Pattern
Concerned with algorithms and the assignment of responsibilities between objects, they focus on the patterns of communication between classes or objects

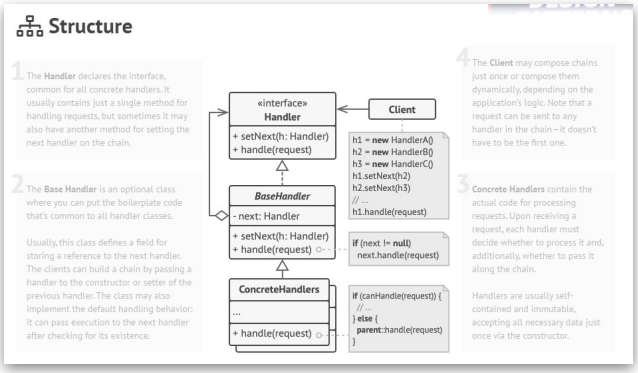
A. Chain of responsibility:
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain

Example:
payment processing systems where a purchase request can go through various handlers to process different types of payments (credit card, PayPal, etc.). Each handler checks if it can process the request and either handles it or passes it to the next handler in the chain.
This pattern is also used in command objects for operations like undo/redo where commands are passed through a series of processors that can either execute, reverse, or pass the command along.

Real-Life scenario:
In web development, the Chain of Responsibility pattern can be seen in event handling. When an event occurs, it can be handled by the element that triggered it or be passed up the DOM tree until an element with a corresponding event handler is found.
In software design, it's often used in logging systems where a log message could be passed through multiple handlers, each responsible for a different logging action (e.g., writing to a file, sending an email, etc.).

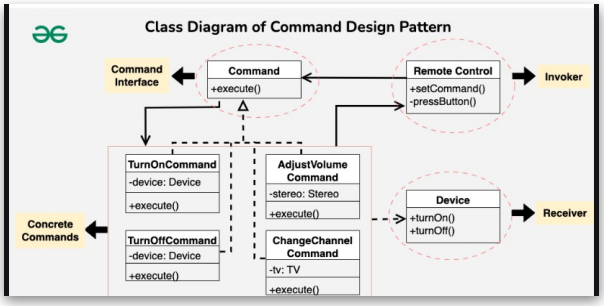
When to use:
1. When your program is expected to process different kind of requests in various ways. It lets you link several handlers into one chain and upon receiving a request, "ask" each handler whether it can process it
2. When it's essential to execute several handlers in a particular order

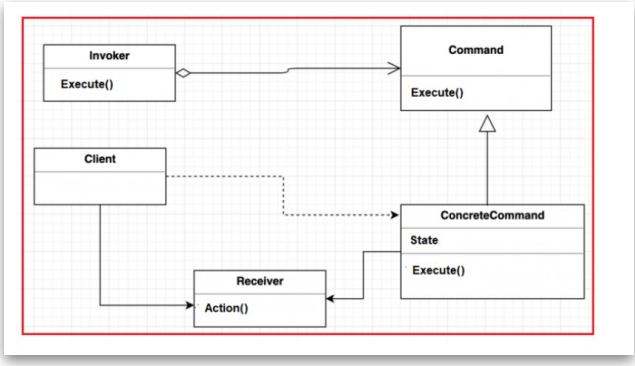
Advantages:
1. You can control the order of request handling
2. Decouple classes that invoke operations from classes that perform operations



B. Command:
Turns request into stand-alone object allowing parameterization of clients with different requests, queuing of requests, and support for undoable operations (Action thar can be reversed or undone in a system)

Components:
1. Command: This is an interface for executing an operation.
2. ConcreteCommand: This class extends the Command interface, implementing the Execute method by invoking the corresponding operation(s) on the Receiver object.
4. Receiver: This class knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.
5. Invoker: This class asks the command to carry out the request.
6. Client: The client creates a ConcreteCommand object and sets its receiver.





Receiver: This class contains the actual implementation of the method the client wants to call. Our example is the Document class's Open, Save, and Close method.
Command: This will be an interface specifying the Execute operation. In our example, the ICommand interface has only one method, i.e., Execute.
ConcreteCommand: These classes will implement the ICommand interface and provide implementations for the Execute operation. As part of the Execute method, it will invoke operation(s) on the Receiver object. Our example is the OpenCommand, SaveCommand, and CloseCommand classes.
Invoker: The Invoker will be a class and ask the command to carry out the action. In our example, it is the MenuOptions class.
Client: This is the class that creates and executes the command object. In our example, it is the Main method of the Program class.

C. Iterator:
Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree…)

D. Mediator:
Reduce complex communication between classes or objects to a centralized point. Restrict communication between objects and forces them to collaborate only via a mediator object.

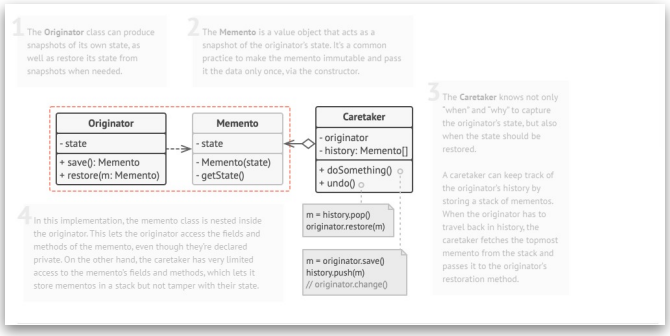
Real-Life example:
In web development, a real-life example of the Mediator pattern is an event management system where components communicate through a central event manager rather than directly with each other. This can be seen in frameworks like React, where components update their state based on events without needing to know about the inner workings of other components.

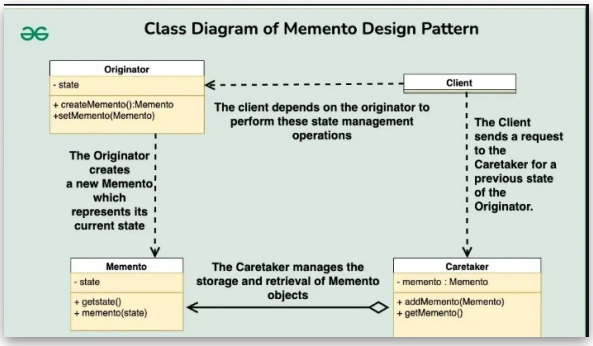
In software design, an IDE (Integrated Development Environment) can serve as a mediator between plugins and the core software. The IDE provides a central interface for plugins to extend functionality without needing to modify the core system directly.

E. Memento:
Lets you save and restore the previous state of an object without revealing the details of its implementation

Real-Life scenario:
text editor with an undo feature. In this case, each time the user makes changes to the text, such as typing, deleting, or formatting, the editor's state (the content and formatting of the document) is saved as a memento. If the user decides to undo their last action, the editor can restore its state from the most recent memento. This allows users to revert their documents to previous states without needing to know how the states are represented internally in the editor.

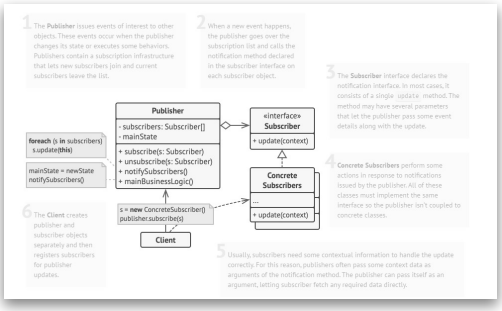
When to use:
1. when you want to produce snapshots of the object's state to be able to restore a previous state of the object, it let you make full copies of an objects state including private fields and store them separately from the object
2. When direct access to the object fields/getters/setters violated its encapsulation





When it gets impacted: The Memento pattern is particularly useful when you need to maintain historical states of an object that can be restored later. It's impacted when there are requirements for reversible actions or snapshots of object states.

F. Observer:
Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing



Real-Life scenario:

In web development, the Observer pattern is often used in the following scenarios:
_Model-View-Controller (MVC) frameworks: The model is the subject, and views are observers. When the model changes (like a database update), it notifies the views to update the UI accordingly.
_Event listeners: JavaScript heavily relies on event listeners, which are a form of the Observer pattern. Elements on a webpage (subjects) notify event handler functions (observers) when an event occurs, like a button click.
_WebSocket connections: In real-time applications, WebSocket servers (subjects) send notifications to connected clients (observers) when new data is available.

In software development more broadly, it's used for:
_Notification systems: Any application with a notification feature likely uses an observer pattern to update users about new events or messages.

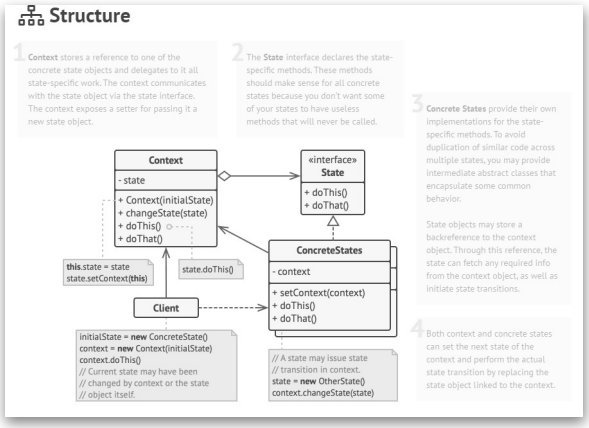
_Data binding: Frameworks that support data binding use observers to sync the UI with underlying data models automatically.

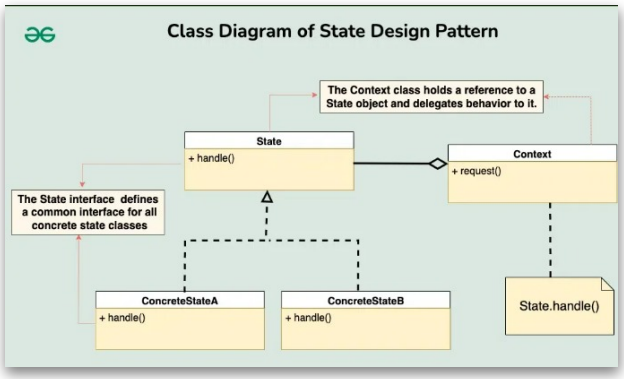
The Observer pattern helps keep different parts of a system isolated, making them easier to maintain and extend. It's particularly impactful in systems where changes to one component can result in changes to others, and you want to manage these dependencies flexibly.

G. State:
Lets an object alter its behavior when its internal state change. It appears as if the object changes its class

Example: represents a light switch with two possible states: on and off. The LightSwitch class contains a reference to a State interface, which is implemented by two concrete state classes: OnState and OffState. Each state class has a handleRequest method that performs an action (turning the light on or off) and then switches the LightSwitch's current state to the opposite state. When the pressSwitch method of the LightSwitch is called, it delegates the action to the current state's handleRequest method, resulting in toggling the light and changing the switch's state.

When to use:
1. Multiple states with distinct behaviors: If your object exists in several states (e.g., On/Off, Open/Closed, Started/Stopped), and each state dictates unique behaviors, the State pattern can encapsulate this logic effectively.
2. Complex conditional logic: When conditional statements (if-else or switch-case) become extensive and complex within your object, the State pattern helps organize and separate state-specific behavior into individual classes, enhancing readability and maintainability.
3. Frequent state changes: If your object transitions between states frequently, the State pattern provides a clear mechanism for managing these transitions and their associated actions.
4. Adding new states easily: If you anticipate adding new states in the future, the State pattern facilitates this by allowing you to create new state classes without affecting existing ones.

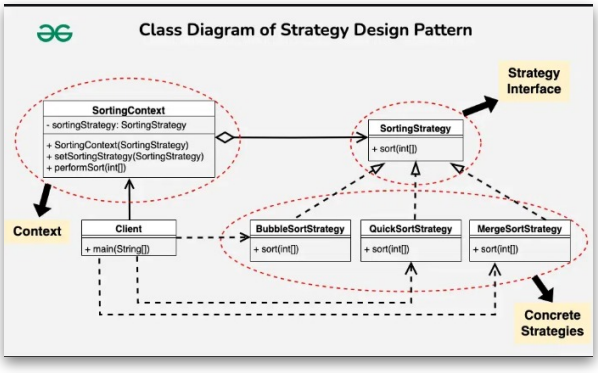




H. Strategy:
Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable

Example:
Here are some scenarios where you might use the Strategy pattern:

Sorting: If you have different sorting algorithms and want to choose one based on the size or type of the dataset.
Compression: When you need to support different compression algorithms (like ZIP, RAR) and want to allow users to choose.
Navigation: In a navigation app, you might have

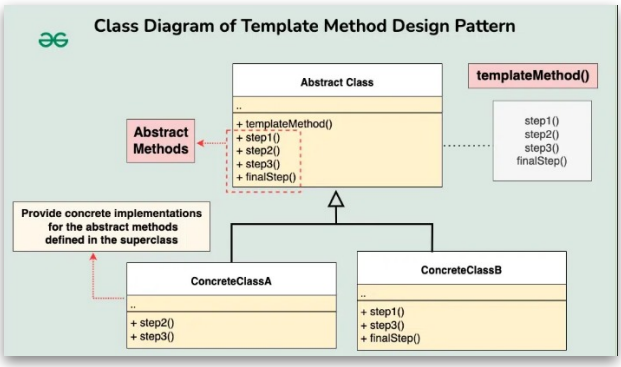


different routing strategies (fastest, shortest, avoiding tolls) that the user can select from.
In web development, it could be used for form validation where different strategies are applied based on the form's context or in e-commerce platforms to apply different discount strategies during checkout.

In software development, it's impactful for maintaining and extending algorithms without changing the clients that use them, providing flexibility and reusability.

When to use:
Here are some situations where you should consider using the Strategy pattern:
1. Multiple Algorithms:
When you have multiple algorithms that can be used interchangeably based on different contexts, such as sorting algorithms (bubble sort, merge sort, quick sort), searching algorithms, compression algorithms, etc.
2. Encapsulating Algorithms:
When you want to encapsulate the implementation details of algorithms separately from the context that uses them, allowing for easier maintenance, testing, and modification of algorithms without affecting the client code.
3. Runtime Selection:
When you need to dynamically select and switch between different algorithms at runtime based on user preferences, configuration settings, or system states.
4. Reducing Conditional Statements:
When you have a class with multiple conditional statements that choose between different behaviors, using the Strategy pattern helps in eliminating the need for conditional statements and making the code more modular and maintainable.
5. Testing and Extensibility:
When you want to facilitate easier unit testing by enabling the substitution of algorithms with mock objects or stubs. Additionally, the Strategy pattern makes it easier to extend the system with new algorithms without modifying existing code.

I. Template method:
Defines the skeleton of an algorithm in a superclass but lets subclasses override specific steps of the algorithm without changing its structure



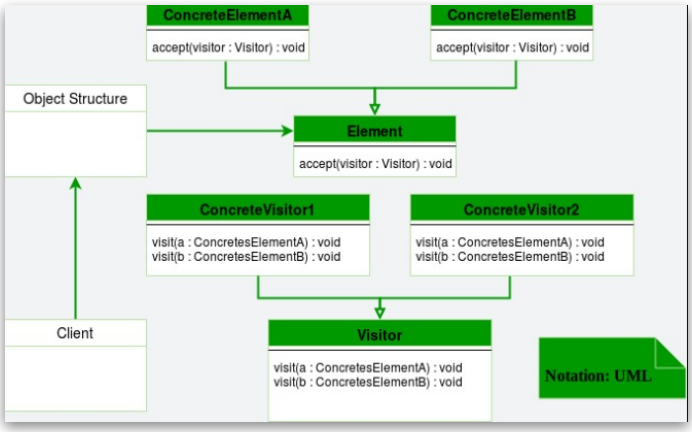
Example:
A practical use of the Template Method pattern is in the development of an application framework for data processing. The framework could define a generic structure for processing data with steps such as data loading, data transformation, and data storage. Each step is represented by a method in an abstract class, and the sequence of these steps is fixed in a template method. Developers using the framework can then create subclasses for specific data sources or processing requirements, overriding the necessary methods to customize the behavior of each step without altering the overall processing sequence. This allows for flexibility and reuse of the common processing algorithm while enabling customization for different types of data or processing logic.

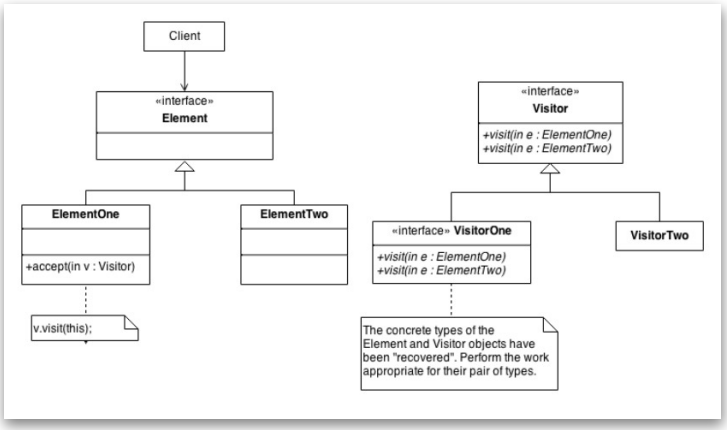
Real-Life scenario:
In web development, a real-life scenario for the Template Method pattern could be a web page generation framework where the overall structure of a page is defined in a base class with methods for rendering the header, content, and footer. Subclasses can override these methods to provide specific content or styling for different types of pages, such as product pages, articles, or user profiles, while maintaining a consistent layout across the site.

In software design, the Template Method pattern might be used in an application that requires different types of reports. The base class could define the structure of a report and provide methods for generating common sections like title, table of contents, and summary. Subclasses would implement the details of each section based on the type of report, such as financial, sales, or inventory reports, ensuring that all reports follow a consistent format while allowing for specific content in each type.

When to use:
1. When you want to let clients extend only particular steps of an algorithm but not the whole algorithm or structure
2. When you have several classes that contain almost identical algorithms with some minor differences

J. Visitor:
Used when we have to perform an operation on a group of similar kind of Objects.





Example:
Consider a shape hierarchy with shapes like square and circle. Imagine you want to implement functionality to calculate the area for all shapes. Traditionally, you'd define an area() method in each shape class. The Visitor pattern offers another approach. Here, a separate "AreaCalculator" visitor class would implement a visitSquare() method to calculate the square's area and a visitCircle() method for the circle's area. The shapes would have an accept() method that takes the visitor and calls the appropriate visit method based on the shape's type. This isolates area calculation logic and makes adding new shapes (with a corresponding visit method in the visitor) easier without modifying the existing shape classes.

Real-Life scenario:
In web development, you can use the Visitor pattern to perform operations like rendering, validation, or serialization on different types of elements in a web page without coupling the operations to the element classes. For example, if you have a form with various input elements, you can create a ValidationVisitor that checks each element for validity. Each input element class would have an accept method that allows the ValidationVisitor to perform the appropriate validation logic.

In software development, the Visitor pattern is useful for operations that need to be performed across a diverse set of objects within an application. For instance, if you have a graphics editor with shapes like circles, rectangles, and lines, you could use a ExportVisitor to export the shapes into different file formats (SVG, PNG, etc.) without adding export logic to the shape classes themselves.

This pattern helps keep your codebase flexible and maintainable by separating concerns and allowing you to add new operations without modifying existing classes.

References:

https://www.tutorialspoint.com/design_pattern/index.htm
https://refactoring.guru/design-patterns
https://dotnettutorials.net/lesson/design-patterns-online-training/
https://www.dofactory.com/net/design-patterns
https://www.geeksforgeeks.org/software-design-patterns/?ref=lbp