# Name

Intent: Brief synopsis (as on previous slides).

Motivation: The context of the problem.

Applicability: Circumstances under which the pattern applies.

Structure: Class diagram of solution.

Participants: Explanation of the classes/objects and their roles.

Collaborations: Explanation of how the classes / objects cooperate.

Consequences: Discussion of impact, benefits, & liabilities.

Implementation: Discussion of techniques, traps, language dependent issues....

## Sample Code

Known Uses: Well-known systems already using the pattern.

Related patterns: Sequentially access the elements of a collection without exposing implementation.

…

Name: Factory

Intent: provides an interface for creating objects but allows subclasses to decide which class to instantiate. It encapsulates the object creation logic, providing a flexible way to create objects without tightly coupling the client code to the specific classes being instantiated.

Motivation: addresses the problem of creating objects without directly coupling the client code to their concrete implementations. It promotes loose coupling and enhances the extensibility of the codebase. By centralizing object creation within a factory, it becomes easier to manage and modify the creation process, supporting the Open-Closed Principle.

Applicability: The Factory pattern is applicable when:

1. The client code needs to create objects without knowing their specific classes.
2. The creation process may vary or be extended with additional subclasses.
3. The client code should not depend on concrete classes but instead rely on abstractions.

Participants:

* Creator: Abstract class or interface defining the factory method(s) for creating objects.
* ConcreteCreator: Subclasses of the Creator class that implement the factory method(s) to create specific objects.
* Product: Abstract class or interface defining the common interface for the objects that the factory creates.
* ConcreteProduct: Concrete classes implementing the Product interface.

Collaborations: The client code interacts with the Creator class to create objects through the factory method. The Creator class delegates the object creation to the ConcreteCreator subclass, which, in turn, creates the specific ConcreteProduct object.

Consequences:

* The Factory pattern promotes loose coupling between the client code and the concrete classes, as the client only relies on the abstractions provided by the Product and Creator interfaces.
* It enhances code extensibility, allowing for the addition of new ConcreteProduct subclasses without modifying existing client code.
* The pattern centralizes object creation logic, making it easier to manage and modify the creation process.
* However, introducing the Factory pattern can add complexity by requiring the creation of additional classes.

Implementation:

* The Factory pattern can be implemented using abstract classes or interfaces to define the Creator and Product abstractions.
* The factory method in the Creator class should be declared as abstract to be implemented by the ConcreteCreator subclasses.
* Language-dependent issues may arise, such as the need to choose between using abstract classes or interfaces based on the programming language's features and requirements.

Known Uses: java.util.Calendar class uses the Factory pattern to create instances of different calendar systems.

Related patterns: The Factory pattern is related to other creational patterns, such as the Abstract Factory pattern, which provides an interface for creating families of related or dependent objects, and the Singleton pattern, which ensures that only one instance of a class is created. Additionally, the Factory pattern can be combined with other patterns, such as the Strategy pattern, to vary the behavior of created objects dynamically.

Pattern Name: Abstract Factory

Intent: provides an interface for creating families of related or dependent objects without specifying their concrete classes. It encapsulates the creation of objects belonging to different but related product families, ensuring their compatibility and promoting loose coupling between client code and the specific implementations.

Motivation: The Abstract Factory pattern addresses the problem of creating families of objects that are designed to work together. It allows the client code to work with these objects without being tied to their concrete classes. This pattern supports the Open-Closed Principle by allowing easy addition of new product families and ensuring that the client code remains unchanged.

Applicability: The Abstract Factory pattern is applicable when:

1. The client code needs to create multiple families of related or dependent objects.
2. The client code should not depend on specific concrete classes but instead rely on abstractions.
3. The system should be easily extended to support new product families in the future.

Participants:

* AbstractFactory: Abstract class or interface defining the interface for creating the product objects. It declares a set of factory methods, each responsible for creating a different product.
* ConcreteFactory: Subclasses of the AbstractFactory that implement the factory methods to create specific product objects.
* AbstractProduct: Abstract class or interface defining the interface for the product objects created by the factory.
* ConcreteProduct: Concrete classes implementing the AbstractProduct interface.

Collaborations: The client code interacts with the AbstractFactory to create the desired product objects through the factory methods. The AbstractFactory delegates the creation of the product objects to the ConcreteFactory subclass, which creates the specific ConcreteProduct objects belonging to the corresponding product family.

Consequences:

* The Abstract Factory pattern promotes loose coupling between the client code and the concrete classes by relying on abstractions.
* It ensures that the created product objects are compatible within their respective families.
* The pattern supports the addition of new product families without modifying existing client code.
* However, introducing the Abstract Factory pattern can increase the complexity of the codebase due to the creation of additional classes.

Implementation:

* The Abstract Factory pattern can be implemented using abstract classes or interfaces to define the AbstractFactory and AbstractProduct abstractions.
* The ConcreteFactory subclasses implement the factory methods to create the specific ConcreteProduct objects.
* Language-dependent issues may arise, such as the need to choose between using abstract classes or interfaces based on the programming language's features and requirements.

Known Uses: The Abstract Factory pattern is commonly used in software systems where families of related objects need to be created, such as graphical user interface toolkits. For example, in the Java Swing framework, the javax.swing.UIManager class uses the Abstract Factory pattern to create platform-specific Look and Feel objects.

Related patterns: The Abstract Factory pattern is related to other creational patterns, such as the Factory Method pattern, which focuses on creating individual objects, and the Singleton pattern, which ensures that only one instance of a class is created. The Abstract Factory pattern can also be combined with other patterns, such as the Bridge pattern, to decouple the product hierarchy from the concrete implementations.

Pattern Name: Bridge

Intent: The Bridge design pattern decouples an abstraction from its implementation, allowing them to vary independently. It provides a way to separate the interface or abstract class from its concrete implementation, enabling them to evolve independently.

Motivation: The Bridge pattern addresses the problem of an unmanageable and rigid class hierarchy when both the abstraction and its implementation have multiple variations. It promotes loose coupling and enhances flexibility, as changes in one part of the hierarchy won't affect the other. It also supports the Open-Closed Principle by allowing easy extension without modification of existing code.

Applicability: The Bridge pattern is applicable when:

1. You want to avoid a permanent binding between an abstraction and its implementation.
2. Both the abstraction and its implementation need to be extended independently.
3. Changes in the implementation of an abstraction should not affect clients using the abstraction.
4. You have a complex hierarchy that can be simplified by separating the abstraction and implementation.

Participants:

* Abstraction: The high-level interface or abstract class that defines the abstraction's behavior. It maintains a reference to the Implementor object.
* RefinedAbstraction: A subclass of Abstraction that extends or further customizes the behavior defined by the abstraction.
* Implementor: The interface or abstract class that defines the operations that can be performed on the implementation objects.
* ConcreteImplementor: Concrete subclasses of Implementor that provide the actual implementation of the operations.

Collaborations: The Abstraction delegates the implementation-specific operations to the Implementor object. The Implementor object defines the specific implementation for these operations, allowing the Abstraction and Implementor to vary independently.

Consequences:

* The Bridge pattern decouples the abstraction from its implementation, promoting loose coupling and flexibility.
* It allows for independent extensibility of both the abstraction and its implementation.
* Changes in the abstraction or implementation do not affect each other or the client code.
* However, introducing the Bridge pattern can lead to an increased number of classes and complexity, especially for simple hierarchies.

Implementation:

* The Bridge pattern can be implemented by defining abstract classes or interfaces for the Abstraction and Implementor.
* The Abstraction class should maintain a reference to the Implementor object and delegate implementation-specific operations to it.
* Language-dependent issues may arise, such as the need to choose between using abstract classes or interfaces based on the programming language's features and requirements.

Known Uses: The Bridge pattern is commonly used in frameworks and libraries where multiple variations of an abstraction and its implementation exist. For example, in graphical user interface toolkits, the Bridge pattern is often employed to separate the high-level windowing functionality from the platform-specific windowing system.

Related patterns: The Bridge pattern is related to other structural patterns, such as the Adapter pattern, which focuses on making unrelated classes work together, and the Composite pattern, which deals with hierarchies of objects. Additionally, the Bridge pattern can be combined with the Abstract Factory pattern to provide a family of related abstractions and implementations.

Pattern Name: Builder

Intent: separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It provides a step-by-step approach to create objects, allowing control over the object's construction and enabling the creation of different configurations.

Motivation: The Builder pattern addresses the problem of creating complex objects with multiple parts or attributes. It provides a way to construct these objects in a controlled manner, allowing the client code to specify the desired configuration without exposing the construction process. It promotes code clarity, reusability, and flexibility in creating complex objects.

Applicability: The Builder pattern is applicable when:

1. The object being created has multiple parts or attributes.
2. The construction process involves multiple steps or variations.
3. The client code needs control over the construction process to create different configurations.
4. The construction process should be isolated from the client code.

Participants:

* Builder: Abstract class or interface defining the construction steps for creating the complex object.
* ConcreteBuilder: Concrete subclasses of the Builder that implement the construction steps to build the object. Each ConcreteBuilder may create a different representation or configuration of the object.
* Director: Controls the construction process by interacting with the Builder to execute the construction steps in a specific order.
* Product: The complex object being constructed. It represents the final result of the construction process.

Collaborations: The client code interacts with the Director to initiate the construction process. The Director uses the Builder to execute the construction steps in a specific order. The Builder performs the actual construction of the object, step by step, according to the configuration specified by the client. Once the construction is complete, the client can obtain the final Product from the Builder.

Consequences:

* The Builder pattern separates the construction logic from the client code, providing control over the construction process and creating different configurations of the object.
* It improves code readability and maintainability by encapsulating complex construction steps within the Builder and Director classes.
* The pattern supports the creation of immutable objects by constructing them in a step-by-step manner.
* However, implementing the Builder pattern can lead to the creation of additional classes and increased complexity, especially for simple objects that don't require extensive configuration.

Implementation:

* The Builder pattern can be implemented using abstract classes or interfaces to define the Builder and Product abstractions.
* ConcreteBuilder subclasses implement the construction steps in a specific order to create different representations or configurations of the Product.
* The Director class controls the construction process by interacting with the Builder and orchestrating the execution of construction steps.
* Language-dependent issues may arise, such as the need to handle concurrent access to the Builder or synchronization of the construction process.

Known Uses: commonly used in scenarios where the creation of complex objects with multiple parts is required. In the Java programming language, the StringBuilder class uses the Builder pattern to construct strings in a step-by-step manner.

Related patterns: Abstract Factory pattern, which provides an interface for creating families of related objects, and the Prototype pattern, which focuses on creating objects by cloning existing instances. The Builder pattern can also be combined with the Composite pattern to construct complex composite objects.

Pattern Name: Composite

Intent: allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly by using a common interface. The pattern enables the clients to work with complex structures in a simple and consistent manner.

Motivation: The Composite pattern addresses the need to represent part-whole hierarchies, where individual objects and groups of objects are treated uniformly. It allows the clients to interact with both leaf nodes and composite nodes without needing to distinguish between them. The pattern promotes code simplicity, flexibility, and ease of adding new components to the structure.

Applicability: The Composite pattern is applicable when:

1. You need to represent part-whole hierarchies of objects.
2. You want clients to treat individual objects and compositions uniformly.
3. You want to add new types of components to the hierarchy without affecting the client code.
4. You need to perform operations recursively on the objects in the structure.

Participants:

* Component: The common interface or abstract class that defines the behavior for all objects in the composition. It declares operations that are applicable to both leaf and composite objects.
* Leaf: The individual objects that do not have any child components. They implement the operations defined by the Component interface.
* Composite: The composite objects that can have child components. They implement the operations defined by the Component interface but also maintain a collection of child components.
* Client: The client code that interacts with objects through the Component interface, treating individual objects and compositions uniformly.

Collaborations: The client code interacts with objects through the Component interface, calling the operations defined by the interface. For composite objects, the operations are recursively delegated to child components, allowing the client to traverse and manipulate the entire hierarchy.

Consequences:

* The Composite pattern simplifies the client code by allowing it to treat individual objects and compositions uniformly.
* It provides flexibility in adding new types of components to the hierarchy without modifying existing client code.
* The pattern enables recursive operations on the objects in the structure, allowing complex tree traversal and manipulation.
* However, introducing the Composite pattern can make the code more complex, especially if there is a wide variation in the behavior and structure of the components.

Implementation:

* The Composite pattern can be implemented using abstract classes or interfaces to define the Component and Leaf abstractions.
* The Composite class maintains a collection of child components and implements the Component operations by delegating them to the child components.
* The Leaf class represents individual objects and implements the Component operations directly.
* Language-dependent issues may arise, such as the need to handle type safety when working with collections of different component types.

Known Uses: commonly used in user interface frameworks, file systems, and graphics modeling systems. For example, in the Java Swing framework, the Swing component hierarchy uses the Composite pattern to represent the structure of graphical user interface elements.

Related patterns: related to other structural patterns, such as the Decorator pattern, which allows additional behavior to be added to objects dynamically, and the Iterator pattern, which provides a way to traverse elements of an aggregate object without exposing its underlying structure. The Composite pattern can also be combined with the Iterator pattern to traverse the components of a composite structure.

Pattern Name: Decorator

Intent: The Decorator design pattern allows behavior to be added to an individual object dynamically, without affecting the behavior of other objects in the same class. It provides a flexible alternative to subclassing for extending functionality.

Motivation: The Decorator pattern addresses the need to add additional responsibilities or behavior to objects at runtime, without modifying their underlying class structure. It promotes the principle of Open-Closed, allowing new functionality to be added to existing objects without changing their code. It also supports the Single Responsibility Principle by separating concerns into individual decorators.

Applicability: The Decorator pattern is applicable when:

1. You want to add behavior or responsibilities to objects dynamically without modifying their original class.
2. You have a class hierarchy with many possible combinations of behaviors, and it is impractical to create subclasses for each combination.
3. You want to avoid a "bloated" class hierarchy with multiple subclasses for each possible combination of behaviors.

Participants:

* Component: The abstract class or interface that defines the interface for objects that can have additional responsibilities. It is the common interface for both the original objects and the decorators.
* ConcreteComponent: The original objects to which additional responsibilities can be added. They implement the Component interface.
* Decorator: The abstract class that extends the Component interface and acts as a base for concrete decorators. It maintains a reference to a Component object and delegates calls to it.
* ConcreteDecorator: The concrete subclasses of Decorator that add specific responsibilities or behavior to the component. They extend the Decorator class and override or add additional methods.

Collaborations: The client code interacts with objects through the Component interface, treating both the original objects and decorators uniformly. Decorators wrap the original objects and add their own behavior by delegating calls to the wrapped component and performing additional operations before or after the delegation.

Consequences:

* The Decorator pattern allows behavior to be added to objects dynamically at runtime.
* It provides a flexible alternative to subclassing for extending functionality, avoiding the need for an excessive number of subclasses.
* Decorators can be combined to create different combinations of behavior, enabling a high degree of flexibility.
* However, the pattern can result in a large number of small classes if multiple decorators are used, leading to increased complexity.

Implementation:

* The Decorator pattern can be implemented by defining abstract classes or interfaces for the Component and Decorator, and concrete subclasses for ConcreteComponent and ConcreteDecorator.
* Decorators maintain a reference to the wrapped component and delegate method calls to it, possibly adding their own behavior before or after the delegation.
* Language-dependent issues may arise, such as the need to handle constructor chaining, ensuring the correct order of decoration, and managing the reference to the wrapped component.

Known Uses: The Decorator pattern is commonly used in graphical user interface frameworks and input/output streams. For example, in Java, the java.io package uses the Decorator pattern extensively, where various decorators add functionalities such as buffering, encryption, and compression to the basic input/output streams.

Related patterns: The Decorator pattern is related to other structural patterns, such as the Adapter pattern, which provides a different interface for an object, and the Composite pattern, which allows objects to be composed into tree structures. The Decorator pattern can also be combined with the Factory pattern to create decorated objects dynamically.

Pattern Name: Prototype

Intent: The Prototype design pattern allows objects to be created by cloning or copying existing objects, rather than creating new objects from scratch. It provides a way to create new objects with pre-initialized state, avoiding the overhead of creating objects through constructors or factory methods.

Motivation: The Prototype pattern addresses the need to create new objects that are similar to existing objects, but with potentially different initial state or configuration. It avoids the costly process of object creation by cloning or copying existing objects. It is especially useful when creating complex objects that are costly to create or when a class hierarchy of objects needs to be created.

Applicability: The Prototype pattern is applicable when:

1. Creating objects by cloning or copying existing objects is more efficient than creating new objects from scratch.
2. Objects can be created with different initial states or configurations.
3. A class hierarchy of objects needs to be created, and the exact class of the objects may not be known in advance.

Participants:

* Prototype: The abstract class or interface that defines the cloning or copying operations. It declares a method for creating a copy of itself.
* ConcretePrototype: The concrete subclasses that implement the cloning or copying operations defined by the Prototype. They provide their own implementation of the clone method.
* Client: The client code that interacts with the Prototype to create new objects by cloning or copying existing objects.

Collaborations: The client code interacts with the Prototype to create new objects. It requests the Prototype to create a copy of itself by calling the clone method. The concrete prototypes implement the clone method to create a copy of themselves, including their internal state or configuration.

Consequences:

* The Prototype pattern allows new objects to be created by cloning or copying existing objects, avoiding the overhead of object creation from scratch.
* It provides a flexible way to create objects with different initial states or configurations.
* The pattern promotes encapsulation, as the cloning or copying logic is encapsulated within the Prototype and its concrete subclasses.
* However, cloning complex objects may be challenging, especially when dealing with deep copies or circular references. Care must be taken to ensure that all internal objects are properly cloned.

Implementation:

* The Prototype pattern can be implemented by defining an abstract class or interface for the Prototype and concrete subclasses for the ConcretePrototype.
* The clone method is implemented in the concrete prototypes to create a copy of the object, including its internal state or configuration.
* Shallow or deep copying techniques can be used, depending on the requirements of the application.
* Language-dependent issues may arise, such as handling object references during cloning or copying and ensuring proper initialization of the cloned objects.

Known Uses: The Prototype pattern is commonly used in scenarios where object creation is expensive or complex. It is often seen in applications that involve creating instances of complex classes, such as graphical editors, game engines, or database frameworks.

Related patterns: The Prototype pattern is related to other creational patterns, such as the Abstract Factory pattern, which provides an interface for creating families of related objects, and the Builder pattern, which focuses on step-by-step object construction. The Prototype pattern can also be combined with the Composite pattern to clone complex composite structures.

Pattern Name: Facade

Intent: The Facade design pattern provides a simplified interface or a high-level interface to a complex subsystem or a set of related interfaces. It encapsulates the complexity of the subsystem and provides a single entry point for the client code to interact with it, making it easier to use and understand.

Motivation: The Facade pattern addresses the need to simplify and unify the interface of a complex subsystem. It hides the complexity of the underlying system and provides a more straightforward interface for clients to interact with. It promotes loose coupling between the client code and the subsystem, improving maintainability and ease of use.

Applicability: The Facade pattern is applicable when:

1. You want to provide a simple and unified interface to a complex subsystem.

2. The complexity of the subsystem should be hidden from the client code.

3. You want to decouple the client code from the subsystem, promoting flexibility and maintainability.

4. You need to provide a higher-level interface that abstracts and organizes lower-level functionality.

Participants:

- Facade: The facade class or interface that provides a simplified interface to the subsystem. It encapsulates the complexity of the subsystem and delegates client requests to appropriate subsystem objects.

- Subsystem: The complex subsystem or a set of related classes and interfaces that the facade simplifies. The subsystem objects perform the actual work requested by the facade and may have their own interfaces.

- Client: The client code that interacts with the facade to utilize the functionality of the subsystem. The client code does not need to directly interact with the subsystem objects.

Collaborations: The client code interacts with the facade, using its simplified interface to access the subsystem functionality. The facade delegates the client requests to the appropriate subsystem objects, coordinating their actions if needed. The client code remains unaware of the complexity and internal workings of the subsystem.

Consequences:

- The Facade pattern simplifies the client code by providing a high-level interface to a complex subsystem, reducing its cognitive load.

- It decouples the client code from the subsystem, promoting flexibility and ease of maintenance.

- The pattern improves code organization and encapsulation, as the complexity of the subsystem is encapsulated within the facade.

- However, introducing a facade may create an additional layer of abstraction, and there is a risk of the facade becoming bloated if it exposes too many methods or functionalities.

Implementation:

- The Facade pattern can be implemented by creating a facade class or interface that wraps and delegates requests to the appropriate subsystem objects.

- The facade can provide a simplified interface that exposes only the necessary functionality of the subsystem to the client code.

- It can coordinate the actions of multiple subsystem objects if required.

- Language-dependent issues may arise, such as the need to handle object creation and initialization, and the management of subsystem dependencies.

Known Uses: The Facade pattern is commonly used in various software systems, such as libraries, frameworks, and APIs. It is often seen in complex systems where a simplified interface is needed to interact with a multitude of subsystem components. For example, in web development, a web framework may provide a facade that simplifies database access, session management, and routing for the client code.

Related patterns: The Facade pattern is related to other design patterns, such as the Adapter pattern, which provides a different interface to an existing object, and the Mediator pattern, which defines a central mediator object that encapsulates communication between other objects. The Facade pattern can also be combined with other patterns, such as the Singleton pattern, to provide a globally accessible facade instance.

Pattern Name: Singleton

Intent: The Singleton design pattern ensures that a class has only one instance and provides a global point of access to it. It is used when there should be exactly one instance of a class available throughout the system, and clients need a convenient way to access that instance.

Motivation: The Singleton pattern addresses the need to have a single, shared instance of a class in scenarios where creating multiple instances would be wasteful or inappropriate. It provides a centralized access point to the instance, allowing clients to easily obtain the same instance across different parts of the system.

Applicability: The Singleton pattern is applicable when:

1. There should be exactly one instance of a class in the system.

2. Clients need a global point of access to the instance.

3. The single instance needs to be easily accessible by different parts of the system.

4. Lazy initialization of the instance is desired.

Participants:

- Singleton: The class that implements the Singleton pattern. It ensures that only one instance of the class is created and provides a global access point to that instance.

- Client: The code that uses the Singleton instance. It accesses the Singleton through the global access point provided by the Singleton class.

Collaborations: Clients access the Singleton instance through the global access point provided by the Singleton class. The Singleton class is responsible for creating the single instance if it does not already exist and returning the same instance on subsequent requests.

Consequences:

- The Singleton pattern ensures that there is only one instance of a class in the system.

- It provides a global access point to the instance, allowing clients to conveniently access it from anywhere in the system.

- The pattern promotes encapsulation, as the details of instance creation and management are hidden within the Singleton class.

- However, the pattern can introduce global state and make testing and maintenance more difficult. It may also introduce potential thread-safety issues if not implemented correctly.

Implementation:

- The Singleton pattern can be implemented by declaring a private constructor to prevent external instantiation, and a static method or property that provides access to the single instance.

- The first time the static method or property is called, the Singleton class creates the single instance. On subsequent calls, it returns the existing instance.

- Lazy initialization techniques can be employed to defer the creation of the instance until it is actually needed.

- Care must be taken to ensure thread-safety in multi-threaded environments, either through synchronization or other concurrency control mechanisms.

Known Uses: The Singleton pattern is commonly used in various scenarios, such as managing access to shared resources, managing database connections, logging, caching, and configuration settings. It is often seen in frameworks, libraries, and applications where a single instance of a particular class needs to be accessed globally.

Related patterns: The Singleton pattern is related to other creational patterns, such as the Abstract Factory pattern, which provides an interface for creating families of related objects, and the Prototype pattern, which focuses on creating new objects by cloning existing ones. The Singleton pattern can also be combined with other patterns, such as the Facade pattern, to provide a globally accessible instance of the facade.

Pattern Name: Flyweight

Intent: The Flyweight design pattern aims to minimize memory usage and improve performance by sharing common data across multiple objects. It achieves this by splitting objects into intrinsic and extrinsic state, where intrinsic state is shared among multiple objects, while extrinsic state is unique to each object. This pattern is particularly useful when dealing with large numbers of similar objects.

Motivation: The Flyweight pattern addresses the need to reduce memory consumption when dealing with a large number of objects with similar characteristics. By sharing common data, the pattern allows for efficient storage and improved performance. It is especially beneficial in situations where the cost of creating and maintaining objects is high.

Applicability: The Flyweight pattern is applicable when:

1. An application needs to support a large number of objects that have similar characteristics.

2. The state of objects can be divided into intrinsic and extrinsic parts, where the intrinsic state can be shared.

3. The application requires a significant reduction in memory usage and improved performance.

Participants:

- Flyweight: The interface or abstract class that defines the methods for manipulating the shared and non-shared state of flyweight objects.

- ConcreteFlyweight: The concrete implementation of the Flyweight interface that represents the shared intrinsic state. Multiple objects can refer to the same ConcreteFlyweight instance.

- FlyweightFactory: The factory class that manages the creation and retrieval of flyweight objects. It ensures that flyweight objects are shared and reused.

- Client: The client code that uses flyweight objects. It maintains references to the extrinsic state of the flyweight objects and interacts with them through the Flyweight interface.

Collaborations: The client code interacts with the FlyweightFactory to obtain flyweight objects. The factory creates and manages the flyweight objects, ensuring that the same flyweight is shared when requested multiple times. The client provides the unique extrinsic state to the flyweight objects when necessary.

Consequences:

- The Flyweight pattern reduces memory consumption by sharing common state among multiple objects.

- It improves performance by avoiding the need to create new objects for each similar situation.

- The pattern enhances the scalability of applications by efficiently handling a large number of objects.

- However, the pattern may introduce additional complexity, especially when dealing with mutable state. Care must be taken to ensure that the shared state is not modified unintentionally.

Implementation:

- The Flyweight pattern can be implemented by defining a Flyweight interface or abstract class for the shared state and creating ConcreteFlyweight classes that represent specific instances of the shared state.

- The FlyweightFactory is responsible for creating and managing the flyweight objects, ensuring that they are shared and reused.

- The client code interacts with the Flyweight objects through the Flyweight interface, providing the unique extrinsic state as required.

- Language-dependent issues may arise, such as handling thread-safety and managing object identity and equality.

Known Uses: The Flyweight pattern is commonly used in situations where a large number of objects with similar characteristics need to be efficiently managed. It is often seen in graphical applications, such as image processing or document editors, where shared resources like fonts, colors, or images are used by multiple objects.

Related patterns: The Flyweight pattern is related to other structural patterns, such as the Composite pattern, which allows objects to be composed into tree structures, and the Proxy pattern, which provides a surrogate or placeholder for another object. The Flyweight pattern can also be combined with other patterns, such as the Singleton pattern, to ensure that only one instance of a particular flyweight exists.

Pattern Name: Adapter

Intent: The Adapter design pattern converts the interface of a class into another interface that clients expect. It allows incompatible classes to work together by providing a common interface. The Adapter pattern enables the reuse of existing classes without modifying their original code.

Motivation: The Adapter pattern addresses the need to make existing classes work together when their interfaces are incompatible. It avoids the need to modify the existing classes by introducing an adapter that translates requests from the client to the appropriate calls in the adapted class. This promotes code reuse and flexibility.

Applicability: The Adapter pattern is applicable when:

1. Two or more classes with incompatible interfaces need to work together.

2. You want to reuse an existing class that does not have the desired interface without modifying its code.

3. A class needs to interact with multiple classes that have different interfaces.

Participants:

- Target: The interface that defines the desired interface that the client code expects.

- Adaptee: The class that needs to be adapted to work with the target interface. Its interface is incompatible with the target interface.

- Adapter: The class that adapts the interface of the Adaptee to the Target interface. It acts as a bridge between the client code and the Adaptee, translating requests from the target interface to the appropriate calls in the Adaptee.

Collaborations: The client code interacts with the Adapter through the Target interface, making requests that are translated by the Adapter to the appropriate calls in the Adaptee. The Adapter communicates with the Adaptee to fulfill the requests from the client.

Consequences:

- The Adapter pattern allows classes with incompatible interfaces to work together.

- It promotes code reuse by adapting existing classes without modifying their original code.

- The pattern enhances flexibility by allowing a class to interact with multiple classes through a common interface.

- However, the introduction of the Adapter may add an extra layer of indirection and can impact performance. Additionally, it may increase code complexity and maintenance efforts.

Implementation:

- The Adapter pattern can be implemented by creating a class that implements the Target interface and internally holds an instance of the Adaptee.

- The Adapter class maps the requests from the Target interface to the corresponding methods or operations of the Adaptee.

- The Adaptee class can be either an existing class or a third-party class that needs to be adapted.

- The Adapter can be implemented as either a class adapter (using inheritance) or an object adapter (using composition).

- Language-dependent issues may arise, such as handling different programming languages, incompatible data types, or adapting between synchronous and asynchronous communication.

Known Uses: The Adapter pattern is widely used in various software systems and libraries, especially when integrating different components or dealing with legacy code. It is often seen in scenarios where existing classes or third-party libraries need to be adapted to fit into a unified interface or framework.

Related patterns: The Adapter pattern is related to other structural patterns, such as the Bridge pattern, which focuses on decoupling abstraction from implementation, and the Facade pattern, which provides a simplified interface to a complex subsystem. The Adapter pattern can also be combined with other patterns, such as the Decorator pattern, to provide additional functionality to the adapted class.

Pattern Name: Proxy

Intent: The Proxy design pattern provides a surrogate or placeholder for another object to control access to it. It allows for the implementation of additional behavior or restrictions on the object being proxied while maintaining the same interface.

Motivation: The Proxy pattern addresses the need to control access to an object or add additional functionality to it, without modifying the object itself. It allows for the separation of concerns by introducing a proxy object that acts as an intermediary between the client code and the real object.

Applicability: The Proxy pattern is applicable when:

1. There is a need to control access to an object or add additional behavior to it, such as logging, caching, or security checks.

2. The object to be accessed is resource-intensive or remote, and it is desirable to create a lightweight representation of it.

3. The object to be accessed needs to be created on-demand or only when necessary.

Participants:

- Subject: The interface that both the Proxy and the RealSubject implement. It defines the common methods or operations that the client code can invoke on the object.

- RealSubject: The real object that the Proxy represents. It implements the Subject interface and contains the actual business logic or functionality.

- Proxy: The proxy object that acts as a surrogate for the RealSubject. It implements the Subject interface and controls access to the RealSubject. It may perform additional tasks before or after delegating the request to the RealSubject.

Collaborations: The client code interacts with the Proxy through the Subject interface. The Proxy intercepts the client requests, performs any necessary pre- or post-processing, and then delegates the request to the RealSubject.

Consequences:

- The Proxy pattern allows for additional functionality to be added to an object without modifying its implementation.

- It provides a level of indirection and control over access to the RealSubject.

- The pattern can enhance performance by lazily initializing or caching the RealSubject, especially in resource-intensive or remote scenarios.

- However, the introduction of a proxy may introduce some overhead due to the additional level of indirection and delegation.

Implementation:

- The Proxy pattern can be implemented by creating a Proxy class that implements the Subject interface and internally holds an instance of the RealSubject.

- The Proxy class intercepts client requests, performs any necessary preprocessing or post-processing, and delegates the actual execution to the RealSubject.

- The Proxy can control access to the RealSubject by applying additional checks or restrictions.

- Different types of proxies can be implemented, such as virtual proxies, remote proxies, protection proxies, or caching proxies, depending on the specific requirements.

- Language-dependent issues may arise, such as handling object creation, serialization, or communication between different components.

Known Uses: The Proxy pattern is commonly used in various scenarios, such as remote communication, logging, caching, lazy initialization, access control, or performance optimization. It is often seen in distributed systems, web services, ORM frameworks, and UI frameworks.

Related patterns: The Proxy pattern is related to other structural patterns, such as the Adapter pattern, which provides a different interface to an existing object, and the Decorator pattern, which dynamically adds functionality to an object. The Proxy pattern can also be combined with other patterns, such as the Singleton pattern, to control the instantiation and access to a single instance of the RealSubject.

Pattern Name: Chain of Responsibility

Intent: The Chain of Responsibility design pattern decouples the sender of a request from its receivers by allowing multiple objects to have the opportunity to handle the request. It enables the building of a chain of objects, each having a chance to process the request or pass it to the next object in the chain.

Motivation: The Chain of Responsibility pattern addresses the need to handle a request in a flexible and dynamic manner, where the sender of the request is unaware of the exact receiver or processing logic. It promotes loose coupling between objects and provides a way to achieve a level of decoupling and flexibility in processing requests.

Applicability: The Chain of Responsibility pattern is applicable when:

1. There are multiple objects that can handle a request, and the specific handler is not known in advance.

2. The request needs to be processed by one or more objects dynamically at runtime.

3. The sender of the request should not be coupled to the receivers, allowing for easy addition or removal of handlers.

Participants:

- Handler: The interface or abstract class that defines the common interface for handling requests. It contains a reference to the next handler in the chain.

- ConcreteHandler: The concrete implementation of the Handler interface that handles the request if it is capable, or passes it to the next handler in the chain.

- Client: The code that initiates the request and starts the chain of handlers. It is unaware of the specific handlers in the chain and communicates with the first handler.

Collaborations: The client code initiates the request and passes it to the first handler in the chain. Each handler has the opportunity to handle the request or pass it to the next handler in the chain. The chain is traversed until a handler successfully handles the request or until the end of the chain is reached.

Consequences:

- The Chain of Responsibility pattern decouples the sender of a request from its receivers, promoting flexibility and extensibility.

- It allows for dynamic selection and sequencing of handlers without modifying the client code.

- The pattern simplifies the addition or removal of handlers, as they can be easily inserted or reconfigured in the chain.

- However, it may lead to requests going unhandled if the chain is not properly configured or if there is no appropriate handler in the chain.

Implementation:

- The Chain of Responsibility pattern can be implemented by creating a chain of handler objects, each implementing the same interface or extending the same abstract class.

- Each handler holds a reference to the next handler in the chain and has the option to handle the request or pass it to the next handler.

- The client code initiates the request by passing it to the first handler in the chain.

- Handlers can be added or removed dynamically at runtime, allowing for flexible configuration of the chain.

- Care should be taken to avoid creating circular dependencies in the chain and ensure proper termination conditions.

Known Uses: The Chain of Responsibility pattern is commonly used in various scenarios, such as event handling, logging systems, user input processing, and authorization frameworks. It is often seen in systems where there is a need to dynamically process requests based on their characteristics and without tightly coupling the sender and receiver.

Related patterns: The Chain of Responsibility pattern is related to other behavioral patterns, such as the Command pattern, which encapsulates a request as an object, and the Observer pattern, which provides a way to subscribe and notify multiple objects about changes. The Chain of Responsibility pattern can be combined with these patterns to enhance the flexibility and functionality of the system.

Observer