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.

…

Pattern Name: **Factory** (Creational)

Intent: provides an interface for creating objects but allows subclasses or implementing classes to decide which class to instantiate. It encapsulates the object creation logic and promotes loose coupling between the client code and the created objects.

Motivation: In software systems, there is often a need to create objects of different types based on certain conditions or parameters. However, directly instantiating objects in the client code can result in tight coupling and dependencies on specific classes. The Factory pattern addresses this by introducing a separate factory class that encapsulates the creation logic. This way, the client code can use the factory to create objects without being aware of the specific implementation classes.

Applicability:

1. The client code needs to create objects of different types, but it should not depend on concrete class implementations.

2. The client code should be decoupled from the process of object creation and should not have knowledge of the specific classes being instantiated.

3. The creation of objects may involve complex logic or dependencies that should be encapsulated in a separate class.

Participants:

- Creator: The abstract class or interface that declares the factory method(s) for creating objects. It may also provide default implementations or common behavior for the created objects.

- ConcreteCreator: The concrete subclass or implementing class of the Creator. It overrides the factory method(s) to provide the specific implementation for creating objects.

- Product: The abstract class or interface that defines the common interface or behavior of the objects created by the factory.

- ConcreteProduct: The concrete class that implements the Product interface. It represents the specific type of object created by the factory.

Collaborations: The client code interacts with the Creator class and calls its factory method(s) to create objects. The Creator delegates the object creation to the ConcreteCreator, which returns an instance of the ConcreteProduct. The client code receives the object through the Product interface, allowing it to work with the object without knowing its specific implementation.

Consequences:

- The Factory pattern promotes loose coupling between the client code and the created objects by encapsulating the object creation logic in a separate class.

- It allows for flexibility in creating objects of different types by relying on subclasses or implementing classes to decide the specific class to instantiate.

- The pattern simplifies the client code by providing a consistent interface to create objects, without exposing the details of the instantiation process.

- It enhances code maintainability and extensibility by centralizing the creation logic, making it easier to add new product types or modify the creation process.

- However, introducing new product types may require modifying the Creator interface and all its implementations.

Implementation:

- Implement the Factory pattern by defining a Creator abstract class or interface that declares the factory method(s) for creating objects.

- ConcreteCreator classes extend or implement the Creator and provide the specific implementation for the factory method(s), instantiating and returning the ConcreteProduct.

- The Product interface or abstract class defines the common interface or behavior of the objects created by the factory.

- ConcreteProduct classes implement the Product interface and represent the specific types of objects created by the factory.

Known Uses: The Factory pattern is widely used in various software systems and frameworks. It is commonly seen in scenarios where object creation involves complex logic, dependency injection, or configuration, such as in dependency injection frameworks, abstract factory patterns, and plugin architectures.

Related patterns: related to other creational patterns, such as the Abstract Factory pattern, which provides an interface for creating families of related objects, and the Singleton pattern, which ensures that only a single instance of a class is created. The Factory pattern can be combined with these patterns to provide a more flexible and configurable approach to object creation.

Pattern Name: **Abstract Factory** (Creation)

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** (Structural)

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 (Creational)

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 (Structural)

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 (Structural)

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** (Creational)

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 (Structural)

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 (Creational)

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 (Structural)

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 (Structural)

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 (Structural)

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** (Behavioural)

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.

Pattern Name: **Observer** (Behavioural)

Intent: The Observer design pattern defines a one-to-many dependency between objects, where changes in one object (the subject) trigger updates to its dependent objects (the observers). It allows for loose coupling between objects and enables them to maintain consistency and stay in sync with each other.

Motivation: The Observer pattern addresses the need for objects to be notified and updated when the state of another object changes. It promotes decoupling and flexibility by allowing multiple observers to subscribe to and receive updates from a subject, without requiring them to have explicit knowledge of each other.

Applicability: The Observer pattern is applicable when:

1. There is a one-to-many relationship between objects, where changes in one object should be propagated to multiple other objects.

2. The subjects and observers have a loosely coupled relationship, and they should be able to interact without explicit knowledge of each other.

3. Objects need to maintain consistency and stay synchronized with each other.

Participants:

- Subject: The object that maintains the state and sends notifications to its observers when the state changes. It provides methods for subscribing and unsubscribing observers, as well as notifying them of changes.

- Observer: The interface or abstract class that defines the common interface for receiving updates from the subject. It contains a method that is called by the subject when a change occurs.

- ConcreteSubject: The concrete implementation of the Subject interface. It maintains the state and sends notifications to its observers when changes occur.

- ConcreteObserver: The concrete implementation of the Observer interface. It registers with the subject to receive updates and implements the update method to handle the notifications.

Collaborations: The observers register with the subject to receive updates. When the subject's state changes, it notifies all registered observers by invoking their update method. The observers can then retrieve the updated information from the subject and perform any necessary actions.

Consequences:

- The Observer pattern promotes loose coupling between subjects and observers, allowing for flexibility and extensibility.

- It supports the principle of separation of concerns by decoupling the business logic from the update mechanism.

- The pattern allows for dynamic addition and removal of observers at runtime.

- However, there may be potential performance overhead if the subject has a large number of observers, and there can be issues related to the order of notification if observers depend on each other.

Implementation:

- The Observer pattern can be implemented by creating a Subject class that maintains a list of registered observers and provides methods for adding, removing, and notifying them.

- Observers implement the Observer interface, which typically includes an update method that is called by the subject when a change occurs.

- ConcreteSubject classes extend the Subject class and provide specific implementations of the state and the mechanism for notifying observers.

- Observers can be implemented as separate classes or as anonymous functions, depending on the language and requirements.

- Subjects should provide methods to manage subscriptions and handle notifications in a thread-safe manner if used in multi-threaded environments.

Known Uses: The Observer pattern is commonly used in various scenarios, such as event-driven systems, graphical user interfaces, model-view-controller architectures, and reactive programming frameworks. It is often seen in situations where changes in one object should be propagated and reflected in other objects that have expressed interest in those changes.

Related patterns: The Observer pattern is related to other behavioral patterns, such as the Mediator pattern, which centralizes communication and coordination between objects, and the Publish-Subscribe pattern, which uses a message broker to distribute notifications to multiple subscribers. The Observer pattern can be combined with these patterns to achieve more complex communication and coordination scenarios.

Pattern Name: Command (Behavioural)

Intent: encapsulates a request as an object, thereby decoupling the sender of the request from the object that performs the action. It allows for the parameterization of clients with different requests, queueing or logging requests, and supporting undoable operations.

Motivation: addresses the need to decouple the sender of a request from its receiver, providing flexibility and extensibility. It enables the encapsulation of a request as an object, allowing it to be parameterized, queued, logged, or even undone. This pattern promotes loose coupling and allows for the separation of concerns between the sender and receiver of a request.

Applicability: The Command pattern is applicable when:

1. You want to parameterize objects with requests, allowing clients to make requests without knowing the specific operations or receivers involved.

2. You need to queue, log, or audit requests for later execution.

3. You want to support undoable operations, where each command object stores the necessary information to reverse its effects.

Participants:

- Command: The interface or abstract class that declares the common methods for executing a request. It may also include methods for undoing or redoing the command.

- ConcreteCommand: The concrete implementation of the Command interface. It encapsulates a specific request and the receiver object responsible for executing that request.

- Client: The object that creates and sets up the command objects. It specifies the receiver of the request and the method to be invoked.

- Receiver: The object that performs the actual operations or actions associated with a request. It contains the business logic or functionality related to the request.

Collaborations: The client creates a command object, specifies the receiver of the request, and sets up the command with the necessary information. The client then passes the command object to an invoker, which may queue, log, or execute the command later. When the command is executed, it invokes the appropriate method on the receiver object, which performs the actual operations.

Consequences:

- The Command pattern decouples the sender of a request from the object that performs the action, promoting loose coupling and flexibility.

- It allows for the parameterization and queuing of requests, enabling complex operations and workflows.

- The pattern supports the implementation of undoable operations by storing the necessary information in the command objects.

- However, the Command pattern can introduce an overhead of creating and managing command objects, and it may increase code complexity in scenarios with a large number of command classes.

Implementation:

- The Command pattern can be implemented by defining a Command interface or abstract class with an execute method that represents the operation to be performed.

- ConcreteCommand classes implement the Command interface and encapsulate a specific request and the receiver object responsible for executing that request.

- The Client creates and configures the command objects, specifying the receiver and any necessary parameters.

- An Invoker class may be used to manage and execute the commands, potentially supporting undo and redo operations or maintaining a queue of commands.

- Care should be taken to properly handle the dependencies and lifecycle management of the receiver objects.

Known Uses: The Command pattern is commonly used in various scenarios, such as GUI applications, transactional systems, logging frameworks, and multi-level undo/redo mechanisms. It is often seen in situations where requests need to be encapsulated, queued, logged, or undone.

Related patterns: The Command pattern is related to other behavioral patterns, such as the Memento pattern, which supports capturing and restoring the internal state of objects, and the Composite pattern, which can be used to create hierarchies of commands. The Command pattern can be combined with these patterns to achieve more complex functionality and interactions.

Pattern Name: State (Behavioural)

Intent: The State design pattern allows an object to alter its behavior when its internal state changes. It enables the object to appear as if it has changed its class, providing a clean separation between state-specific behavior and general functionality.

Motivation: The State pattern addresses the need to change an object's behavior dynamically based on its internal state. Rather than using conditional statements to handle different states, the pattern encapsulates each state into a separate class and allows the object to delegate behavior to the current state. This promotes loose coupling, improves maintainability, and simplifies the addition of new states.

Applicability: The State pattern is applicable when:

1. An object's behavior needs to change based on its internal state.

2. The state-specific behavior can be encapsulated into separate classes, allowing for easy addition or modification of states.

3. The context object (the object whose behavior changes) should not have to know the details of each state, promoting loose coupling and separation of concerns.

Participants:

- Context: The object whose behavior changes based on its internal state. It maintains a reference to the current state object and delegates state-specific behavior to that object.

- State: The interface or abstract class that defines the common methods for the different states. It encapsulates the behavior associated with a particular state.

- ConcreteState: The concrete implementation of the State interface. Each ConcreteState class represents a specific state and provides the implementation for the state-specific behavior.

Collaborations: The Context object interacts with the State objects through the State interface. When the internal state of the Context changes, it delegates the state-specific behavior to the current State object. The State objects can also modify the internal state of the Context if necessary.

Consequences:

- The State pattern promotes loose coupling by encapsulating each state into a separate class, allowing for easy addition or modification of states.

- It simplifies the Context class by removing conditional statements related to different states and delegating behavior to State objects.

- The pattern improves the maintainability and extensibility of the codebase by isolating state-specific behavior into separate classes.

- However, the State pattern may increase the number of classes in the system, especially if there are numerous states with unique behavior.

Implementation:

- The State pattern can be implemented by defining a State interface or abstract class that declares the common methods for the different states.

- ConcreteState classes implement the State interface and provide the specific implementation for each state's behavior.

- The Context class maintains a reference to the current state object and delegates state-specific behavior to that object.

- The Context class may provide methods to set the state or modify the internal state based on certain conditions.

- Care should be taken to ensure thread safety if the Context object is accessed by multiple threads.

Known Uses: The State pattern is commonly used in various scenarios, such as workflow management systems, game development, document editors, and vending machines. It is often seen in situations where an object's behavior needs to change dynamically based on its internal state.

Related patterns: The State pattern is related to other behavioral patterns, such as the Strategy pattern, which allows the selection of an algorithm or behavior at runtime, and the Decorator pattern, which dynamically adds functionality to an object. The State pattern can be combined with these patterns to achieve more complex behavior and flexibility in the system.

Pattern Name: Interpreter (Behavioural)

Intent: The Interpreter design pattern defines a representation for a grammar or language and provides a way to evaluate sentences or expressions in that language. It allows for the interpretation of textual input and the execution of corresponding actions.

Motivation: The Interpreter pattern addresses the need to interpret and evaluate expressions or sentences in a language. It is useful when there is a need to build a language processor or when a specific domain problem can be expressed in a language that can be interpreted. The pattern promotes the decoupling of grammar rules from their implementation and enables the definition of new operations or behaviors by extending the grammar.

Applicability: The Interpreter pattern is applicable when:

1. There is a need to interpret and evaluate sentences or expressions in a language.

2. The grammar or language can be represented as an abstract syntax tree (AST) or a set of rules.

3. The language or grammar is relatively simple and does not require complex parsing or compilation.

Participants:

- AbstractExpression: The interface or abstract class that declares the interpret method, which represents a rule or expression in the grammar.

- TerminalExpression: The concrete implementation of the AbstractExpression interface for terminal expressions in the grammar. It represents the basic building blocks of the language.

- NonterminalExpression: The concrete implementation of the AbstractExpression interface for nonterminal expressions in the grammar. It represents complex expressions that can be composed of multiple terminal and nonterminal expressions.

- Context: The context object that contains the information or state required for the interpretation.

Collaborations: The client creates an abstract syntax tree (AST) of the language or expression to be interpreted. The AST is composed of terminal and nonterminal expressions. When the client wants to evaluate the expression, it invokes the interpret method on the root of the AST, which recursively evaluates the subexpressions and performs the corresponding actions.

Consequences:

- The Interpreter pattern provides a way to interpret and evaluate sentences or expressions in a language.

- It separates the grammar rules from their implementation, allowing for flexibility and extensibility.

- The pattern can simplify the process of implementing new operations or behaviors by extending the grammar.

- However, the pattern can lead to a large number of classes if the language or grammar is complex, and it may have performance implications due to the recursive evaluation of expressions.

Implementation:

- The Interpreter pattern can be implemented by defining an AbstractExpression interface or abstract class that declares the interpret method.

- Concrete expressions, such as TerminalExpression and NonterminalExpression classes, implement the AbstractExpression interface and provide the specific implementation for interpreting their respective expressions.

- The client is responsible for creating and configuring the abstract syntax tree (AST) of the language to be interpreted.

- The Context object contains the necessary information or state required for the interpretation and is passed to the interpret method of expressions.

- The interpretation process often involves recursively evaluating the subexpressions and performing the corresponding actions.

Known Uses: The Interpreter pattern is commonly used in scenarios where a language or expression needs to be interpreted and evaluated. It is often seen in areas such as programming language interpreters, query languages, regular expression matching, and rule-based systems.

Related patterns: The Interpreter pattern is related to other patterns that deal with processing languages or expressions, such as the Composite pattern, which can be used to build the abstract syntax tree (AST), and the Visitor pattern, which allows for the separation of operations from the classes they operate on. The Interpreter pattern can be combined with these patterns to achieve more complex language processing capabilities.

Pattern Name: Strategy (Behavioural)

Intent: The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it, promoting flexibility, and enabling runtime selection of algorithms.

Motivation: The Strategy pattern addresses the need to dynamically select and interchange different algorithms or behaviors at runtime. It promotes the separation of algorithms from the classes that use them, allowing for easy extension, maintenance, and customization. The pattern enables the client to select the appropriate strategy without needing to know the implementation details.

Applicability: The Strategy pattern is applicable when:

1. Different variations of an algorithm or behavior need to be implemented and selected at runtime.

2. The algorithm implementation details should be encapsulated and kept separate from the client code to promote flexibility and maintainability.

3. The client should be able to use different strategies interchangeably without modifying its code.

Participants:

- Strategy: The interface or abstract class that declares the common methods for different strategies. It defines a contract that all strategies must follow.

- ConcreteStrategy: The concrete implementation of the Strategy interface. Each ConcreteStrategy class encapsulates a specific algorithm or behavior variation.

- Context: The object that contains a reference to a Strategy object and uses it to perform its operations or actions. It is not aware of the specific strategy implementation and relies on the Strategy interface.

Collaborations: The Context object interacts with the Strategy object through the Strategy interface. When the client needs to use a specific algorithm or behavior, it sets the appropriate ConcreteStrategy object into the Context. The Context delegates the execution of the algorithm or behavior to the currently set Strategy object.

Consequences:

- The Strategy pattern allows for the selection and interchangeability of algorithms or behaviors at runtime, promoting flexibility.

- It encapsulates the algorithm implementation details and separates them from the client code, improving maintainability and extensibility.

- The pattern enables the addition of new strategies without modifying existing client code.

- However, the number of classes and the complexity of the system may increase due to the introduction of multiple ConcreteStrategy classes.

Implementation:

- The Strategy pattern can be implemented by defining a Strategy interface or abstract class that declares the common methods for different strategies.

- ConcreteStrategy classes implement the Strategy interface and provide the specific implementation for each strategy.

- The Context class contains a reference to the Strategy object and delegates its operations or actions to that object.

- The client can select and set the appropriate strategy into the Context based on runtime conditions or requirements.

Known Uses: The Strategy pattern is commonly used in various scenarios, such as sorting algorithms, pricing strategies, encryption algorithms, and user interface frameworks. It is often seen in situations where different variations of an algorithm or behavior need to be used interchangeably.

Related patterns: The Strategy pattern is related to other behavioral patterns, such as the Template Method pattern, which provides a framework for defining an algorithm's skeleton and allowing subclasses to override specific steps. The Strategy pattern can be combined with the Template Method pattern to customize the behavior of the algorithm at runtime.

Pattern Name: Iterator (Behavioural)

Intent: The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It decouples the traversal algorithm from the aggregate, allowing for multiple iterations and traversal strategies.

Motivation: The Iterator pattern addresses the need to traverse the elements of a collection or aggregate object without exposing its internal structure. It promotes the principle of "separation of concerns" by separating the traversal logic from the collection, making the collection simpler and more focused on its core functionality. The pattern also provides a consistent interface for iterating over different types of collections.

Applicability: The Iterator pattern is applicable when:

1. You need to access the elements of an aggregate object sequentially without exposing its internal representation.

2. You want to provide a uniform interface for traversing different types of collections or aggregates.

3. You need to support multiple iterations or different traversal strategies for the same collection.

Participants:

- Iterator: The interface or abstract class that defines the common methods for iterating over elements of a collection. It typically includes methods like hasNext() to check if there are more elements, and next() to retrieve the next element.

- ConcreteIterator: The concrete implementation of the Iterator interface. It maintains the current position in the collection and provides the specific implementation of iteration logic.

- Aggregate: The interface or abstract class that defines the methods for creating an Iterator object. It represents the collection or aggregate object that needs to be iterated.

- ConcreteAggregate: The concrete implementation of the Aggregate interface. It creates a specific ConcreteIterator object that is capable of traversing the elements of the collection.

Collaborations: The client interacts with the Aggregate object to obtain an Iterator object. The Iterator object is responsible for traversing the elements of the collection sequentially and providing access to each element. The client can use the Iterator's methods like hasNext() and next() to iterate over the elements until there are no more elements left.

Consequences:

- The Iterator pattern provides a uniform and encapsulated way to iterate over elements of a collection, regardless of its internal structure.

- It allows for the separation of concerns between the collection and the iteration logic, making the collection simpler and more focused.

- The pattern supports multiple iterations and different traversal strategies for the same collection.

- It promotes code reuse and improves the maintainability of the codebase.

- However, the Iterator pattern may add some overhead due to the additional abstraction and object creation.

Implementation:

- The Iterator pattern can be implemented by defining an Iterator interface or abstract class that declares the common methods for iterating over elements.

- ConcreteIterator classes implement the Iterator interface and provide the specific implementation for traversing the elements of a particular collection.

- The Aggregate interface or abstract class defines a method for creating an Iterator object.

- ConcreteAggregate classes implement the Aggregate interface and create a ConcreteIterator object that is appropriate for iterating over the specific collection type.

Known Uses: The Iterator pattern is widely used in programming languages and frameworks. It is often used in scenarios where collections or aggregates need to be iterated, such as in database result sets, lists, trees, and queues.

Related patterns: The Iterator pattern is related to other patterns that involve traversing or accessing elements, such as the Composite pattern, which can be used to traverse hierarchical structures, and the Visitor pattern, which allows for performing operations on elements without modifying their classes. The Iterator pattern can be combined with these patterns to achieve more complex behavior and interactions.

Pattern Name: **Template Method**
(Behavioural)

Intent: The Template Method design pattern defines the skeleton of an algorithm in a base class and allows subclasses to provide specific implementations for certain steps of the algorithm. It promotes code reuse, allows for variations in algorithm steps, and provides a way to enforce a common algorithm structure across multiple classes.

Motivation: The Template Method pattern addresses the need to define an algorithm's structure while allowing subclasses to provide specific implementations for certain steps. It avoids code duplication by encapsulating the common parts of the algorithm in a base class. The pattern enables variations in individual steps, promoting flexibility, and extensibility. It also enforces a consistent structure and behavior across different classes that share the same algorithm.

Applicability: The Template Method pattern is applicable when:

1. You have an algorithm that follows a common structure but has varying implementations for certain steps.

2. You want to avoid code duplication by encapsulating the common parts of the algorithm in a base class.

3. You need to enforce a consistent algorithm structure across multiple classes.

Participants:

- AbstractClass: The base class that defines the overall algorithm structure and contains the template method. It may also provide default implementations for certain steps.

- ConcreteClass: The concrete subclass that extends the AbstractClass and provides specific implementations for the individual steps of the algorithm.

Collaborations: The client interacts with the AbstractClass, which provides the template method that encapsulates the algorithm's structure. The template method calls specific methods (either abstract or with default implementations) at different steps of the algorithm. The ConcreteClass subclasses inherit the template method and provide their implementations for the individual steps, customizing the algorithm's behavior.

Consequences:

- The Template Method pattern promotes code reuse by encapsulating the common parts of the algorithm in a base class.

- It allows for variations in specific steps of the algorithm by letting subclasses provide their implementations.

- The pattern enforces a consistent algorithm structure across different classes that inherit from the same AbstractClass.

- It simplifies the client code by providing a high-level interface to the algorithm.

- However, the pattern may lead to a more complex class hierarchy, especially if there are many variations in the algorithm steps.

Implementation:

- The Template Method pattern can be implemented by defining an AbstractClass that declares the template method, as well as abstract or default methods for the individual steps.

- ConcreteClass subclasses extend the AbstractClass and provide their implementations for the specific steps.

- The template method in the AbstractClass defines the overall algorithm structure by calling the individual steps in a specific order.

- The client interacts with the AbstractClass and uses the template method to invoke the algorithm, without needing to know the details of the individual steps.

Known Uses: The Template Method pattern is commonly used in various frameworks and libraries. It is often seen in scenarios where there is a common algorithm structure shared by multiple classes, such as in GUI frameworks for handling events, frameworks for building parsers or compilers, and in software development lifecycle processes.

Related patterns: The Template Method pattern is related to other patterns that involve defining an algorithm's structure, such as the Strategy pattern, which allows for dynamically selecting different algorithms at runtime, and the Composite pattern, which can be used to build hierarchical structures. The Template Method pattern can be combined with these patterns to achieve more flexible and complex behavior.

Pattern Name: Mediator (Behavioural)

Intent: The Mediator design pattern promotes loose coupling between components by encapsulating their interactions within a central mediator object. It allows components to communicate and collaborate without having direct dependencies on each other, thus simplifying their individual responsibilities.

Motivation: In complex systems, components often need to communicate and coordinate with each other. However, direct dependencies between components can lead to tight coupling, making the system harder to understand, maintain, and extend. The Mediator pattern addresses this by introducing a mediator object that encapsulates the communication logic between components. This way, components can interact through the mediator without having explicit knowledge of each other.

Applicability: The Mediator pattern is applicable when:

1. A set of objects need to communicate and collaborate in a well-defined way.

2. Direct dependencies between objects should be avoided to reduce coupling.

3. Adding new components or changing the interaction logic between existing components should be flexible and easy.

Participants:

- Mediator: The interface or abstract class that defines the communication protocol between components. It typically includes methods for component registration, sending messages, and handling component interactions.

- ConcreteMediator: The concrete implementation of the Mediator interface. It manages the communication and collaboration between components by implementing the specific interaction logic.

- Colleague: The interface or abstract class that defines the common methods and communication interface for the components. It typically includes methods for sending and receiving messages through the mediator.

- ConcreteColleague: The concrete implementation of the Colleague interface. It represents a specific component that collaborates with other components through the mediator.

Collaborations: Colleague objects communicate with each other indirectly through the Mediator object. When a Colleague needs to send a message to another Colleague, it invokes the appropriate method on the Mediator, which handles the message and relays it to the intended recipient Colleague. The Mediator encapsulates the communication logic and facilitates the collaboration between the Colleague objects.

Consequences:

- The Mediator pattern promotes loose coupling between components by encapsulating their interactions within a central Mediator object.

- It simplifies the individual responsibilities of components by abstracting the communication logic.

- The pattern enhances maintainability and extensibility as changes in component interactions can be localized to the Mediator.

- However, the Mediator can become complex and overloaded if there are too many components or complex communication requirements.

Implementation:

- Implement the Mediator pattern by defining a Mediator interface or abstract class that declares the communication methods.

- ConcreteMediator classes implement the Mediator interface and provide the specific interaction logic between components.

- Colleague classes define the common interface for components and hold a reference to the Mediator.

- ConcreteColleague classes implement the Colleague interface and communicate with other components through the Mediator.

Known Uses: The Mediator pattern is commonly used in scenarios where there are complex interactions between multiple components, such as in graphical user interfaces, event-driven systems, chat applications, and multiplayer games. It is also used in software architectures that emphasize loose coupling and modularity.

Related patterns: The Mediator pattern is related to other patterns that deal with component communication and coordination, such as the Observer pattern, which allows components to subscribe and receive notifications from a central subject. The Mediator pattern can be combined with the Observer pattern to facilitate communication between multiple components and ensure consistency in their interactions.

Pattern Name: Memento (Behavioural)

Intent: The Memento design pattern allows capturing and restoring the internal state of an object without violating encapsulation. It provides a way to save and restore an object's state, enabling undo/redo functionality or checkpoints in the application.

Motivation: Objects often need to maintain different states during their lifetime, and there are cases where it is necessary to save and restore these states. However, exposing the internal state or providing direct access to it violates encapsulation. The Memento pattern solves this problem by introducing a separate Memento object that holds the state of the originator object. This way, the originator can save and restore its state without exposing it directly.

Applicability: The Memento pattern is applicable when:

1. An object's internal state needs to be saved and restored without violating encapsulation.

2. The saved states need to be independent and accessible for later use.

3. There is a need for undo/redo functionality or checkpoints in the application.

Participants:

- Originator: The object whose state needs to be saved and restored. It creates a Memento object to capture its current state or uses a Memento object to restore its state.

- Memento: The object that stores the state of the Originator. It provides methods to access and restore the state, but only the Originator can access its contents.

- Caretaker: The object that requests the Originator to save or restore its state. It holds the Memento objects and is responsible for their safekeeping.

Collaborations: The Originator creates a Memento object to save its state or restores its state from a Memento object. The Caretaker can request the Originator to save its state by creating a Memento object and storing it. Later, the Caretaker can request the Originator to restore its state by providing a previously stored Memento object.

Consequences:

- The Memento pattern allows capturing and restoring an object's internal state without exposing it directly, preserving encapsulation.

- It enables the implementation of undo/redo functionality and checkpoints in an application.

- The pattern promotes separation of concerns by separating state management from the originator object.

- However, storing and managing a large number of Memento objects can lead to increased memory usage.

Implementation:

- Implement the Memento pattern by defining an Originator class that creates Memento objects and uses them to save and restore its state.

- The Memento class stores the state of the Originator and provides methods to access and restore it, which can be accessed only by the Originator.

- The Caretaker class requests the Originator to save or restore its state using Memento objects. It is responsible for the storage and management of Memento objects.

Known Uses: The Memento pattern is commonly used in applications that require undo/redo functionality, such as text editors, graphic editors, and software development environments. It is also used in systems that need checkpoints or state snapshots, such as game engines and database systems.

Related patterns: The Memento pattern is related to other patterns that deal with capturing and restoring object states, such as the Command pattern, which can be used to encapsulate and parameterize state-changing operations. The Memento pattern can be combined with the Command pattern to provide a more flexible and granular approach to state management and undo/redo functionality.

Pattern Name: Visitor (Behavioural)

Intent: The Visitor design pattern allows adding new operations to a set of classes without modifying their structure. It separates the algorithms from the objects on which they operate, promoting extensibility and flexibility in code maintenance.

Motivation: In object-oriented systems, it is often necessary to perform different operations on a set of related classes. However, adding new operations directly to each class can lead to a proliferation of methods and a violation of the open-closed principle. The Visitor pattern addresses this by defining a separate visitor object that encapsulates the operations to be performed on the classes. This way, new operations can be added by introducing new visitor classes, without modifying the existing class hierarchy.

Applicability: The Visitor pattern is applicable when:

1. There is a set of related classes that need to have different operations performed on them.

2. New operations need to be added to the classes without modifying their structure.

3. The behavior of the operations may vary depending on the class being visited.

Participants:

- Visitor: The interface or abstract class that declares the visit methods for each class in the object structure. Each visit method corresponds to an operation that can be performed on the visited class.

- ConcreteVisitor: The concrete implementation of the Visitor interface. It provides the specific implementation for each visit method, defining the behavior for each operation on the visited classes.

- Element: The interface or abstract class that declares the accept method, which accepts a visitor object. This method allows the visitor to access the element and perform the necessary operation.

- ConcreteElement: The concrete implementation of the Element interface. It implements the accept method and defines how the visitor accesses the element to perform the operation.

- ObjectStructure: The collection or structure that holds a set of elements to be visited. It provides a way to iterate over the elements and accepts visitor objects to perform operations on them.

Collaborations: The client creates a visitor object and passes it to the elements in the object structure. Each element accepts the visitor and calls the appropriate visit method. The visitor object performs the operation on the element based on its specific implementation of the visit method.

Consequences:

- The Visitor pattern separates the operations from the classes, promoting extensibility by allowing new operations to be added without modifying the existing classes.

- It centralizes the behavior in visitor classes, simplifying the code structure and reducing duplication.

- The pattern supports the open-closed principle by allowing new visitors to be added without modifying the existing elements.

- However, introducing new elements to the object structure requires modifying the visitor interface and all its implementations.

Implementation:

- Implement the Visitor pattern by defining a Visitor interface or abstract class that declares the visit methods for each class in the object structure.

- ConcreteVisitor classes implement the Visitor interface and provide the specific implementation for each visit method.

- The Element interface or abstract class declares the accept method, which accepts a visitor object.

- ConcreteElement classes implement the Element interface and define how the visitor accesses the element to perform the operation.

- The ObjectStructure class holds the elements and provides a way to iterate over them. It accepts visitor objects and calls the accept method on each element.

Known Uses: The Visitor pattern is commonly used in situations where a set of related classes needs to have different operations performed on them, such as in abstract syntax tree (AST) processing, compilers, interpreters, and code analysis tools. It is also used in frameworks that provide plugin architectures, where new operations can be added through visitor classes.

Related patterns: The Visitor pattern is related to other patterns that involve processing elements in a structure, such as the Iterator pattern, which can be used to traverse the elements in the structure, and the Composite pattern, which can be used to build hierarchical structures of elements. The Visitor pattern can be combined with these patterns