Design Patterns

Contents

1 Creational Patterns

notes on structural

1.1 Factory Method

Definition

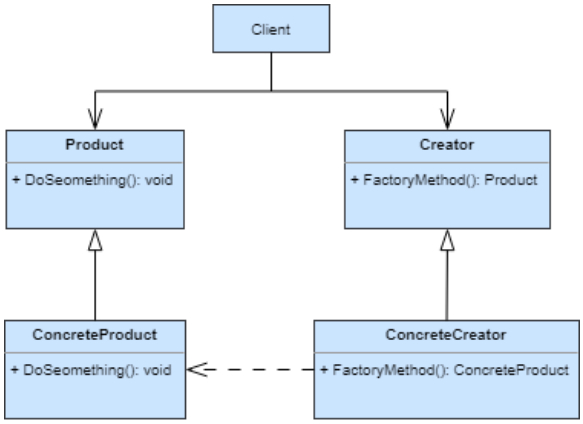
- It is a creational design pattern.
- Defines an interface for creating an object, but let concrete creator subclasses decide which product class to instantiate.
- Factory Method lets a client class defer instantiation to creator subclasses.
- Also known as the Virtual Constructor

Applicability

- **Encapsulate object creation:** If complex object creation process or if the process may vary based on conditions.
- **Decouple client code from concrete classes:** Abstract away the specific implementation details of the concrete classes from the client code.
- **Support multiple product variations:** Different variations of a product or if new types of products may be introduced in the future.
- **Support customization or configuration:** Encapsulate configuration logic, allowing clients to customize the creation process by providing parameters or configuration options to the factory method.

Structure

- **Product:** defines the interface of objects the factory method creates.
- **ConcreteProduct:** implements the product interface.
- **Creator:** declares the factory method, which returns an object of type Product.
- **ConcreteCreator:** overrides the factory method to return an instance of a Concrete Product.



Mechanism

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct..

Consequences

Advantages

- **Encapsulation:** hides object creation from the client making the code less dependent on specific implementations.
- **Decoupling:** separates object creation logic from the client code allowing for changes to the creation process with no modifications to client code.
- **Extensibility:** allows new product types without change in the client code. Create a new Concrete Creator class and implement the factory method to produce the new product class.
- **Code Reusability:** reused in different parts of the application where object creation is needed. This promotes centralizing and reusing object creation logic
- **Testability:** simplifies unit testing by allowing you to mock or stub out product creation during tests. Test different product implementations without relying on actual object creation.

Disadvantages

- **Increased Complexity:** It introduces additional interfaces and claases and thus adding a layer of abstraction that can make the code more complex to understand and maintain.
- **Overhead:** The use of polymorphism and dynamic binding can slightly impact performance, although this is often negligible in most applications.
- **Tight Coupling Within Product Hierarchies:** Concrete Creators are still tightly coupled to their corresponding Concrete Products. Changes to one often necessitate changes to the other.
- **Dependency on Concrete Subclasses:** The client code still depends on the abstract Creator class, requiring knowledge of its concrete subclasses to make correct factory method calls
- **Over-engineering:** potential for overuse in the application. Simple object creation can often be handled directly without the need for a factory.
- **Testing Challenges:** Testing the factory logic itself can be more complex as a result of many different concreate classes.

Implementation

Two major varieties

- When the Creator class is an abstract class and does not provide an implementation for the factory method it declares. This requires subclasses to define an implementation, because there's no reasonable default.
- When the Creator is a concrete class and provides a default implementation for the factory method that subclasses can override is appropriate. This uses inheritance.

Parameterized factory methods

- The factory method takes a parameter that identifies the kind of object to create. All these objects will share the same object interface.
- Overriding a parameterized factory method lets you easily extend or change the products that a Creator produces by introducing new identifiers for new kinds of products, or you can associate existing identifiers with different products.

Naming conventions

- Use naming conventions that make it clear you're using factory methods. Use "DoMakeClass()" as a naming for the factory method.

1.2 Abstract Factory

Definition

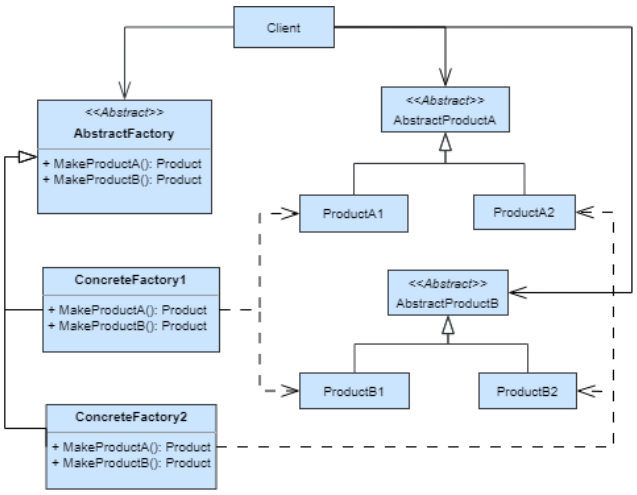
- It is a creational design pattern.
- It is a way of organizing how you create groups of things that are related to each other.
- Provide an interface for creating families of related objects without specifying their concrete classes.
- Similar to Factory Pattern and is considered as another layer of abstraction over factory pattern.
- Also known as Kit.

Applicability

- **Multiple families of related products:** when a system needs to be configured with multiple families of related products, and ensure that the products from one family are compatible with the products from another family.
- **Flexibility and extensibility:** allow for variations or extensions in the products or their families by providing a way to introduce new product variants without modifying existing client code.
- **Encapsulation of creation logic:** The pattern encapsulates the creation of objects, making it easier to change or extend the creation process without affecting client code.
- **Consistency across product families:** enforce consistency among the products created by different factories by maintaining a uniform interface.

Structure

- **AbstractFactory:** declares an interface for operations that create abstract product objects.
- **ConcreteFactory:** implements the operations to create concrete product objects.
- **AbstractProduct:** declares an interface for a type of product object.
- **ConcreteProduct:** defines a product object to be created by the corresponding concrete factory. Implements the AbstractProduct interface.
- **Client:** uses only interfaces declared by AbstractFactory and AbstractProduct classes.



Mechanism

- The client requests objects to be created by the AbstractFactory which defers creation of product objects to its ConcreteFactory subclass.
- Concrete factory creates product objects having a particular implementation and returns them to the client.

Consequences

Advantages

- **Isolation of concrete classes:** it isolates clients from implementation classes and manipulate these instances through their abstract interfaces. ConcreateProduct class names are isolated in the implementation of the concrete factory; they do not appear in client code.
- **Exchanging Product Families easily:** The ConcreteFactory class appears only once in an application, where it's instantiated, making it easy to modify if rquired. Since an AbstractFactory creates a complete family of products, the whole product family changes at once.
- **Promoting consistency among products:** When Product objects in a family are designed to work together, consistency in properties and functionality is enforced.

Disadvantages

- **Complexity:** AbstractFactory can introduce additional complexity to the codebase with multiple layers of abstraction.
- **Rigidity with New Product Types:** If product speficiation changes, modifications to concrete factories and the AbstractFactory interface are both required, impacting existing code
- **Increased Number of Classes:** Introduction of more product families require more abstract factories in the system.
- **Differentiation of classes:** All products are returned to the client with the same abstract interface. The client will not be able to differentiate between the class of a product.
- **Over-engineering:** The Abstract Factory pattern may be overkill for smaller, less complex systems where the overhead of defining abstract factories and products outweighs the benefits of the pattern.

Implementation

Factories as singletons

- Best implemented as a singleton as an application usually needs only one instance of a ConcreteFactory per product family.

Creating the products

- AbstractFactory declares an interface for creating products and the ConcreteProduct subclasses creates them by defining a factory method for each product.
- A concrete factory will specify its products by overriding the factory method for each. this requires a new concrete factory subclass for each product family, even if the product families differ only slightly.

Defining extensible factories

- AbstractFactory defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it.
- Adding a parameter (e.g. enum) to the AbstractFactory, to indicate the kind of object to be created reduces the method to a single "Make" operation.

1.3 Prototype

Definition

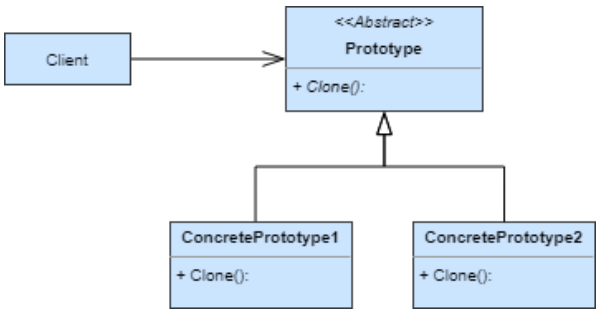
- It is a creational pattern.
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- enables the creation of new objects by copying an existing object.

Applicability

- **Creating Objects is Costly:** When creating objects is more expensive or complex than copying existing ones.
- **Variations of Objects:** When the system requires a variety of objects with slight variations.
- **Dynamic Configuration:** When the system requires dynamic configuration and creation objects with different configurations at runtime.
- **Building Similar Factories:** to avoid building a class hierarchy of factories that parallels the class hierarchy of products.

Structure

- **Prototype:** declares an interface for cloning itself.
- **ConcretePrototype:** implements an operation clone() for cloning itself.
- **Client:** creates a new object by asking a prototype to clone itself.



Mechanism

- A client asks a prototype to clone itself.

Consequences

Advantages

- **Efficient Object Creation:** It allows you to create new objects by copying existing ones, which can be faster and more efficinet than creating objects from scratch or if creation involves significant resources, such as database calls.
- **Flexibility:** Flexible way to create objects with different configurations or states reducing the need for complex initialization than compared to the other creational patterns (e.g builder).
- **Reduces Code Duplication:** Instead of creating multiple classes for each variation, prototypes can be created and cloned with modifications.
- **Adding and removing products at run-time:** Prototypes let you incorporate a new concrete product class into a system by registering a prototype instance with the client at run-time.
- **Specifying new objects by varying structure:** build composite objects from parts and subparts by adding subcircuitry as a prototype to the palette of available circuit elements. For circuits with different structures to be prototypes, the composite circuit object must implement Clone() as a deep copy.

Disadvantages

- **Complexity:** Implementing the Prototye Pattern can be complex, especially when dealing with deep copying of complex objects.
- **Not Suitable for All Scenarios:** It may not be suitable for all scenarios, especially when objects have circular references or complex interdependencies.
- **Memory Usage:** If not managed properly, cloning objects can lead to increased memory usage, potentially causing memory leaks.

Implementation

Using a prototype manager

- The prototype manager keeps a registry of available prototypes. It has operations for registering a prototype under a key and for unregistering it
- Clients will store and retrieve prototypes them from the registry by a key-value. A client will ask the registry for a prototype before cloning it.

Implementing the Clone operation

- Cloning prototypes with complex structures requires a deep copy, because the clone and the original must be independent. That is, instance variables are not shared between copies.

Initializing clones

- If the client requires variations of copies, then then use initialization parameters as arguments in the initialization operations to set the clone's internal state accordingly.

1.4 Builder

Definition

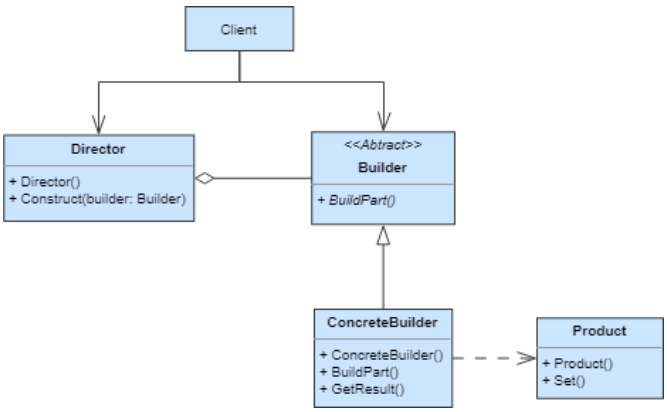
- The Builder Design Pattern is a creational pattern
- It allows the construction of a product in a step-by-step fashion, where the construction process can vary based on the type of product being built.
- The pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations

Applicability

- **Complex Object Construction:** Provide a clear separation between the construction process and the actual representation of the object.
- **Step-by-Step Construction:** Construction of an object involves a step-by-step process where different options need to be set at different stages
- **Remove Constructors with many parameters:** Replace constructors with a large number of parameters to cater for different configurations of objects.
- **Immutable Objects:** Create immutable objects, and the pattern allows you to construct the object gradually before making it immutable
- **Configurable Object Creation:** Ccreate objects with different configurations, and a more flexible and readable way to specify these configurations.
- **Common Interface for Multiple Representations:** Provide a common interface for constructing different representations of an object

Structure

- **Builder:** specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder:** constructs and assembles parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for retrieving the product.
- **Director:** constructs an object using the Builder interface.
- **Product:** represents the complex object under construction. It includes classes that define the constituent parts, including interfaces for assembling the parts into the final result assembled.



Mechanism

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences

Advantages

- **Reusability:** While making the various representations of the products, we can use the same construction code for other representations as well.
- **Single Responsibility Principle:** We can separate out both the business logic as well as the complex construction code from each other
- **Construction of the object:** Here we construct our object step by step, defer construction steps or run steps recursively

Disadvantages

- **Code complexity increases:** The complexity of our code increases, because the builder pattern requires creating multiple new classes
- **Mutability:** It requires the builder class to be mutable
- **Initialization:** Data members of the class are not guaranteed to be initialized

Implementation

Assembly and construction interface

- Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A model where the results of construction requests are simply appended to the product is usually sufficient.

No abstract class for products

- the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class
- the client usually configures the director with the proper concrete builder, the client is in a position to know which concre te subclass o fBuilder is in use and can handle its products accordingly.

Empty methods as default in Builder

- Methods are defined as empty methods, letting clients override only the operations they're interested in.

1.5 Singleton

Definition

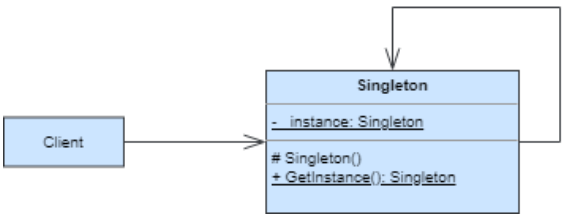
- It is a creational design pattern.
- Ensures a class only has one instance, and provide a global point of access to it.
- It is also called the antipattern.

Applicability

- **Services:** Services that should have a single instance, such as a data connections, logging, API service, or utility service.
- **Managing Shared Resources:** Localised point of management for shared resources like database connections, network connections, or thread pools.
- **Preventing Multiple Instantiations:** When instantiating multiple instances of a class would cause issues or inefficiencies.
- **Lazy Initialization:** If the instantiation of an object is resource-intensive and delay the creation until it is actually needed.
- **Preventing Cloning:** Prevent the cloning of an object, which could lead to multiple instances.

Structure

- **Singleton:** defines an Instance operation that lets clients access its unique instance. Instance is a class operation (i.e., a static member). may be responsible for creating its own unique instance.



Mechanism

- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences

Advantages

- **Controlled access to sole instance:** Since the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
- **Reduced name space:** The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
- **Permits refinement of operations and representation:** The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. The application can be configured with an instance of the class needed at run-time.
- **Permits a variable number of instances:** The pattern allows more than one instance of the Singleton class. Moreover, the same approach can be used to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- **More flexible than class operations:** A way to package a singleton's functionality is to use class operations (i.e, a static member) which is much easier to implement than other logic.

Disadvantages

- **Concurrency Issues:** If not implemented carefully, Singletons can introduce concurrency issues in multi-threaded applications. Synchronization mechanisms need to be used to ensure safe access to the Singleton instance, which can add complexity to the code.
- **Singleton Pattern Overuse:** Developers might overuse the Singleton pattern, leading to an abundance of Singleton instances in an application. This can defeat the purpose of the pattern and result in increased memory usage.
- **Initialization Overhead:** Lazy initialization, while often an advantage, can introduce some overhead when the Singleton instance is first accessed. If the initialization process is resource-intensive, it can affect the application's startup time.
- **Difficulties in Debugging:** Debugging a Singleton-based codebase can be challenging when issues related to the Singleton's state arise. It can be hard to trace the source of problems when multiple parts of the code may have modified the Singleton's data.
- **Limited Dependency Injection:** Using dependency injection and inversion of control becomes less straightforward when relying on Singleton instances. It may be challenging to inject alternative implementations or configurations because the Singleton instance is typically accessed globally.

Implementation

Ensuring a unique instance

- The Singleton pattern makes the sole instance a normal instance of a class written so only one instance can ever be created.
- This done by encapsulating the creation process in a class operation that has access to the variable that holds the unique instance, and it ensures the variable is initialized with a single instance before returning its value. Impletmented though the Instance() method.

Subclassing the Singleton class

- If there is more than one subclass for a singleton or multiple singletons required, then registering all these by name in a common registry that maps between string names and singletons is ideal.
- This requires a common interface that all Singleton must implement and the register can store these in a map or library.
- The client consults the registry asking for the singleton by name and the registry returns it or creates it.
- The register is a singleton itself and the client interacts with the register to add and request singletons.

2 Structural Patterns

notes on structural

2.1 Adapter

Definition

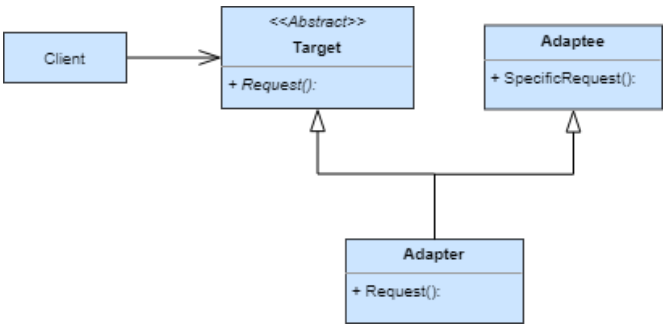
- It is a structural design pattern.
- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also known as the wrapprer method.

Applicability

- **Integration of Existing Code:** use an existing class, and its interface does not match what is needed.
- **Reuse of Existing Functionality:** Reuse classes or components that provide valuable functionality but do not conform to the desired interface.
- **Interoperability:** Make different systems or components work together, especially when they have different interfaces.
- **Object adapter only:** Need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one (multiple adapters). An object adapter can adapt the interface of its parent class.

Structure

- **Target:** defines the domain-specific interface that Client uses.
- **Client:** collaborates with objects conforming to the Target interface
- **Adaptee:** defines an existing interface that needs adapting.
- **Adapter:** adapts the interface of the Adaptee to the Target interface.



Mechanism

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

Advantages

- **Compatibility:** Allows integration of new and old systems or components with different interfaces.
- **Reusability:** Existing classes can be reused without modification, reducing the risk of introducing bugs.
- **Flexibility:** New adapters can be easily added to adapt various classes to a common interface.
- **Maintainability:** Isolates changes to the adapter, making it easier to maintain and update systems.
- **Overide Operations:** lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- **Adaptability:** (For Object adapter only) lets a single Adapter work with many Adaptees. That is, the Adaptee itself and all of its subclasses. The Adapter can also add functionality to all Adaptees at once.

Disadvantages

- **Complexity:** Introducing adapters can add complexity to the codebase, especially when multiple adapters are used making codebase less intuitive and harder to understand.
- **Performance Overhead:** Adapters can introduce some performance overhead due to the additional method calls.
- **Not Always Suitable:** Some classes may be too different to adapt easily, making the Adapter Pattern impractical in certain situations.
- **Class Dependency:** adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- **Subclassing Problem:** (For Object adapter only) makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Implementation

Item

-

2.2 Bridge Method

Definition

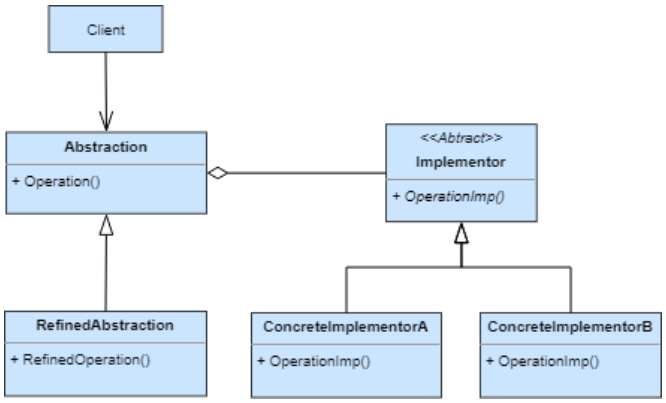
- Decouple an abstraction from its implementation so that the two can vary independently.
- Also known as Handle/Body.

Applicability

- **Decoupling:** you want to avoid a permanent binding between an abstraction and itsimplementation.
- **Extend class behavior:** both the abstractions and their implementations should be extensible by subclassing.
- **Item:** changes in the implementation of an abstraction should have no impact on clients.
- **Item:** you want to hide the implementation of anabstraction completely from clients.
- **Item:** you have a proliferation of classes or deep class hierarchies in need for splitting into two parts.
- **Item:** want to share an implementation among multiple objects and this fact should be hidden from the client.

Structure

- **Abstraction:** defines theabstraction's interface and maintains a reference to an object of type Implementor.
- **RefmedAbstraction:** Extends the interface defined by Abstraction.
- **Implementor:** defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface. The Implementor interface provides primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor:** implements the Implementor interface and define its concrete implementation.



Mechanism

- Abstraction forwards client requests to its Implementor object.

Consequences

Advantages

- **Scalability:** Provides a scalable way to manage and organise a growing number of abstractions and implementations in the system.
- **Decupling interface and implementation:** An implementation is not bound permanently to an interface allowing changes in one part of the system to have minimal impact on the other and implementations can be selected or switched at run-time.
- **Abstraction** Decoupling encourages layering that can lead to a better structured system. Clients have a consistent interface to work with only has to know about Abstraction and Implementor.
- **Maintainability:** Makes it easier to add new abstractions or implementations without modifying the existing code.
- **Implementation Interchangibility:** Allows different implementations to be interchangable with each other without altering the abstraction code. E.g. supporting multiple databases.
- **Extendibility:** Allows combining different abstractions and implementations and extend them independently without affecting each other.

Disadvantages

- **Overhead:** The Bridge pattern can introduce some overhead due to the need for multiple classes and interactions between them.
- **Increased Complexity:** Added complexity may make the code harder to understand, navigate, test and maintain.

Implementation

Item

-

2.3 Composite Method

Definition

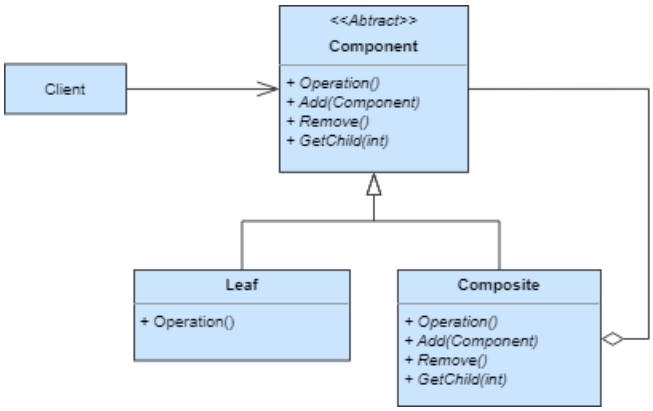
- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Applicability

- **Item:** you want to represent part-whole hierarchies of objects.
- **Item:** you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure

- **Component:** declares the interface for objects in the composition. Implements default behavior for the interface common to all classes, as appropriate. declares an interface for accessing and managing its child components
- **Leaf:** represents leaf objects in the compositi onA. leaf has n cohil dren. Defines behavior for primitive objects in the composition.
- **Composite:** defines behavior for components having children. stores child components. implements child-related operations in the Component interface.
- **Client:** Manipulates the object in the commposition through the Component's interface.



Mechanism

- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

Advantages

- **Hierarchical Structure:** defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so onrecursive ly.
- **Simplified Client Code:** Clients can work both with individual objects and composites without needing to know the difference.
- **Flexibility:** Allows the removal of objects in the hierarchy and the addition of new kinds of components. New composite or leaf subclasses work automaticallly with existing structure and client code.
- **Scalability:** Allows the creation of more complex structures by nesting composites within composites, making it a scalable solution for modeling part-whole hierarchies.
- **Code Reusability:** The pattern encourages the reuse of code. You can apply the same operations to both individual objects and composites, reducing duplication of code.

Disadvantages

- **Complex Implementation:** Implementing the Composite Pattern can be more complex compared to a non-composite approach.
- **Performance Overhead:** Traversing and performing operations on a composite structures can result in performance overhead when dealing with deep hierarchies as a result of multiple layers of objects.
- **Limited Type Safety:** Since the Composite Pattern involves a common interface for both leaf and composite objects, it can lead to a lack of type safety. It's possible to call methods on composite objects that are not applicable, leading to runtime errors. for developers who are not familiar with the pattern.
- **Extra Memory Usage:** Composite structures can consume additional memory due to the need to store references to child objects within composite objects. This can be a concern when dealing with large hierarchies.

Implementation

Item

-

2.4 Decorator Method

Definition

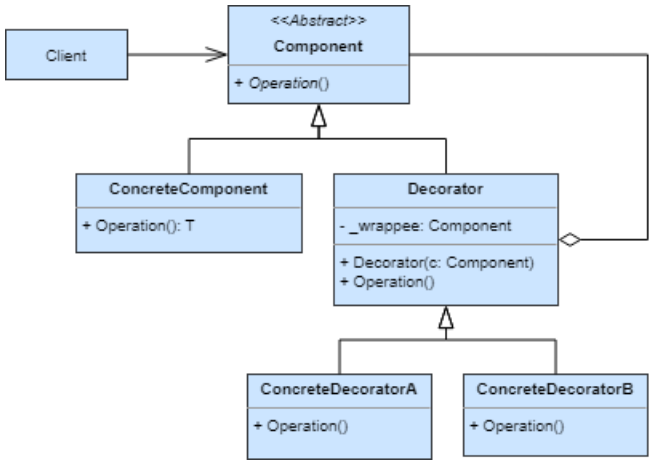
- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

Applicability

- **Dynamic Behavior Addition:** dynamically add or modify the behavior of an object at runtime.
- **Avoiding Subclass Explosion:** large number of combinations of functionalities that would otherwise lead to a combinatorial explosion of subclasses
- **Enhancing or Extending Functionality:** to extend the functionality of a class without modifying its existing code.
- **Maintaining Open/Closed Principle:** for classes to be open for extension but closed for modification.
- **Reusable and Composable Functionality:** create reusable and composable functionality that can be applied to different objects

Structure

- **Component:** defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent:** defines an object to which additional responsibilities can be attached.
- **Decorator:** maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator:** adds responsibilities to tchoemponent.



Mechanism

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

Advantages

- **Open-Closed Principle:** allows introduction of new functionality to an existing class without changing its source code.
- **Flexibility:** add or remove responsibilities from an object at runtime.
- **Reusable Code:** combine several behaviors by wrapping an object into multiple decorators.
- **Composition over Inheritance:** divide a monolithic class that implements many possible variants of behaviour into several smaller classes.

Disadvantages

- **Complexity:** It's hard to remove a specific wrapper from the wrappers stack.
- **Lots of little objects:** When using the Decorator pattern, you often end up with a large number of small, specialized decorator classes which may increase maintenance overhead.
- **Order of Decoration:** It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.

Implementation

Item

-

2.5 Facade

Definition

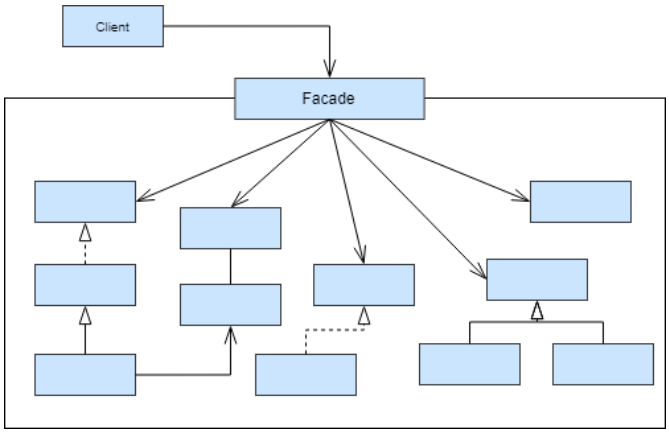
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability

- **Simple view:** Provide a simple interface to a complex subsystem for clients. Only clients needing more customizability will need to look beyond the facade.
- **Decoupling:** There are many dependencies between clients and the implementation classes of an abstraction. A Facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- **Laerying of a susbsytem:** Defines an entry point to each subsystem level. If subsystem are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure

- **Facade:** knows which subsystem classes are responsible for a request. delegates client requests to appropriate subsyste m objects
- **subsystem classes:** implement subsystem functionality. handle work assigned by the Facade object. have no knowledge of the facade; that is they keep no references to it.



Mechanism

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objec tsdirectl y.

Consequences

Advantages

- **Simplified Interface:** Provides a clear and concise interface to a complex system, making it easier for clients to use. This reduces the number of objects that clients deal with and making the subsystem easier to use.
- **Reduced Coupling:** Decouples clients from the underlying system, making them less dependent on its internal structure.
- **Reusable Code:** Promotes modularity and reusability of code components and facilitates independent development and testing of different parts of the system
- **Encapsulation:** Encapsulates the complex interactions within a subsystem, protecting clients from changes in its implementation.
- **Improved Maintainability:** Easier to change or extend the underlying system without affecting clients.

Disadvantages

- **Increased Complexity:** Introducing a facade layer adds an extra abstraction level, potentially increasing the overall complexity of the system.
- **Reduced Flexibility:** The facade acts as a single point of access to the underlying system limiting the flexibility for clients who need to bypass the facade or access specific functionalities hidden within the subsystem.
- **Over-engineering:** Larger complex systems can befinit from this pattern, however consider the cost-benefit and trade-off before implementing a facade for simplier system.
- **Potential Performance Overhead:** Adding an extra layer of indirection through the facade can introduce a slight performance overhead for performance-critical scenarios.
- **Bypassing the facade:** It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

Reducing client-subsystem coupling

-

Public versus private subsystem classes

-

2.6 Flyweight Method

Definition

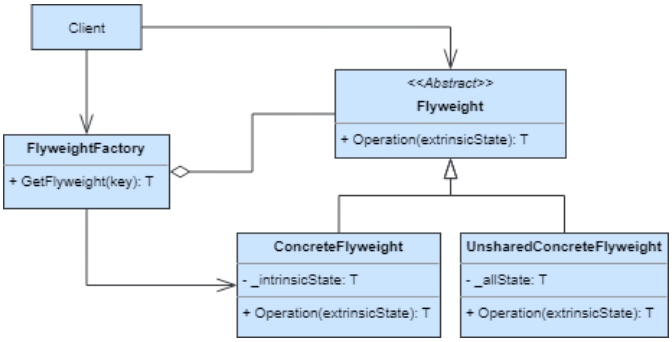
- It is a structural pattern.
- Use sharing to support large numbers of fine-grained objects efficiently.
- A flyweight is a shared object that can be used in multiple contexts simultaneously.
- The flyweight acts as an independent object in each context.

Applicability

- **Many objects:** An application uses a large number of objects.
- **Object Cost:** Storage costs are high because of the quantity of objects.
- **Extrinsic State:** Most object state can be made extrinsic.
- **Object Replacement:** Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- **Common Interface:** The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Structure

- **Flyweight:** declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight:** implements the Flyweight interface and adds storage for intrinsic state. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight:** not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. UnsharedConcreteFlyweight objects usually have ConcreteFlyweight objects as children at some level in the flyweight object structure.
- **FlyweightFactory:** creates and manages flyweight objects. Ensures that flyweights are shared properly and when a client requests a flyweig ht, the FlyweightFactory supplies an existing instance or creates one.
- **Client:** maintains a reference to flyweight(s). Computes or stores the extrinsic state of flyweight(s).



Mechanism

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.
- Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state varies with the flyweight's context and is stored or computed by Client objects.
- Clients pass this state to the flyweight when they invoke its operations.
- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects from the FlyweightFactory to ensure they are shared properly.

Consequences

Advantages

- **Memory Efficiency:** Reduces memory usage by sharing the flyeight object, rather than duplicating them across multiple instances.
- **Performance Improvement:** Sharing common state across multiple objects leads to performance improvements where creating and managing a large number of objects would be expensive.
- **Decoupling:** promotes the separation of intrinsic and extrinsic states.
- **Improved Maintainability:** Changes to the shared state is centralized and applied to all instances as modifications are concentrated in a smaller set of objects.
- **Enhanced Scalability:** Allows systems to scale more effectively by reducing memory and processing overhead associated with creating and managing numerous objects.

Disadvantages

- **Complexity:** Adds complexity, especially in scenarios where the separation of intrinsic and extrinsic state is complex.
- **Overhead:** Run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.
- **Impact on Identity:** The pattern blurs the identity of objects by sharing their internal state. This can be problematic if the identity of individual objects is important in the application logic.
- **Design Rigidity:** Changes to the shared state may require modifications to multiple objects.

Implementation

Removing extrinsic state

- Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.
- extrinsic state can be computed from a separate object structure, one with far smaller storage requirements

Managing shared objects

- Since clients shouldn't instantiate flywight objects directly, the fleight weight factory is responsible to manage a particular flyweight.
- FlyweightFactory objects often use a store to let clients look up flyweights of interest. E.g. a table.
- It must also implement some form of reference countig or garbage collection to reclaim a flyweight's storage when it's no longer needed.

2.7 Proxy

Definition

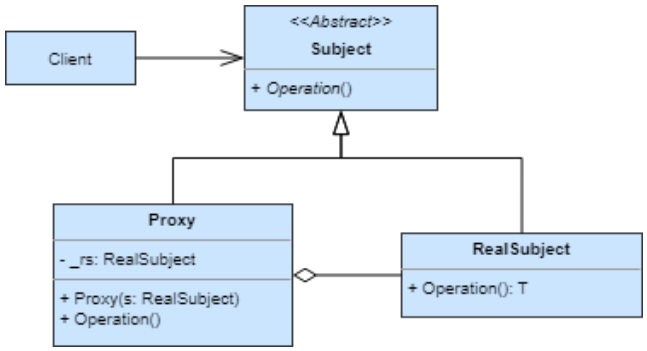
- It is a structural design pattern.
- Provide a surrogate or placeholder for another object to control access to it.

Applicability

- **Deferred Object Creation:** (virtual proxy) postpone the creation of a resource-intensive object until it is required.
- **Access Control and Permissions:** (protection proxy) control and manage security access to an object.
- **Resource Optimization:** optimize the utilization of resources, such as caching results or storing previously fetched data.
- **Remote Object Interaction:** (remote proxy) working in distributed systems where interaction between objects of different addresses or systems occurs.

Structure

- **Proxy:** maintains a reference that lets the proxy access the real subject. Proxy may also refer to a Subject if the RealSubject and Subject interfaces are the same. Provides an interface identical to Subject's so that a proxy can by substituted for the real subject and controls access to the real subject. Other responsabilities depend of the kind of proxy. *Remote proxies* responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space. *Virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. *protection proxies* check that the caller has the access permissions required to perform a request
- **Subject:** defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject:** defines the real object that the proxy represents.



Mechanism

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

Advantages

- **Improved Performance:** improves the performance by controlling the loading and initialization of objects.
- **Separation of Concerns:** promotes a decoupling by isolating the proxy-specific functionality from the real object's implementation making the code modular.
- **Improved Testing:** facilitates testing by isolating the unit under test from its dependencies, making it easier to control and verify behavior.
- **Dynamic Behavior:** Proxies can dynamically alter the behavior of the object allowing additional functionalities, without modifying the original implementation.
- **Reduced Resource Usage:** Postpone the creation of expensive objects until they are needed reducing memory usage.
- **Enhanced Security:** Allows the addition of a layer of security checks before allowing access to the real object.

Disadvantages

- **Complexity:** create a layer of complexity in the program when dealing with multiple proxies.
- **Overhead:** extra overhead introduced if there is significant logic in the proxy class, such as logging, monitoring, or security checks.

Implementation

Item

-

3 Behavioural

notes on structural

3.1 Chain of Responsibility

Definition

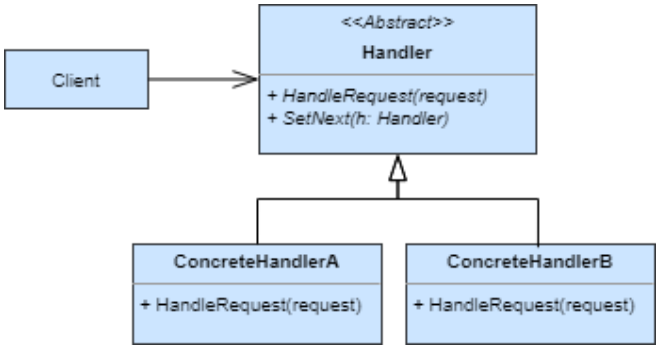
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects and pass the request along the chain until an object handles it.

Applicability

- **Multi class handling:** multiple objects may handle a request, and the handler isn't known from the previous. The handler should be ascertained automatically.
- **Handler declaration:** Issue a request to one of several objects without specifying the handler.
- **Separate sender and receiver:** the set of objects that can handle a request should be specified dynamically.

Structure

- **Handler:** defines an interface for handling requests. It can also implements the successor link.
- **ConcreteHandler:** handles requests it is responsible for. can access itssuccessor. if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- **Client:** initiates the request to a ConcreteHandler object on the chain.



Mechanism

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

Consequences

Advantages

- **Reduced coupling:** Enables sending a request to a series of possible recipients without having to worry about which object will handle the request.
- **Flexibility and Extensibility:** New handlers can be easily added or existing ones can be modified without affecting the client code.
- **Dynamic Order of Handling:** The sequence and order of handling requests can be changed dynamically during runtime, which allows adjustment of the processing logic distributing responsibilities among objects.
- **Enhanced Maintainability:** Each handler performs a specific type of processing, which making maintaining and modifying the individual components easier without impacting the overall system.

Disadvantages

- **Performance Overhead:** The request will go through several handlers in the chain if it is lengthy and complicated, which could cause performance overhead. The processing logic of each handler has an effect on the system's overall performance
- **Complexity in Debugging:** Can make debugging more difficult as tracking the progression of a request and determining which handler is in charge of handling it can be difficult.
- **Receipt isn't guaranteed:** Since a request has no explicit receiver, there's no guarantee it'll be handled—the request can fall off the end of the chain without being handled or if the chain is not configured properly.

Implementation

Item

-

3.2 Command Method

Definition

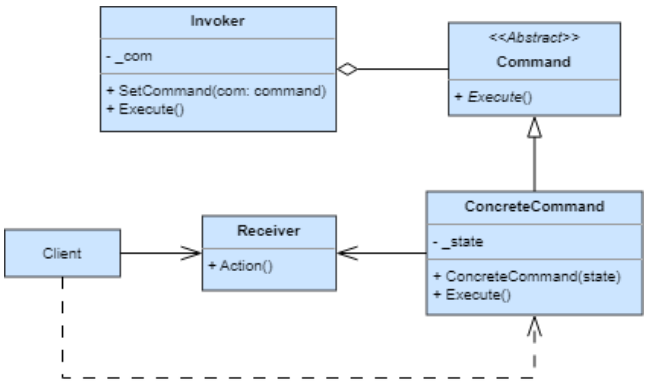
- It is a behavioural design pattern.
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Also known as Action and Transaction.

Applicability

- **Decoupling:** Decouple the sender (requester) of a request from the object that performs the request.
- **Undo/Redo Functionality is Required:** Support undo and redo operations in your application. Each command can encapsulate an operation and its inverse, making it easy to undo or redo actions.
- **Support for Queues:** specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
- **Support for Logging:** changes so that they can be reapplied in case of a system crash.
- **Dynamic Configuration:** Ability to dynamically configure and assemble commands at runtime.

Structure

- **Command:** declares an interface for executing an operation an operation
- **ConcreteCommand:** defines a binding between a Receiver object and action. Implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client:** creates a ConcreteCommand object and sets its receiver.
- **Invoker:** asks the command to carry out the request.
- **Receiver:** knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.



Mechanism

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

Consequences

Advantages

- **Decoupling of Sender and Receiver:** Command decouples the object that invokes the operation from the one that knows how to perform it.
- **Command Queueing:** Commands can be queued allowing for undo and redo operations.
- **Extensibility:** Commands are first-class objects. They can be manipulated and extended like any other object with no effect to existing code.
- **Support for Composite Commands:** Creating composite commands allows the implementation of complex operations required when executing a sequence of actions as a single command.
- **Encapsulation of State:** Commands can encapsulate the state required for their execution ensuring that a command has all the information it needs to perform its action

Disadvantages

- **Complexity:** can lead to a more extensive class hierarchy when dealing with various types of commands and receivers.
- **Memory Usage:** there can be increased memory overhead when dealing with a large number of commands.

Implementation

Item

-

3.3 Interpreter Method

Definition

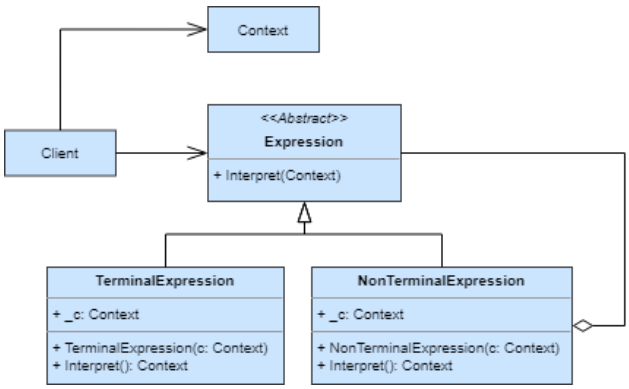
- It is a behavioural design pattern.
- Given a programming language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Applicability

- **Domain-specific languages:** to interpret and execute expressions or commands in a domain-specific language.
- **Grammar to interpret:** requires well-defined grammar for expressions or commands that need to be interpreted.
- **Adding new operations is frequent:** requires the addition of new operations or commands.
- **Avoid complex grammar parsers:** If building and maintaining complex grammar parsers seems daunting or unnecessary for the application.

Structure

- **Expression:** declares an abstra ct Interpret operation that is common to allnodes in the abstract syntax tree.
- **TerminalExpression:** implements anInterpret operation associate dwith terminal symbo ls in the grammar. an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression:** one such class is required for every rule in the grammar. maintains instance variables of type AbstractExpression for each of the symbols RI through Rn. implements an Interpret operation fo r nonterminal symbols in the gramma r. Interpret typical ly calls itself recursively on the variables representing RI through Jin.
- **Context:** contains information that's globa l to theinterpreter.
- **Client:** builds an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes. invokes the Interpret operation.



Mechanism

- The client builds (or is given) the sentence as an abstract syntax tree ofNonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

Advantages

- **Change and extend the grammar:** it is easy the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar.
- **Implementing the grammar:** Classes defining nodes in the abstract syntax tree have similar implementations and are easy to write, and creation can be automated with a compiler or parser generator.
- **Flexibility:** pattern makes it easier to evaluate an expression in a new way.

Disadvantages

- **Complex grammars arehard tomaintain:** the pattern defines at least one class for every rule in the grammar and thus one containing many rules can be hard to manage and maintain.

Implementation

Creating theabstract syntax tree

-

Defining the Interpret operation

-

Sharing terminal symbols with theFlyweight pattern

-

3.4 Iterator

Definition

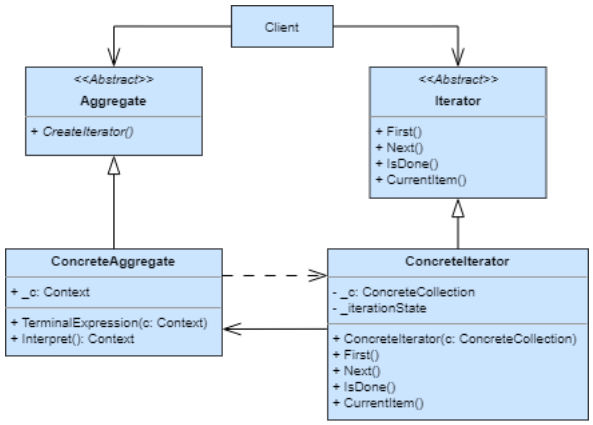
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Also known as the Cursor.

Applicability

- **Need for sequential access:** Access elements of a collection to iterate over different types of collections.
- **Decoupling iteration logic:** Decouple the iteration logic from the collection.
- **Support for multiple iterators:** Need to support multiple iterators over the same collection. Each iterator maintains its own iteration state, allowing multiple iterations to occur concurrently.
- **Simplifying client code:** Simplify client code that iterates over a collection. Clients only need to interact with the iterator interface, abstracting away the complexity of the collection's internal structure.

Structure

- **Iterator:** defines an interface for accessing and traversing elements.
- **ConcreteIterator:** implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.
- **Aggregate:** defines an interfa ce fo r creating an Iterator object.
- **ConcreteAggregate:** implements the Iterator creation interface to return an instance of the proper ConcreteIterator.



Mechanism

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

Advantages

- **Supports variations in the traversal of an aggregate:** Complex aggregates may be traversed in many ways. Iterators make it easy to change the traversal algorithm by replacing the iterator instance with a different one and supports new traversals by Iterator subclasses.
- **Iterators simplify the Aggregate interface:** Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
- **More than one traversal can be pending on an aggregate:** An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.
- **Open/Closed Principle:** You can implement new types of collections and iterators and pass them to existing code without breaking anything.

Disadvantages

- **Complexity:** Applying the pattern can be an overkill if your app only works with simple collections.
- **Over-engineering:** Using an iterator may be less efficient than going through elements of some specialized collections directly.

Implementation

Item

-

Relationships

- Use Iterators to traverse Composite trees.
- Use Factory Method along with Iterator to let collection subclasses return different types of iterators that are compatible with the collections.
- Use Memento along with Iterator to capture the current iteration state and roll it back if necessary.
- Use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.

3.5 Mediator

Definition

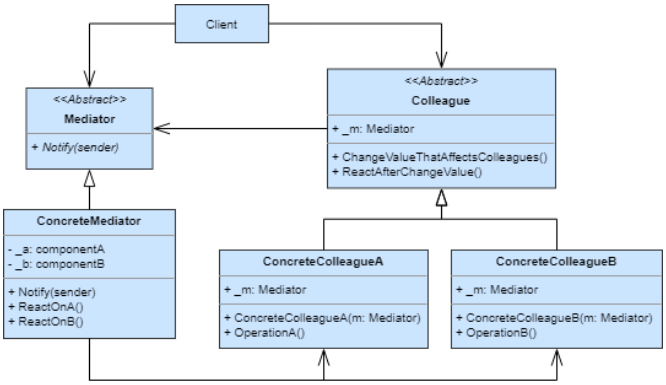
- It is a behavioural design pattern.
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Applicability

- **Complex Communication:** a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- **Loose Coupling:** Promote loose coupling between objects, allowing them to interact without knowing the details of each other's implementations.
- **Centralized Control:** Centralized mechanism to coordinate and control the interactions between objects, ensuring a more organized and maintainable system.
- **Changes in Behaviour:** Anticipate changes in the behaviour of components and encapsulation these changes within the mediator will prevents widespread modifications.
- **Enhanced Reusability:** Reuse individual components in different contexts without altering their internal logic or communication patterns.

Structure

- **Mediator:** defines an interface for communicating with Colleague objects.
- **ConcreteMediator:** implements cooperative behavior by coordinating Colleague objects. knows and maintains its colleagues.
- **Colleague classes:** each Colleague class knows its Mediator object. each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.



Mechanism

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences

Advantages

- **Single Responsibility Principle:** Extracts the communications between various components into a single place, making it easier to comprehend and maintain.
- **Open/Closed Principle:** Allows for the introduction of new mediators without having to change the actual components.
- **limits subclassing:** localizes behaviour that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only.
- **It decouples colleagues:** promotes loose coupling between colleagues. Allows varying and reusing Colleague and Mediator classes independently.
- **It abstracts how objects cooperate:** Allows encapsulation in an object lets you focus on how objects interact apart from their individual behaviour. That can help clarify how objects interact in a system.

Disadvantages

- **God object:** Over time a mediator can evolve into a God Object.

Implementation

Item

-

3.6 Memento Method

Definition

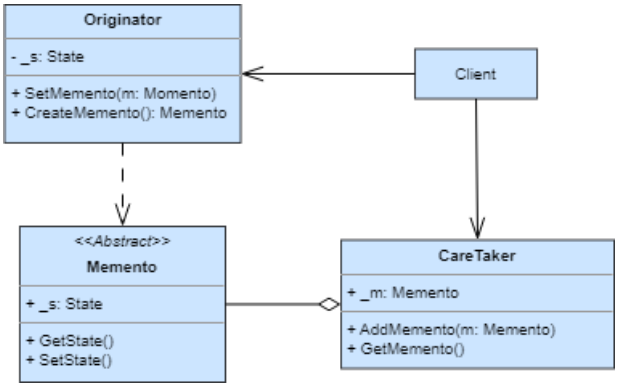
- It is a beahvioural design pattern.
- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Also known as Token.

Applicability

- **Undo functionality:** implement an undo feature in your application that allows users to revert changes made to an object's state.
- **Snapshotting:** save the state of an object at various points in time to support features like versioning or checkpoints.
- **Transaction rollback:** rollback changes to an object's state in case of errors or exceptions, such as in database transactions.
- **Caching:** cache the state of an object to improve performance or reduce redundant computations.
- **Encapsulation:** a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure

- **Memento:** stores internal state of the Originator object. The memento stores the originator's internal state at its originator's discretion. It protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento as it can only pass the memento to other objects. Originator sees a wide interface, that can access all the data necessary to restore itself to its previous state.
- **Originator:** creates a memento containing a snapshot of its current internal state. uses the memento to rstore its internal state.
- **Caretaker:** is responsible for the momento's safekeeping. Never operates on or examines the conents of a momento.



Mechanism

- A caretaker requests a memento from an originator, holds it for a time, and passe ist back to the originator. Someti mes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state. Memento s a re passive. Only the originator that created a memen towill assign or retrieve its state.

Consequences

Advantages

- **Encapsulation boundaries:** produces snapshots of the object's state without violating its encapsulation.
- **Simplifies Originator:** simplifies the originator's code by letting the caretaker maintain the history of the originator's state

Disadvantages

- **mementos might beexpensive:** The app might consume lots of RAM if clients create mementos too often or if Originator must copy large amounts of information to store in the memento.
- **Defining narrow and wide interfaces:** Most dynamic programming languages, such as PHP, Python and JavaScript, cannot guarantee that the state within the memento stays untouched.
- **Costs in caring for mementos.:** Caretakers tracks all the originator's lifecycle to be able to destroy obsolete mementos encuring large storage costs.

Implementation

Item

-

3.7 Observer

Definition

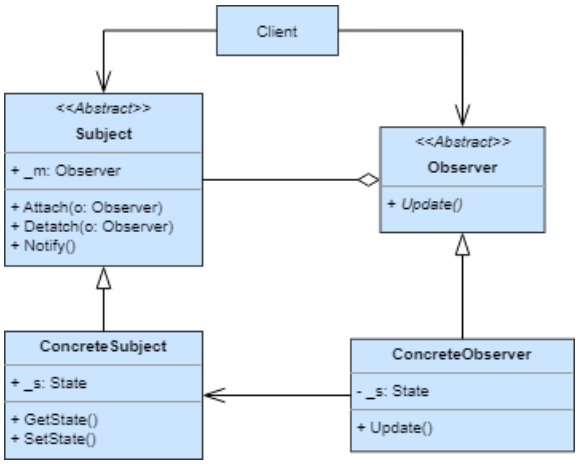
- It is a behavioural design pattern.
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Also known as Dependents and Publish-Subscribe.

Applicability

- **One-to-Many Dependence:** when there is a one-to-many relationship between objects, and changes in one object should notify multiple dependent objects
- **Decoupling:** to achieve loose coupling the subject (publisher) and observers (subscribers) to interact without being aware of each other's specific details.
- **Dynamic Composition:** support dynamic composition of objects with runtime registration and deregistration of observers.
- **Event Handling:** in event handling systems. events occur in a system, observers (listeners) can react to those events without requiring the source of the events to have explicit knowledge of the observers.

Structure

- **Subject:** knows its observers. Any number of Observer objects may observe a subject. provides an interface for attach ingand detaching Observer objects.
- **Observer:** defines an updating interface for objects that should be notified ofchanges in a subject.
- **ConcreteSubject:** stores state of interest toConcreteObserver objects. sends a notification to its observers when its state changes.
- **ConcreteObserver:** maintains a reference to a ConcreteSubject object, stores state that should stay consistent with the subject's and implements the Observer updating interface to keep its state consistent with the subject's.



Mechanism

- The ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.
- ConcreteObserver uses this information to reconcile its state with that of the subject.
- The Observer object that initiates the change request postpones its update until it gets a notification from the subject.
- Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely.

Consequences

Advantages

- **Decoupling:** subjects only know the list of observers, each conforming to the simple interface of the abstract Observer class. Subjects do not know the concrete class of any observer thus the coupling between these is abstract and minimal and they can belong to different layers of abstraction in a system.
- **Broadcast communication:** The notification is broadcasted to all interested objects that subscribed to it and it is up to the observer to handle or ignore the notifcation.
- **Open/Closed Principle.:** introduce new subscriber classes without having to change the publisher's code.
- **Relations:** establish relations between objects at runtime.

Disadvantages

- **Notification order:** Subscribers are notified in random order.
- **Unexpected updates:** A small change on the subject may cause a cascade of updates to observers and their dependent objects. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Implementation

Item

-

3.8 State Method

Definition

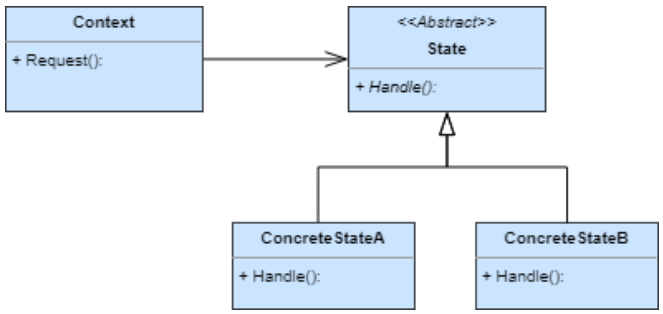
- It is a behavioural design pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Also known as Objects for States.

Applicability

- **Multiple states:** If your object exists in several states and each state dictates unique behaviors.
- **Conditional logic:** When conditional statements become extensive and complex within your object, it helps to organize and separate state-specific behavior into individual classes.
- **State changes:** If an object transitions between states frequently, the State pattern provides a clear mechanism for managing these transitions and their associated actions
- **Extendible:** highly extendability as it allows for the creation of new classes without affecting existing ones.

Structure

- **Context:** defines the interface of interest to clients. It maintains an instance of a ConcreteState subclass that defines the current state.
- **State:** defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses:** each subclass implements behaviour associated with a state of the context.



Mechanism

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

Advantages

- **Clean Code:** Pomotes cleaner code by encapsulating each state's behavior within its own class, making the code more modular.
- **Flexibility:** Allows for easy addition of new states without modifying existing code. Each state is independent and can be added or modified without affecting other states.
- **Scalability:** The pattern scales well as the number of states and state-specific behaviors increases. It provides a scalable and maintainable solution for managing complex systems with a number of states.
- **Facilitates Testing:** Testing becomes manageable as each state can be tested independently without worrying about interactions with other states.

Disadvantages

- **Complexity:** The pattern introduces additional classes, which makes the program more complex.
- **Over-engineering:** The State Pattern can lead to a larger number of classes, which may be overwhelming for small projects that can be handled with conditional logic.

Implementation

Defining state transitions

- Allow the context method to define state transitions, by delegating requests to the state that it is currently on.
- State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context'scurrent state explicitly.

Creating and destroying State objects

- There are two approaches: to create State objects only when they are needed and destroy them thereafter versus creating them ahead of time and never destroying them.
- The first approach avoids creating objects that won't be used, which is important if the State objects store a lot of information.
- The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly.
- The second approach is dependent on languages that do not have garbage collection as this will automatically remove objects that are no longer used.

3.9 Strategy

Definition

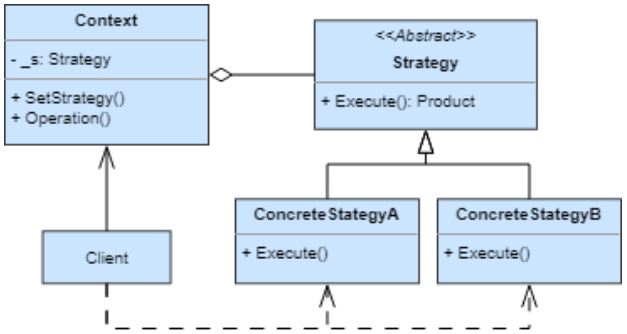
- It is a behavioural design pattern.
- Define a family oafl gorit h ms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known as Policy.

Applicability

- **Multiple Algorithms:** multiple algorithms that can be used interchangeably based on different contexts.
- **Encapsulating Algorithms:** encapsulate the implementation details of algorithms separately from the context that uses them.
- **Runtime Selection:** dynamically switch between different algorithms at runtime based on user preferences, configuration settings, or system states.
- **Reducing Conditional Statements:** a class defines many behaviours, and these appear as multiple conditional statements in its operations.
- **Testing and Extensibility:** to facilitate easier unit testing by enabling the substitution of algorithms with mock objects or stubs.

Structure

- **Strategy:** declares an interface common to all supported algorithms. Context uses this interface to call the algorthim defined by ContreteStrategy.
- **ConcreteStrategy:** implements the algorithm defined by a ConcreteStrategy.
- **Context:** icsonfi gur ed with a ConcreteStrategy object. maintains a reference to a Strategy object. may define an interface that lets Strategy access its data.



Mechanism

- Strategy and Context interact to implement the chosen algorithm.
- A context may pass all data required by the algorithm to the strategy when the algorithm is called.
- Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients create and pass a ConcreteStrategy object to the context; and only interact with the context.
- There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

Advantages

- **Families of related algorithms:** A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
- **An alternative to subclassing:** By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
- **Strategies eliminate conditional statements:** Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.
- **A choice of implementations:** Strategies can provide different implementations of the same behavior. The application can switch strategies at run-time.

Disadvantages

- **Clients must be aware of different Strategies:** Clients must be aware of the differences between strategies to be able to select a proper one
- **Communication overhead between Strategy and Context:** the Strategy interface is shared by all ConcreteStrategy classes even if ConcreteStrategies won't use all the information passed to them through this interface. That means that sometimes the context creates and initializes parameters that never get used.
- **Increased number of objects:**

Implementation

Item

-

3.10 Template Method

Definition

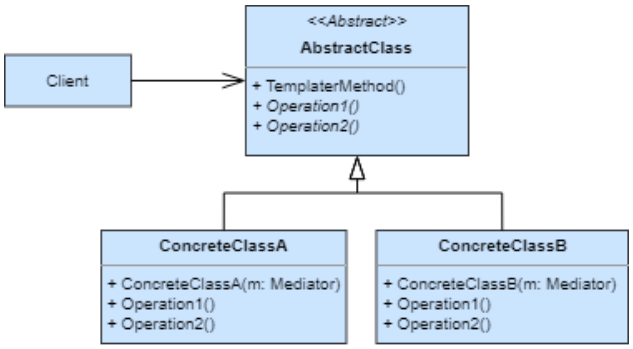
- It is a behavioural design pattern.
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Applicability

- **Common Algorithm with Variations:** to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- **Enforcing Structure:** enforce a specific structure or sequence of steps in an algorithm while allowing for flexibility in certain parts.
- **Reducing Duplication:** By centralizing common behavior in the abstract class and avoiding duplication of code in subclasses.

Structure

- **AbstractClass:** defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm. Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass:** implements the primitive operations to car ry out subclass-specific steps of the algorithm.



Mechanism

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Advantages

- **Item:** lets clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- **Item:** You can pull the duplicate code into a superclass.

Disadvantages

- **Liminations:** Some clients may be limited by the provided skeleton of an algorithm.
- **Liskov Substitution:** You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.
- **Maintainability:** Template methods tend to be harder to maintain the more steps they have.

Implementation

Item

3.11 Visitor Method

Definition

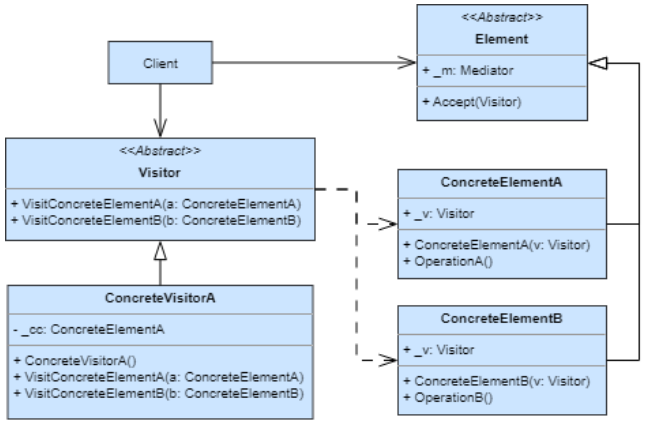
- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability

- **Operations on all objects:** perform an operation on all elements of a complex object structure.
- **Decoupling:** keep related operations together by defining them in one class and clean up the business logic of auxiliary behaviors.
- **Class behaviour:** behavior makes sense only in some classes of a class hierarchy, but not in others. Extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty

Structure

- **Visitor:** declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor and lets the visitor determine the concrete class of the element being visited and lets the visitor access the element directly through its interface.
- **ConcreteVisitor:** implements each operation declared by Visit or. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element:** defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement:** implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure:** can enumerate itselements, m ayprovide a high-level interface to allow th e visitor to visit its elements and may either be a composite or a collection such as a list or a set.



Mechanism

- A client that uses the Visit or pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

Consequences

Advantages

- **Decoupling:** creates cleanness of responsibilities by structuring the algorithm or operation independent of the object that this operation acts on. The separation lets you decouple the pure logic of an operation in a class and gives you concrete visitors as concrete implementations
- **Open/Closed Principle:** introduce a new behaviour easily that can work with objects of different classes without changing these classes.
- **Single Responsibility Principle:** move multiple versions of the same behavior into the same class.
- **Localised Logic:** Related behavior is localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses. this simplifies both the classes defining the elements and the algorithms defined in the visitors.
- **Accumulating State:** Visitors can accumulate state as they visit each element in the object structure. Otherwise, this state would be passed as extra arguments to the operations that perform the traversal.

Disadvantages

- **Extensibility:** need to update all visitors each time a class gets added to or removed from the element hierarchy. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class
- **Limited Access:** visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.
- **Breaking encapsulation:** pattern assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, the pattern forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.

Implementation

Item

-