**DESIGN PATTERN**

**I. The Strategy Pattern**

**📑 Disadvantages of using *inheritance*:**

- Code is duplicated across subclasses.

- Runtime behavior changes are difficult.

- Hard to gain knowledge of all behaviors.

- Changes can unintentionally affect other classes.

☯ **Design principle:** Identify the aspects of your application that vary and separate them from what stays the same.

- Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.

☯ **Design principle:** Program to an interface, not an implementation.

- Using an interface to represent each behavior and each implementation of a behavior will implement one of those interfaces.

**📑 “Program to an interface” really means “Program to a supertype”**

**-** Program to a supertype equivalent to the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be any concrete implementation of the supertype, which means the class declaring them doesn’t have to know about actual object types.

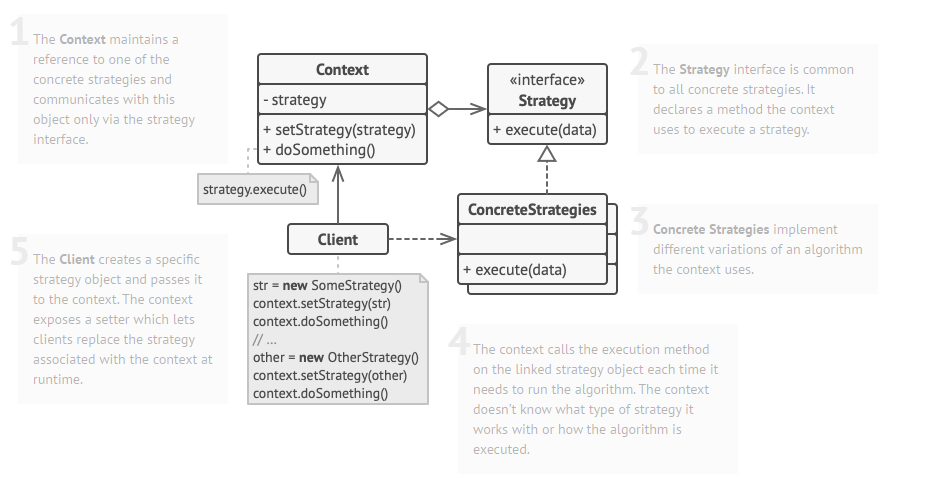
❔ **There is no dumb questions**

- When designing an application, we should anticipate areas that are going to vary and build the flexibility to deal with it into our code. We’ll find that the principles and patterns can be applied at any stage of the development lifecycle.

☯ **Design principle:** Favor composition over inheritance

- System using composition gives us more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you’re composing with implements the correct behavior interface.

📘**The Strategy Pattern** defines a family of algorithms, encapsulated each one, and makes them interchangeable.



**📑 Advantage of shared pattern vocabulary**

- When communicating with another developer using patterns, it’s communicating a whole set of qualities, characteristics and constraints that the pattern represents.

- Description quickly and precisely of the design used.

- Talking using patterns allows us to keep the discussion at the design level.

- Turbo charge development team.

**❔ Bullet points**

- Knowing the OO basics does not make you a good OO designer.

- Good OO designs are reusable, extensible and maintainable.

- Patterns show you how to build systems with good OO design qualities.

- Patterns are proven object-oriented experiences.

- Patterns don’t give you code, they give you general solutions to design problems. You apply them to your specific application.

- Patterns aren’t invented, they are discovered.

- Most patterns and principles address issues of change in software.

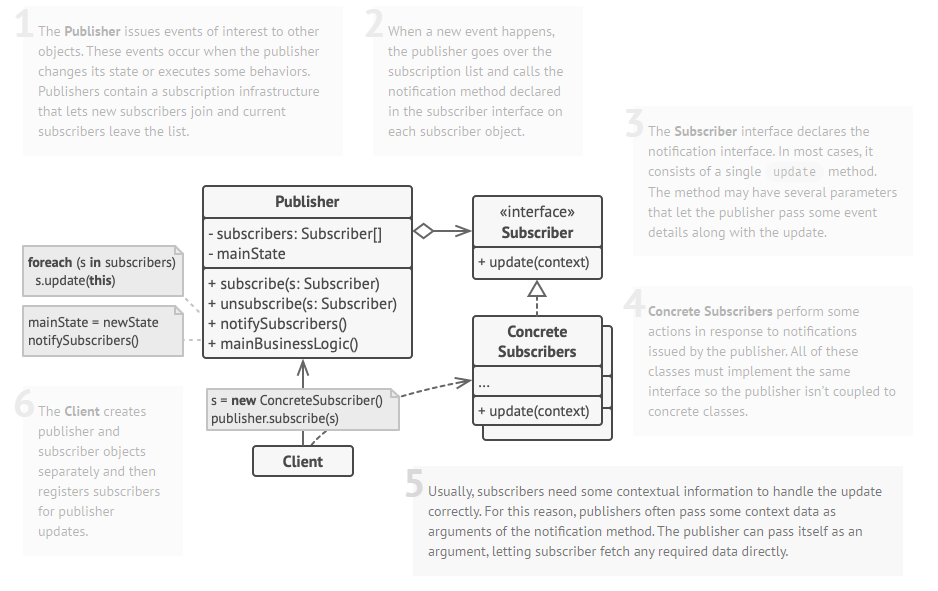
- Most patterns allow some part of a system to vary independently of all other parts.

- We often try to take what varies in a system and encapsulate it.

- Patterns provide a shared language that can maximize the value of your communication with other developers.

**II. The Observer Pattern**

📘 **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



❔ **There is no dumb questions**

**-** One-to-many relationship: one Subject to the many Observers.

- The observers are dependent on the subject to update them when data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

**📑 The power of loose coupling**

- When 2 objects are loosely coupled, they can interact, but have very little knowledge of each other.

- The observer pattern provides an object design where subjects and observers are loosely coupled.

- The only thing the subject knows about an observer is that it implements a certain interface. It doesn’t need to know the concrete class of the observer.

- We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface.

- We never need to modify the subject to add new types of observers. We don’t need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer.

- We can reuse subjects or observers independently of each other. If we have another user for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

- Change to either the subject or an observer will not affect the other as long as the object still meets their obligations to implement the subject or observer interfaces.

☯ **Design principle:** Strive for loosely coupled designs between objects that interact.

- Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

**❔ Bullet points**

**-** The Observer Pattern defines a one-to-many relationship between objects.

- Subjects (Observables) update Observers using a common interface.

- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer Interface.

- You can push or pull data from the Observable when using the pattern.

- Don’t depend on a specific order of notification for your Observers.

**III. Decorator Pattern**

☯ **Design principle (Open-closed Principle):** Classes should be open for extension, but closed for modification.

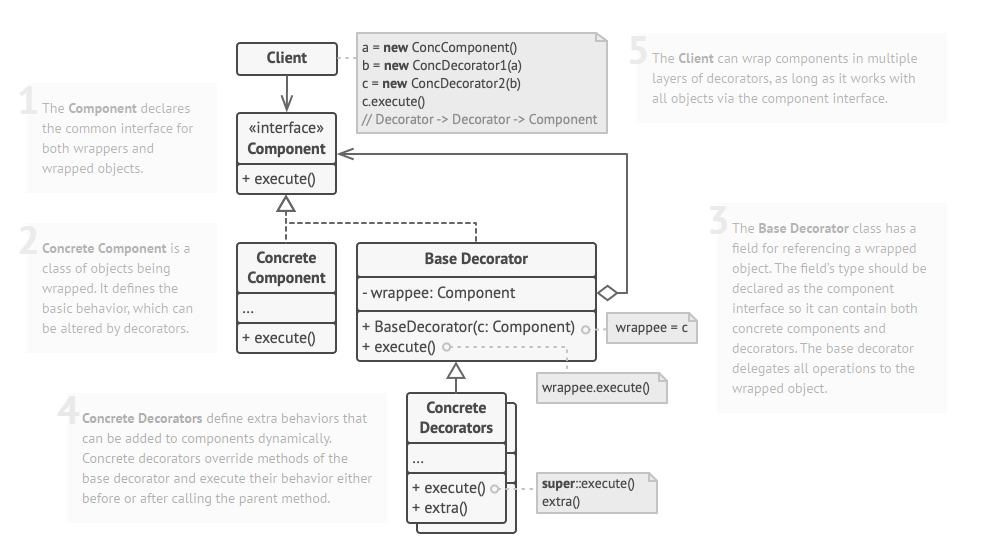
- Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. It takes designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

❔ **There is no dumb questions**

**-** Example for open-close principle: In Observer Pattern, by adding new Observers, we can extend the Subject at any time, without adding code to the Subject.

- There are techniques for allowing code to be extended without direct modification. But be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

📘 **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



❔ **Cubicle Conversation**

**-** We’re subclassing the wrapped object in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as other decorators.

- If we rely on inheritance, then our behavior can only be determined statically at compile time. With composition, we can mix and match decorators any way we like at runtime.

- We could use an interface instead of an abstract class but we always try to avoid altering existing code.

**❔ Bullet points**

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.

- In our designs we should allow behavior to be extended without the need to modify existing code.

- Composition and delegation can often be used to add new behaviors at runtime.

- The Decorator Pattern provides an alternative to subclassing for extending behavior.

- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.

- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)

- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.

- You can wrap a component with any number of decorators.

- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component’s concrete type.

- Decorators can result in many small objects in our design, and overuse can be complex.

**IV. The Factory pattern**

❔ **There is no dumb questions**

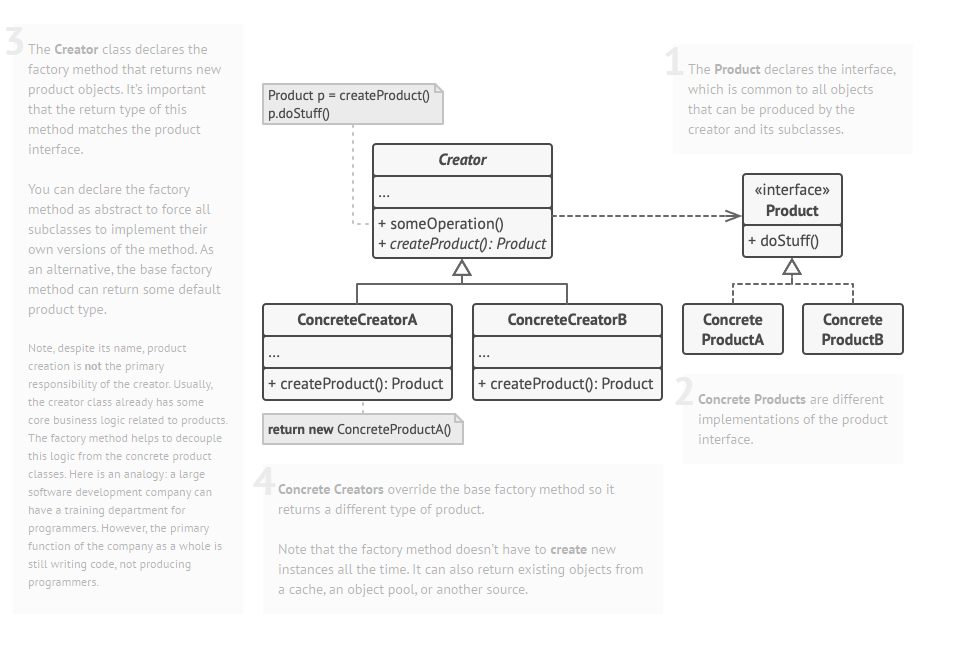
- Advantage of simple factory: By encapsulating the concrete instantiations in one class, we now have only one place to make modifications when the implementation changes.

- A simple factory defined as a static method called a static factory. We don’t need to instantiate objects to make use of the create method. But it has a disadvantage that we can’t subclass and change behavior of the create method.

**📑 The Simple Factory:** isn’t actually a Design Pattern (it’s more of a programming idiom), but it is commonly used.

📘 **Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclass.

**-** Factory method pattern defines a factory method that handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.



❔ **There is no dumb questions**

**-** Advantage of Factory Method Pattern when we have only one concrete creator: we are decoupling the implementation of the product from its use. If products are added or a product’s implementation changed, it will not affect the creator.

- In Factory Method Pattern, the concrete classes are extending a class which has defined the factory method as an abstract method. Thus, we can decide what concrete classes will be returned. While in Simple Factory, the factory is another object composed with the creator, which gives us a way to encapsulate the object creator but does not give us the flexibility to vary the classes we are creating.

- We can define a default factory method to instantiate some concrete classes (not using the abstract method). Then we always have a means of instantiating classes even if there are no subclasses of the creator.

- We can implement what is known as the parameterized factory method. It can make more than one object based on a parameter passed in. (However, a factory often produces one object and is not parameterized).

☯ **Design principle (Dependency Inversion Principle):** Depend upon abstractions. Do not depend upon concrete classes.

- This principle is like the “Program to an interface, not an implementation” principle but it makes a stronger statement about abstraction. It suggests that the high-level components should not depend on the low-level components; rether, they should both depend on abstractions.

- The “Inversion” means that not only low-level components depend on abstraction but also high-level components.

**📑 Guidelines help us avoid OO designs that violate the Dependency Inversion Principle:**

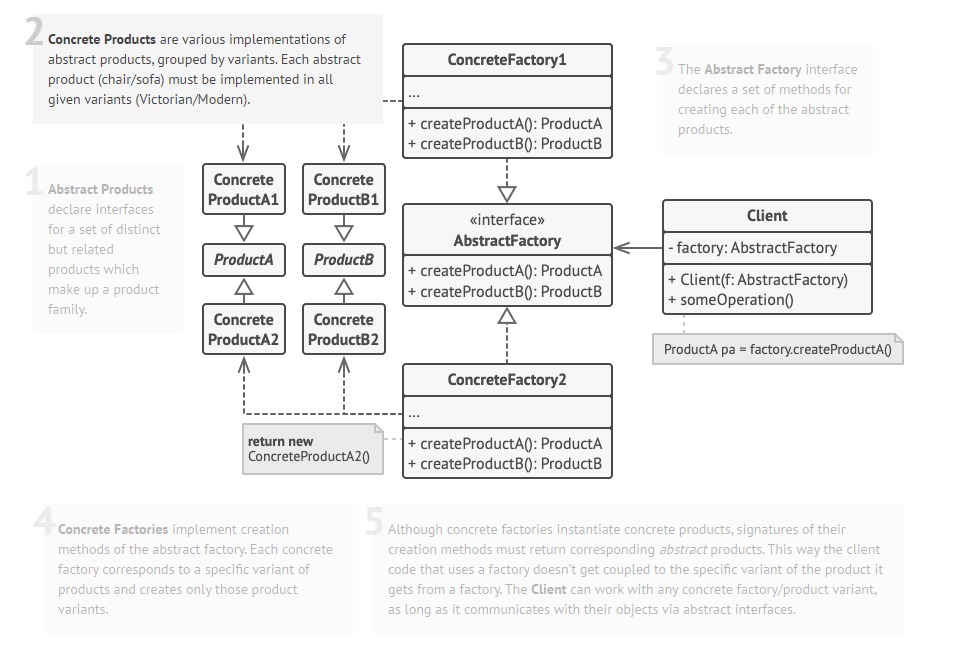
- No variable should hold a reference to a concrete class. (No **new** operator)

- No class should derive from a concrete class. (Classes are derived from an abstraction like interface or an abstract class)

- No method should override an implemented method of any of its base classes. (No override method)

📘 **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products.



**📑 Difference between Factory Method and Abstract Factory**

|  |  |
| --- | --- |
| **Factory Method** | **Abstract Factory** |
| - Creating objects through inheritance.  - Using a subclass to do creation. Clients only need to know the abstract type they are using, the subclass worries about the concrete type.  - Often creating one product.  - Usually implement code in abstract creator that makes use of the concrete types the subclasses create | - Create objects through composition.  - Subclasses define how those products are produced. To use the factory, instantiating one and passing it into some code that is written against the abstract type.  - Changing interface means change the interface of every subclass  - Factory methods are a natural way to implement product methods in abstract factories. |

**❔ Bullet points**

**-** All factories encapsulate object creation

- Simple Factory, while not a bonafide design pattern, is a simple way to decouple clients from concrete classes.

- Factory Method relies on inheritance: object creation is delegated to subclasses which implements the factory method to create objects.

- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.

- All factory patterns promote loose coupling by reducing the dependence of application on concrete classes.

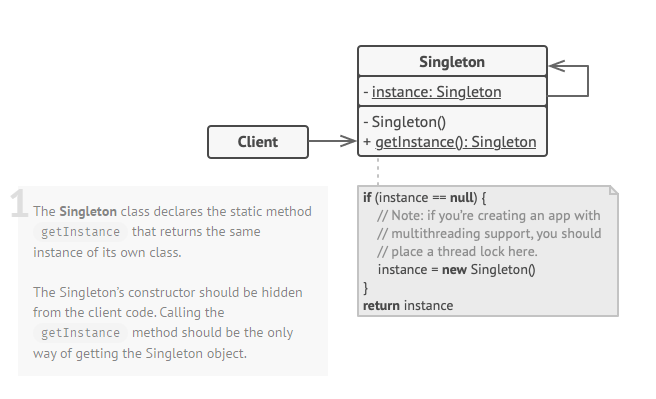
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.

- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.

- Factories are a powerful technique for coding to abstractions, not concrete classes.

**V. The Singleton Pattern**

📘 **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.



**❔ Bullet points**

**-** The Singleton Pattern ensures you have at most one instance of a class in your application.

- The Singleton Pattern also provides a global access point to that instance.

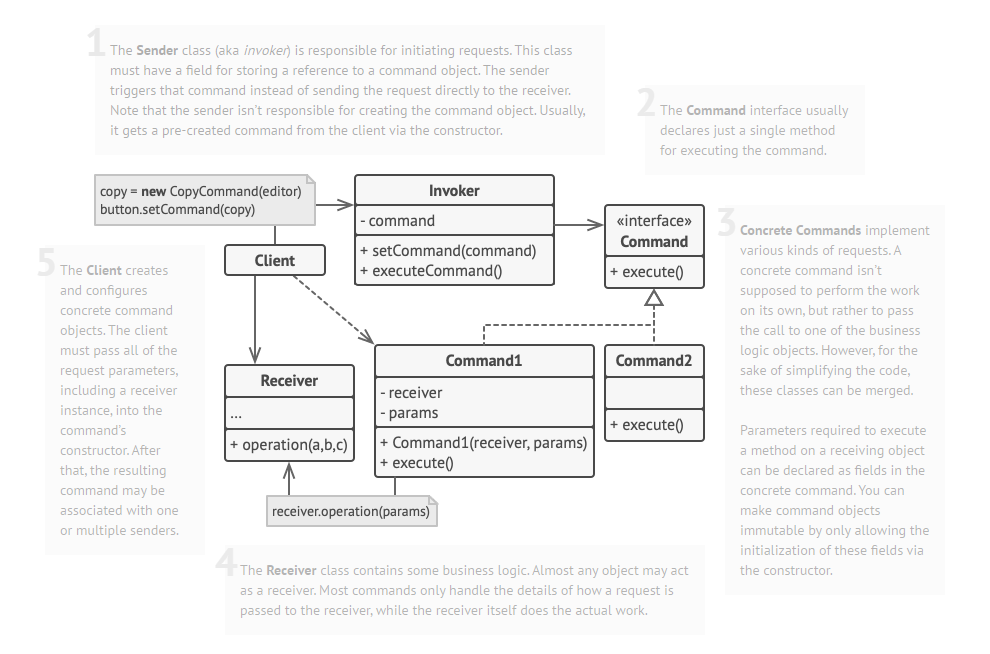
- Implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.

- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multi-threaded applications (we should consider all applications multi-thread).

- Be careful if you are using multi class loaders. This could defeat the Singleton implementation and result in multi-instances.

**VI. The Command pattern**

📘 **The Command Pattern** encapsulates a request as an object, thereby letting you parameterize other objects with different request, queue or log requests, and support undoable operations.



**📑** The **NoCommand** object is an example of a *null* object. A null object is useful when we don’t have a meaningful object to return.

**❔ Bullet points**

**-** The Command Pattern decouples an object, making a request from the one that knows from the one that knows how to perform it.

- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).

- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.

**-** Invokers can be parameterized with Commands, even dynamically at runtime.

- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.

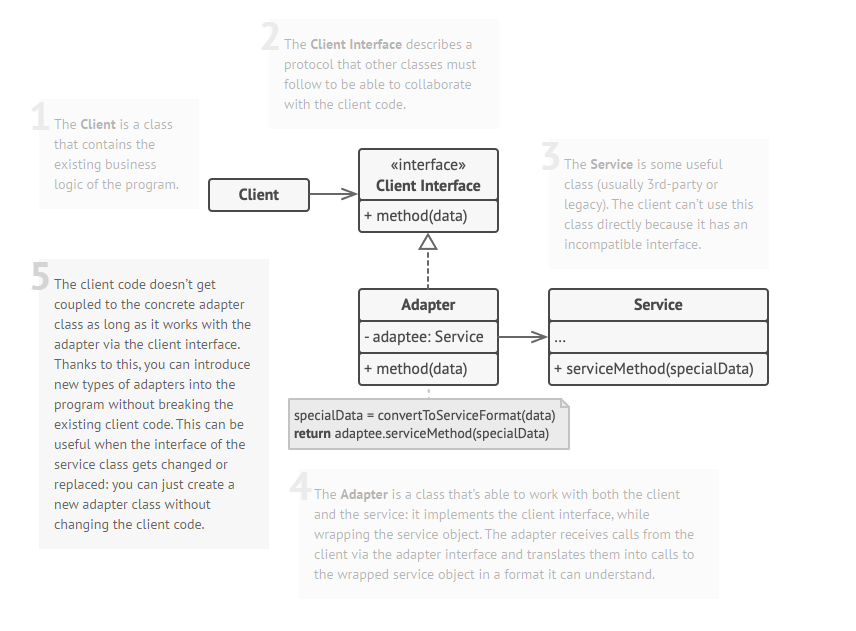
- Macro Commands are a simple extension of Commands that allow multiple commands to be invoked. Likewise, Macro Commands can easily support the undo method.

- In practice, it is not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receiver.

- Commands may also be used to implement logging and transactional systems.

**VII. The Adaptor and Facade Pattern**

📘 **The Adaptor Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

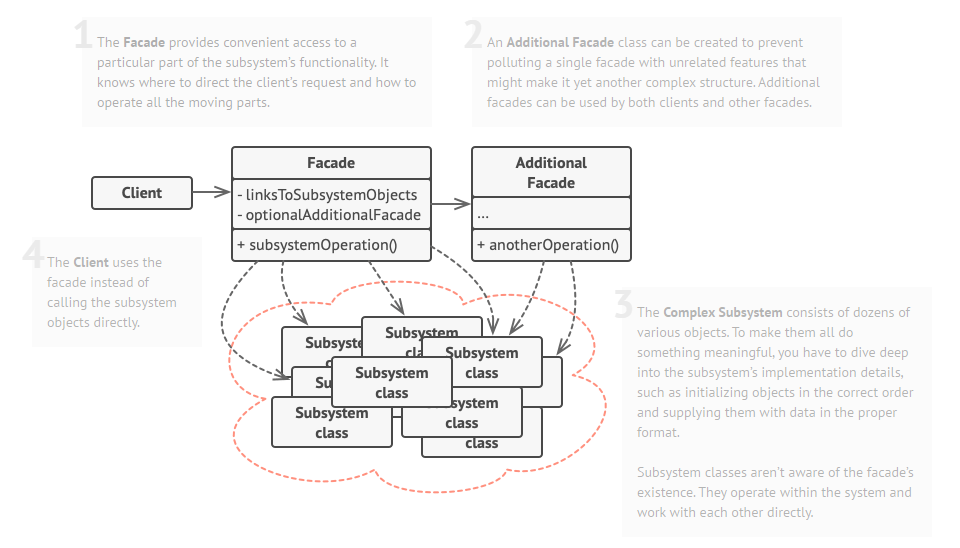


❔ **There is no dumb questions**

- A facade not only simplifies an interface, it decouples a client from a subsystem of components

- Facades and adapters may wrap multiple classes, but a facade’s intent is to simplify, while an adapter’s is to convert the interface to something different.

📘 **The Facade Pattern** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



☯ **Design Principle (The principle of Least Knowledge):** Talk only to your immediate friends.

- This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts.

- When we build a lot of dependencies between many classes, we are building a fragile system that will be costly to maintain and complex for others to understand.

- The principle tells us that we should only invoke method that belongs to:

1. The object itself.

2. Object passed in as a parameter to the method.

3. Any object the method creates or instantiates.

4. Any component of the object.

❔ **There is no dumb questions**

- The principle of Least Knowledge also has another name called the Law of Demeter.

- The principle of Least Knowledge reduces dependencies between objects and software maintenance but increases the complexity and decreases runtime performance.

**❔ Bullet points**

**-** When you need to use an existing class and its interface is not the one you need, use an adapter.

- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.

- An adapter changes an interface into one a client expects.

- A Facade decouples a client from a complex subsystem.

- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.

- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.

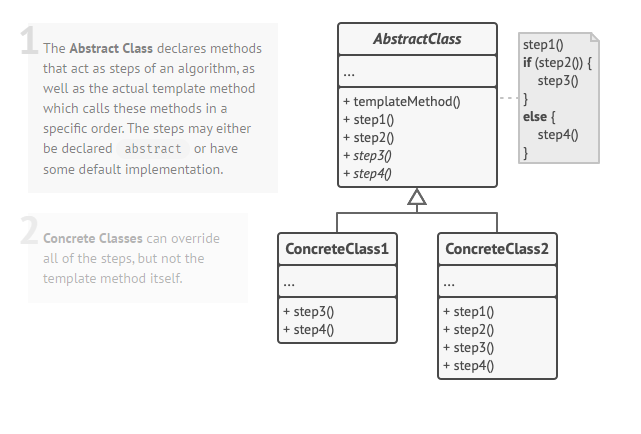
- There are two forms of the Adapter Pattern: object and class adapters. Class adapter requires multi inheritance.

- You can implement more than one facade for a subsystem.

- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade wraps a set of objects to simplify.

**VIII. The Template Method Pattern**

📘 **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.



**📑 A hook** is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

- To use the hook, we override it in our subclasses.

☯ **Design Principle (The Hollywood Principle)** don’t call us, we call you.

- The Hollywood principle gives us a way to prevent “dependency rot”. Dependency rot happens when you have high-level components depending on low-level components depending on high-level components on sideways components depending on low-level components, and so on.

- With the Hollywood principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how.

❔ **There is no dumb questions**

**-** Use abstract method when subclass MUST provide an implementation of method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, as subclasses may choose to implement that hook, but it doesn’t have to.

❔ **There is no dumb questions**

**-** The Dependency Inversion Principle teaches us to avoid the use of the concrete classes and instead work as much as possible with abstractions.

- The Hollywood Principle is a technique for building frameworks or components so that low-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers.

- The Dependency Inversion Principle makes a much stronger and general statement about how to avoid dependencies in design.

**❔ Bullet points**

**-** A template method defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.

- The Template Method Pattern gives us an important technique for code reuse.

- The template method’s abstract class may define concrete methods, abstract methods and hooks.

- Abstract methods are implemented by subclasses.

- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.

- To prevent subclasses from changing the algorithm in the template method, declare the method as final.

- The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low-level modules.

- You’ll see lots of uses of the Template Method Pattern in real world code, but don’t expect it all (like any pattern) to be designed “by the book”

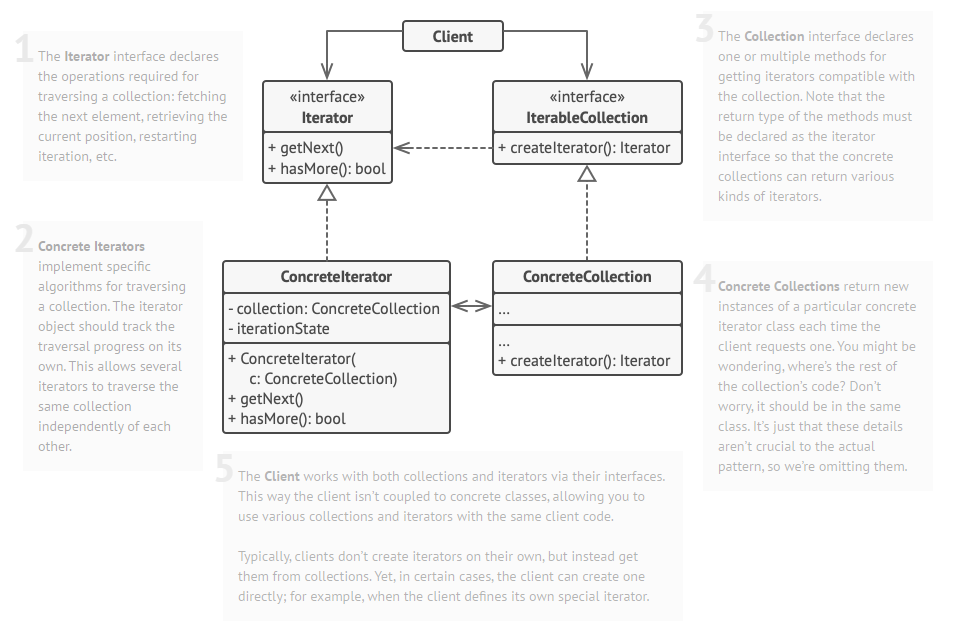
- The Strategy and Template Method both encapsulate algorithms, one by inheritance and one by composition.

- The Factory Method is a specialization of the Template Method.

**IX. The Iterator and Composite Pattern**

📘 **The Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- The Iterator Pattern also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.



❔ **There is no dumb questions**

**-** Iterators imply no ordering.

☯ **Design Principle (Single Responsibility Principle):** A class should have only one reason to change.

- Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

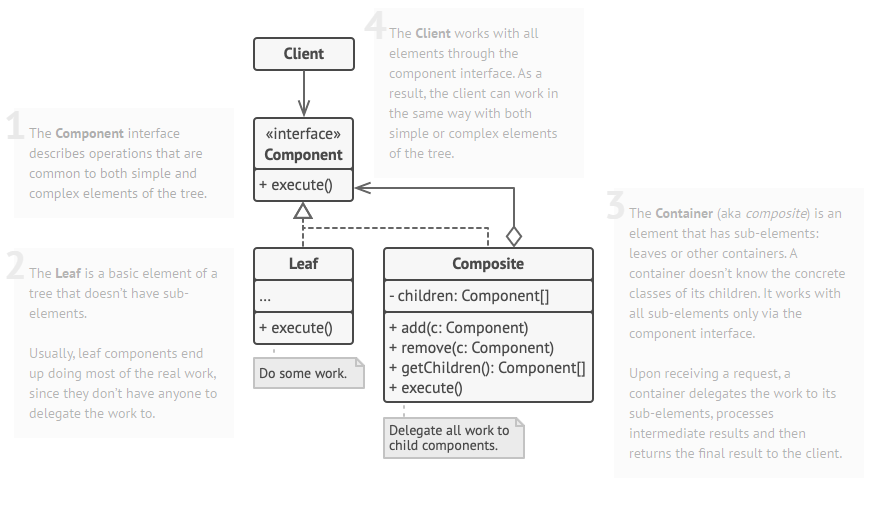
- **Cohesion** is a term used as a measure of how closely a class or a module supports a single purpose or responsibility.

- A module or class has high cohesion when it is designed around a set of related functions, and we say it has low cohesion when it is designed around a set of unrelated functions. And it has low cohesion when it is designed around a set of unrelated functions.

- Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

📘 **The Composite Pattern** allows you to compose object into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



**❔ Bullet points**

**-** An Iterator allows access to an aggregate’s elements without exposing its internal structure.

- An Interator takes the job of iterating over an aggregate and encapsulates it in another object.

- When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.

- An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.

- We should strive to assign only one responsibility to each class.

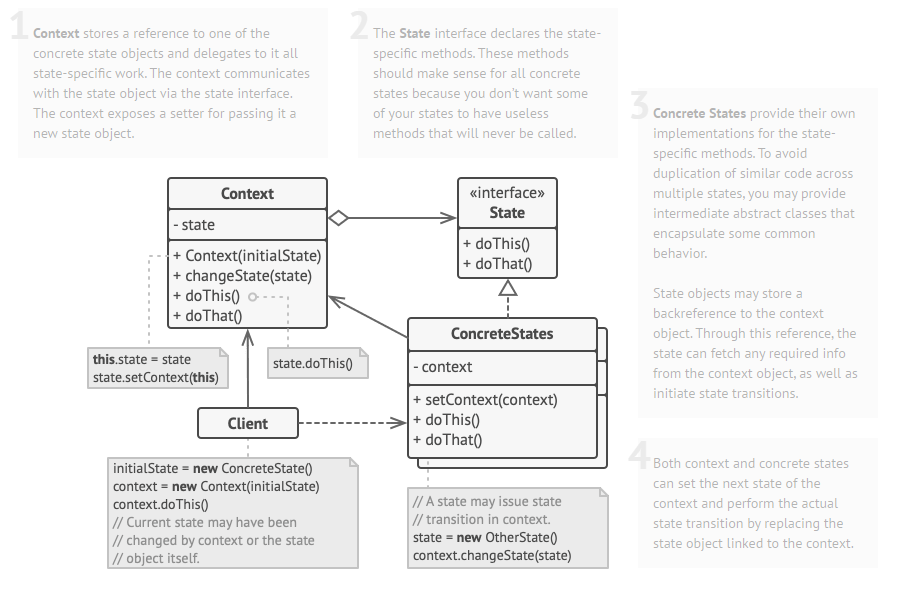
- The composite Pattern allows clients to treat composites and individual objects uniformly.

- A Component is any object in a Composite structure. Components may be other composites or the leaf nodes.

- There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

**X. The State Pattern**

📘 **The State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



**❔ Bullet points**

**-** The State Pattern allows an object to have many different behaviors that are based on its internal state.

- Unlike procedural state machines, the State Pattern represents the state as a full-blown class.

- The Context gets its behavior by delegating to the current state object it is composed with.

- By encapsulating each state into a class, we localize any changes that will need to be made.

- The State and Strategy Patterns have the same class diagram, but they differ in intent.

- Strategy Pattern typically configures Context classes with a behavior or algorithm.

- State Pattern allows a Context to change its behavior as the state of the Context changes.

- State transitions can be controlled by State class or by the Context class.

- Using the State Pattern will typically result in a greater number of classes in your design.

- State classes may be shared among Context instances.

**XI. The Proxy Pattern**

📘 **The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.

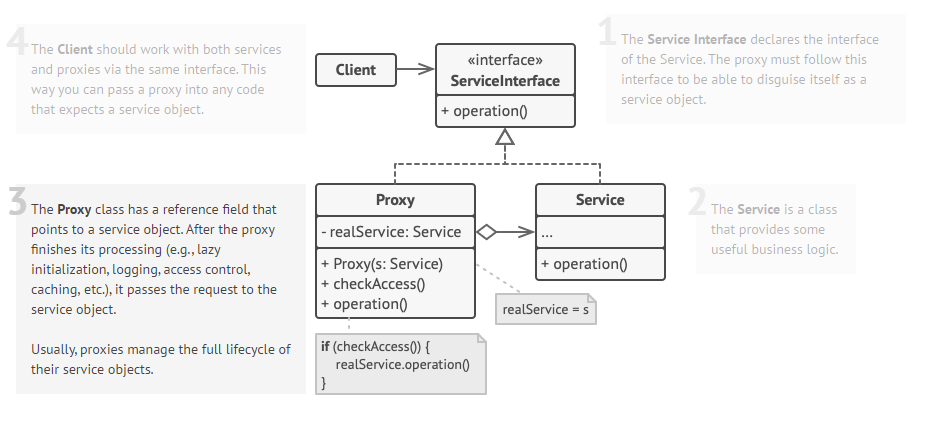
- Use the proxy pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing

- Few way proxies control access:

+ A remote proxy controls access to a remote object.

+ A virtual proxy controls access to a resource that is expensive to create.

+ A protection proxy controls access to a resource based on access rights.



**📑 The Proxy Zoo:** other proxies

- **Firewall Proxy** controls access to a set of network resources, protecting the subject from “bad” clients.

- **Smart Reference Proxy** provides additional actions whenever a subject is referenced, such as counting the number of references to an object.

- **Caching Proxy** provides temporary storage for results of operations that are expensive. It can allow multiple clients to share the results to reduce computation or network latency.

**- Synchronization Proxy** provides safe access to a subject from multiple threads

- **Complexity Hiding Proxy** hides the complexity of and controls access to a complex set of classes. This is sometimes called the Facade Proxy for obvious reasons. The Complexity Hiding Pattern in that the proxy controls access, while the Facade Pattern just provides an alternative interface.

**- Copy-On-Write Proxy** controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of the Virtual Proxy.

**❔ Bullet points**

**-** The Proxy Pattern provides a representative for another object in order to control the client’s access to it. There are a number of ways it can manage that access.

- A Remote Proxy manages interaction between a client and a remote object.

- A Virtual Proxy controls access to an object that is expensive to instantiate.

- A Protection Proxy controls access to the methods of an object based on the caller.

- Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-wire proxies, and so on.

- Proxy is structurally similar to Decorator, but two differ in their purpose.

- The Decorator Pattern adds behavior to an object, while a Proxy controls access.

- Like any wrapper, proxies will increase the number of classes and objects your designs.

**XII. Compound Patterns**

📘 **A compound pattern** combines two or more patterns into a solution that solves a recurring or general problem.

**❔ Bullet points**

- The Model View Controller Pattern is a compound pattern consisting of the Observer, Strategy and Composite patterns.

- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them.

- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.

- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons.

- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.

- The Adapter Pattern can be used to adapt a new model to an existing view and controller.

**XIII. Better Living With Patterns**

📘 **A Pattern** is a solution to a problem in a context

- The **context** is the situation in which the pattern applies. This should be a recurring situation.

- The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

- The **solution** is what you’re after: a general design that anyone can apply which resolves the goal and set of constraints.

**📑Pattern catalogs** consist of:

**-** Name**:** a good name helps the pattern become a part of the vocabulary that you share with other developers.

- Classification (category).

- Intent: describes what the pattern does in a short statement (pattern’s definition).

- Motivation: gives a concrete scenario that describes the problem and how the solution solves the problem.

- Applicability: describes situations in which the pattern can be applied.

- Structure: provides a diagram illustrating the relationships among the classes that participate in the pattern.

- Participants: describe their responsibilities and roles in the pattern.

- Collaborations: tell us how the participants work together in the pattern.

- Consequences: describe the good and bad effects that using this pattern may have.

- Implementation: provides techniques you need to use when implementing this pattern, and issues you should watch out for.

- Sample code: provides code fragments that might help with the implementation.

- Know uses: describes examples of this pattern found in real systems.

- Related patterns: describes the relationships between this pattern and others.

**📑Pattern categories**

- Classified based on purposes

**+ Creational Patterns** involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

+ Singleton, Abstract Factory, Factory Method, Builder, Protopyte

+ **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.

+ Template Method, Iterator, Command, Observer, State, Strategy, Memento, Visitor, Interpreter, Chain of Responsibility, Mediator.

+ **Structural patterns** compose classes or objects into larger structures.

+ Decorator, Proxy, Composite, Facade, Adapter, Bridge, Flyweight

- Classified by whether or not the pattern deals with classes or objects.

**📑The Anti-pattern** tells us how to go from a problem to a bad solution.

**-** An anti-pattern tells you why a bad solution is attractive.

**-** An anti-pattern tells you why that solution in the long term is bad.

- An anti-pattern suggests other patterns that are applicable which may provide good solutions.

**❔ Bullet points**

**-** Let Design Patterns emerge in your designs, don’t force them in just for the sake of using a pattern.

- Design Patterns aren’t set in stone; adapt and tweak them to meet your needs.

- Always use the simplest solution that meets your needs, even if it doesn’t include a pattern.

- Study Design Pattern catalogs to familiarize yourself with patterns and the relationships among them.

- Pattern classifications (or categories) provide groupings for patterns. When they help, use them.

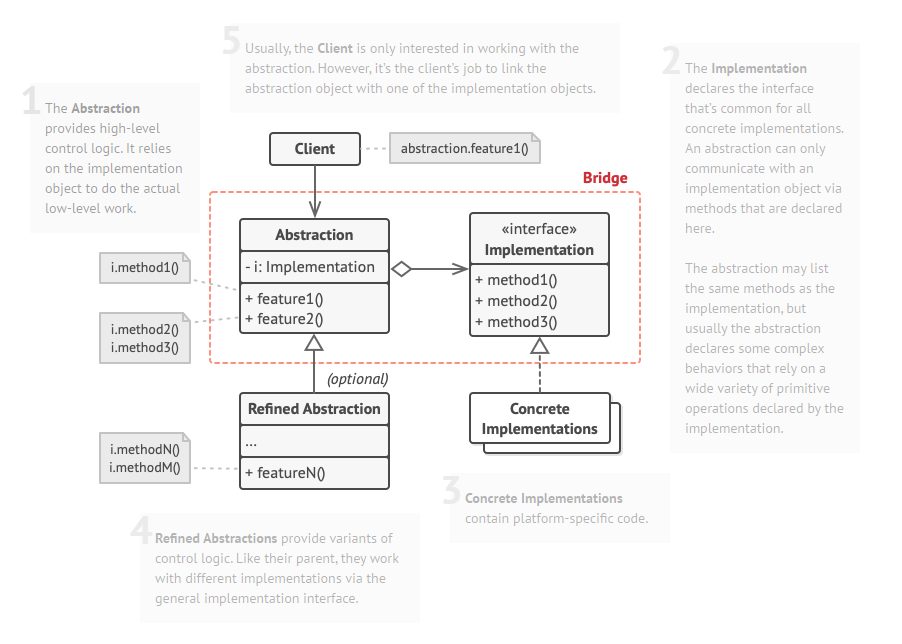
- You need to be committed to be a pattern writer: it takes time and patience, and you have to be willing to do lots of refinement.

- Most of the patterns you encounter will be adaptations of existing patterns, not new patterns.

- Build your team’s shared vocabulary. This is one of the most powerful benefits of using patterns.

**XIV. Leftover Pattern**

📘 **The Bridge Pattern** uses to vary not only your implementations, but also your abstractions.



**📑 Benefits**

**-** Decouple an implementation so that it is not bound permanently to an interface.

- Abstraction and implementation can be extended independently.

- Changes to the concrete abstraction classes don’t affect the client.

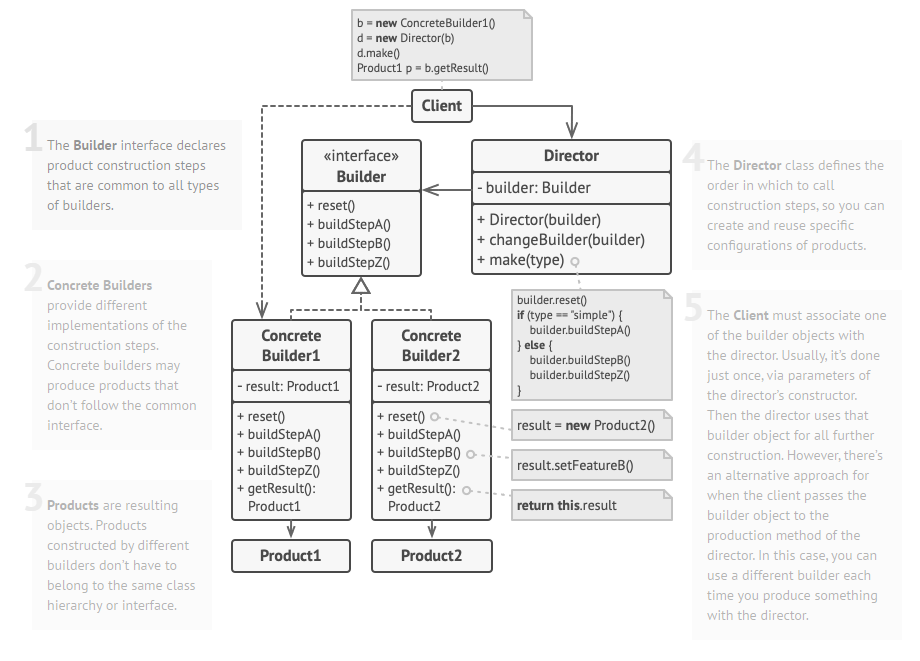
**📑 Use and Drawbacks**

**-** Useful in graphic and windowing systems that need to run over multiple platforms.

- Useful any time you need to vary an interface and an implementation in different ways.

- Increase complexity.

📘 **The Builder Pattern** uses to encapsulate the construction of a product and allow it to be constructed in steps



**📑 Benefits**

**-** Encapsulate the way a complex object is constructed.

- Allow objects to be constructed in a multistep and varying process (as opposed to one step factories).

- Hides the internal representation of the product from the client.

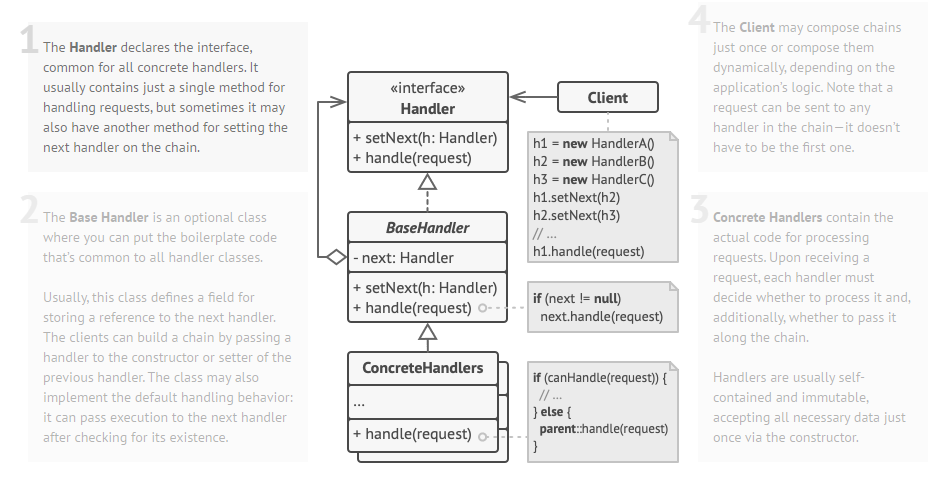
- Product implementation can be swapped in and out because the client only sees an abstract interface.

**📑 Use and Drawbacks**

**-** Often used for building composite structures.

- Constructing objects requires more domain knowledge of the client than when using a Factory.

📘 **The Chain of Responsibility Pattern** uses to give an object more than one a change to handle a request.



**📑 Benefits**

**-** Decouples the sender of the request and its receivers.

- Simplifies object because it doesn’t have to know the chain’s structure and keep direct references to its members.

- Allow you to add or remove responsibilities dynamically by changing the members or order of the chain

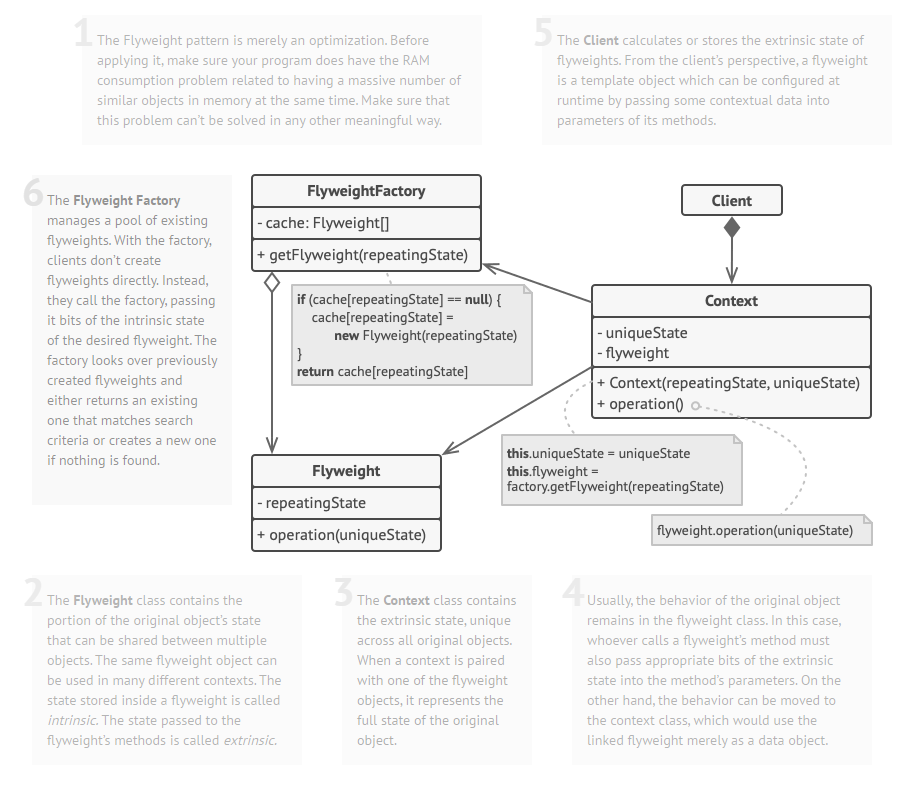
**📑 Use and Drawbacks**

**-** Commonly used in windows systems to handle events like mouse clicks and keyboard events.

- Execution of the request isn’t guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or disadvantage).

- Can be hard to observe the runtime characteristics and debug.

📘 **The Flyweight Pattern** uses to provide many “virtual instances” by on instance of a class.



**📑 Benefits**

**-** Reduces the number of object instances at runtime, saving memory.

- Centralizes state for many “virtual” objects into a single location.

**📑 Use and Drawbacks**

**-** The Flyweight is used when a class has many instances, and they can all be controlled identically.

- A drawback of the Flyweight pattern is that once it has been implemented, logical instances of the class will not be able to behave independently from the other instances.

📘 **The Interpreter Pattern** uses to build an interpreter for a language

**AbstractExpression**

*interpreter(context)*

**TerminalExpression**

*interpreter(context)*

**NonterminalExpression**

*interpreter(context)*

**📑 Benefits**

**-** Representing each grammar rule in a class makes the language easy to implement.

- Because the grammar is represented by classes, you can easily change or extend the language.

- By adding additional methods to the class structure, new behaviors can be added beyond interpretation, like pretty printing and more sophisticated program validation.

**📑 Use and Drawbacks**

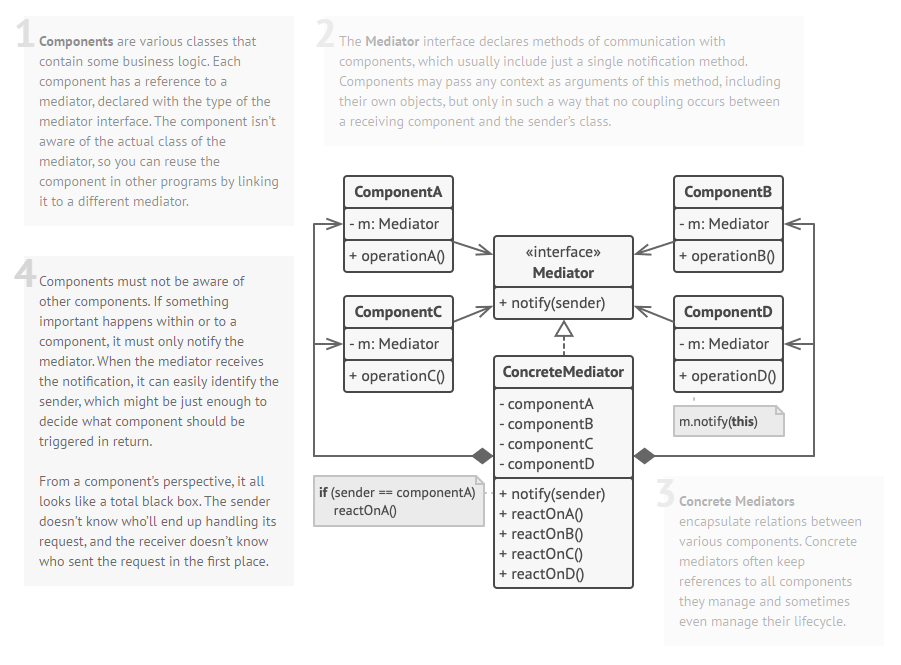
**-** Use Interpreter Pattern when you need to implement a simple language.

- Appropriate when you have a simple grammar and simplicity is more important than efficiency.

- Used for scripting and programming languages.

- This pattern can become cumbersome when the number of grammar rules is large. In these cases, a parser/compiler generator may be more appropriate.

📘 **The Mediator Pattern** uses to centralize complex communications and control between related objects



**📑 Benefits**

**-** Increase the reusability of the objects supported by the Mediator by decoupling them from the system.

- Simplifies maintenance of the system by centralizing control logic.

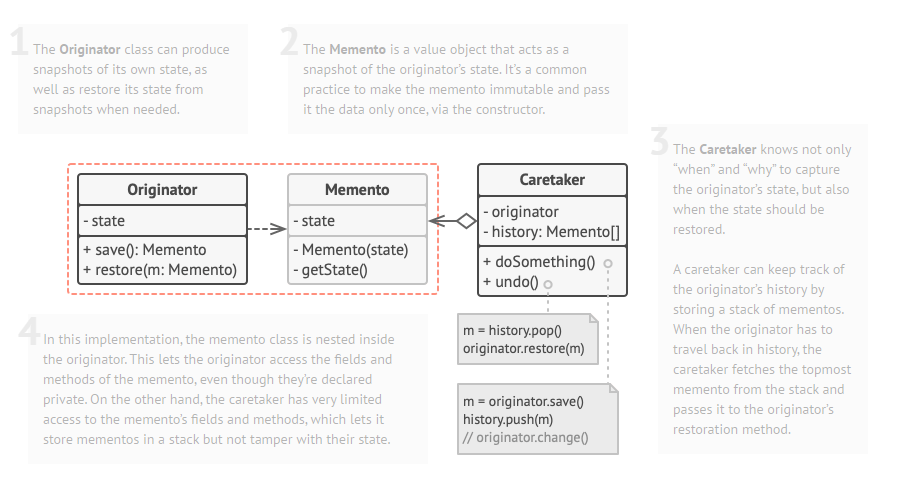
- Simplifies and reduces the variety of messages sent between objects in the system.

**📑 Use and Drawbacks**

**-** The Mediator is commonly used to coordinate related GUI components.

- A drawback of the Mediator pattern is that without a proper design, the Mediator object itself can become overly complex.

📘 **The Memeto Pattern** is used when you need to be able to return an object to one of its previous state.



**📑 Benefits**

**-** Keeping the saved state external from the key object helps to maintain cohesion.

- Keep the key object’s data encapsulated.

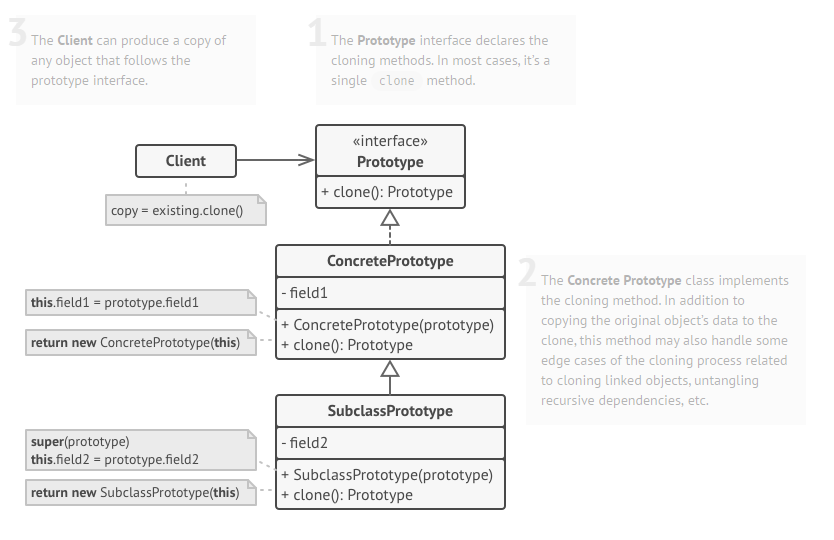
- Provides easy-to-implement recovery capability.

**📑 Use and Drawbacks**

**-** The Memeto is used to save state.

- A drawback to using Memeto is that saving and restoring state can be time consuming.

📘 **The Prototype Pattern** used when creating an instance of a given class is either expensive or complicated.



**📑 Benefits**

**-** Hides the complexities of making new instances from the client.

- Provides the option for the client to generate objects whose type is not known.

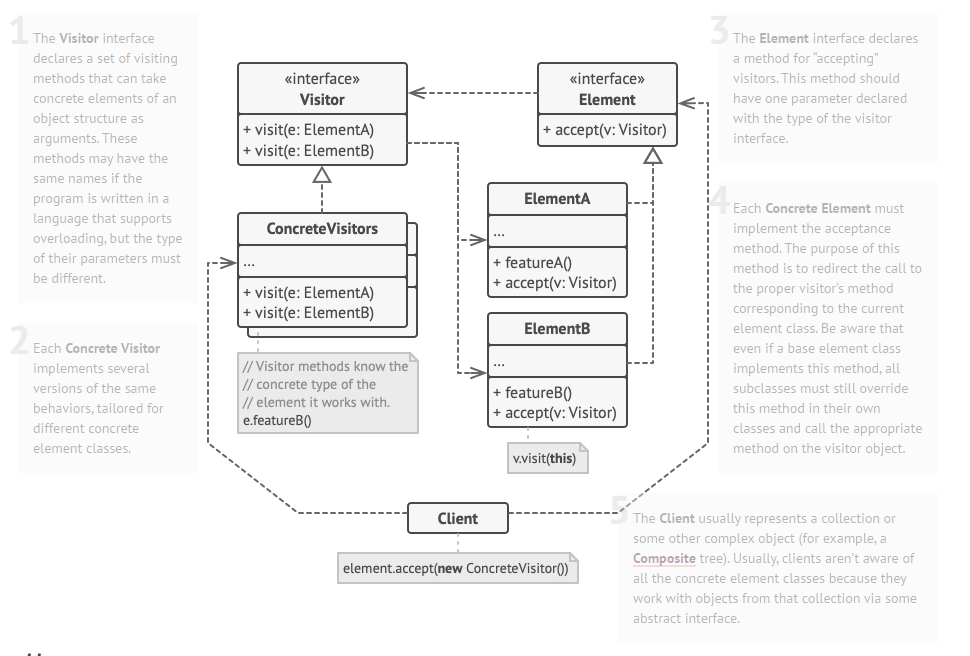
- In some circumstances, copying an object can be more efficient than creating a new object.

**📑 Use and Drawbacks**

**-** Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.

- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

📘 **The Visitor Pattern** uses to add capabilities to a composite of objects and encapsulation is not important.



**📑 Benefits**

**-** Allows you to add operations to a Composite structure without changing the structure itself.

- Adding new operations is relatively easy.

- The code for operations performed by the Visitor is contralized.

**📑 Use and Drawbacks**

**-** The Composite classes’ encapsulation is broken when the Visitor is used.

- Because the traversal function is involved, changes to the Composite structure are more difficult.