Design Patterns

Table of Contents

Preface

In this cheatsheet we are going to talk about design patterns in software development. We will focus on what they are, how they can benefit us, but more importantly where and when to use them.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

1. Introduction

Design patterns are reusable solutions to common software design problems. They provide a way to describe and document software architectures, as well as a common vocabulary for developers to communicate about software design.

There are several types of design patterns, including creational, structural, and behavioral patterns.

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Structural patterns deal with object composition, creating relationships between objects to form larger structures.

Behavioral patterns focus on communication between objects, what goes on between objects and how they operate together.

2. Creational patterns

2.1. Singleton

The Singleton design pattern is used to ensure that a class has only one instance, and to provide a global access point to that instance.

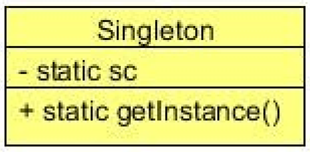

Figure 1. Singleton UML.

- One advantage of using the Singleton design pattern is that it ensures that there is only one instance of a class, which can be useful for classes that manage resources such as database connections or network sockets.
- It also provides a global access point to the instance, which can make it easier to use the instance in different parts of the code.

2.2. Factory

Provides a way to create objects without specifying the exact class of object that will be created. Has a method that creates objects of a specific type. The method takes the type of object to be created as an argument and returns a new object of that type.

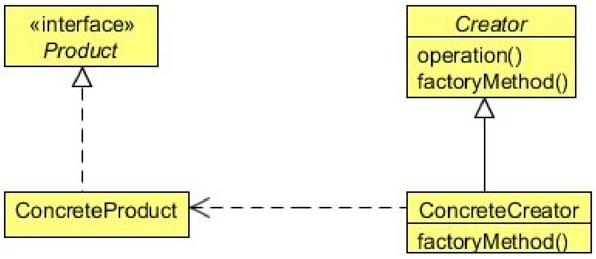

Figure 2. Factory UML.

- One advantage of using the Factory design pattern is that it allows the creation of objects to be centralized in a single location, which can make the code more modular and easier to maintain.
- It also allows the implementation of object creation to be changed easily, which can make the design more flexible and extensible.
- Allows objects to be created without specifying their exact class, which can make the code more generic and reusable.

2.3. Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

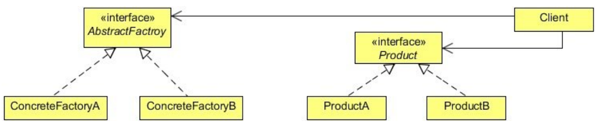

Figure 3. class_diagram_1

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of the multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- Useful when you want to create objects that are compatible with a certain application or framework, but you don't want to specify the concrete classes of the objects until runtime.

2.4. Builder

Allows for the creation of complex objects in a step-by-step manner. It separates the construction of an object from its representation, allowing for different representations to be created.

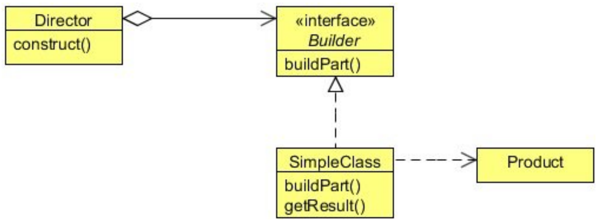

Figure 4. Builder UML.

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

2.5. Prototype

Allows for the creation of new objects by copying existing objects, rather than creating new objects from scratch.

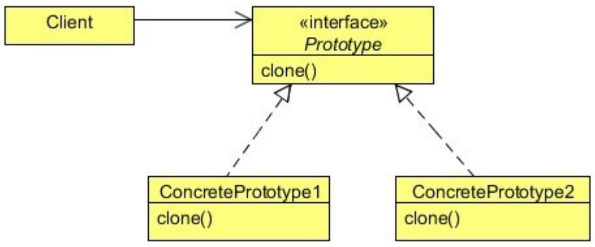

Figure 5. Prototype UML.

- Useful when creating complex objects or when the cost of creating a new object is high.
- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

3. Structural patterns

3.1. Adapter

Allows two incompatible interfaces to work together by wrapping an adapter class around one of the interfaces. This adapter class converts the interface of the adapted class into the interface that the client is expecting.

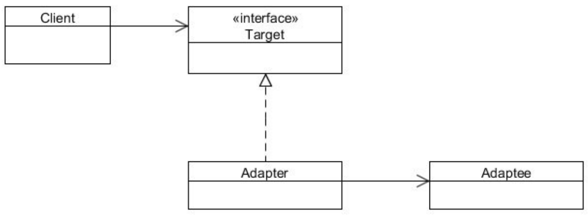

Figure 6. Figure 3

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.
- Possible to create a two-way adapter that can convert the calls in both directions.

3.2. Bridge

Allows for the separation of abstraction and implementation, so that the two can vary independently.

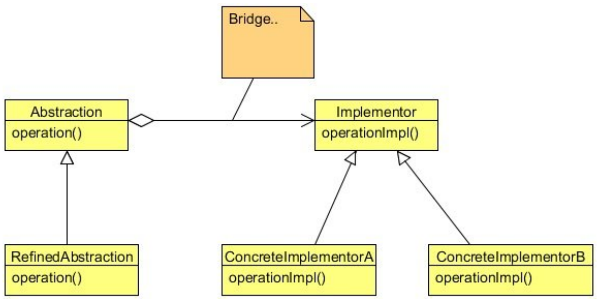

Figure 7. Bridge UML.

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

3.3. Composite

Allows objects to be treated as a single unit. It is used to compose objects into tree structures, and to create complex objects from simpler ones.

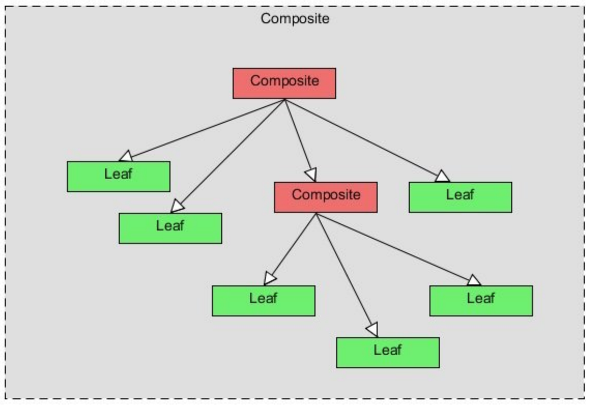

Figure 8. Composite UML.

- Hierarchical representations of objects are needed.
- Objects and compositions of objects should be treated uniformly.

3.4. Decorator

Allows for the dynamic addition of new behavior to an existing object without changing its structure.

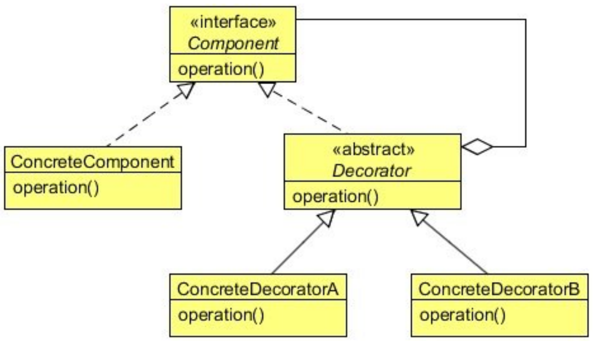

Figure 9. Decorator UML.

- It can be used to add new functionality to a class or to wrap an existing class with additional functionality.

3.5. Facade

Provides a simplified interface to a complex system.

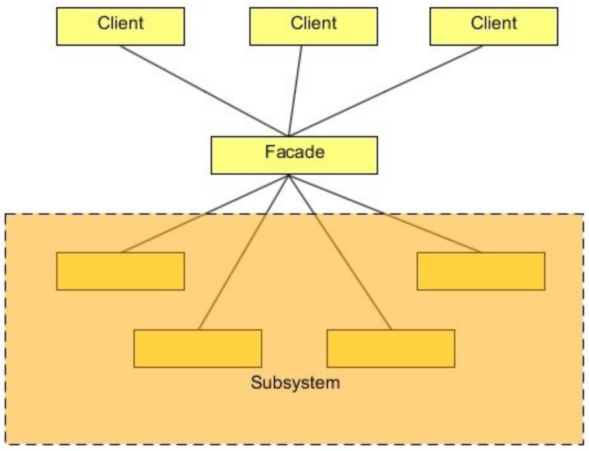

Figure 10. Facade UML.

- Useful when a system has a large number of interconnected classes or when a client only needs to access a limited number of the system's capabilities.
- It decouples the client from the complex subsystems and allows for easier maintenance and modification of the system.

3.6. Flyweight

Aims to minimize the use of memory by sharing common data among objects. This is done by creating a shared object that can be used by multiple objects, rather than each object having its own separate instance of the data.

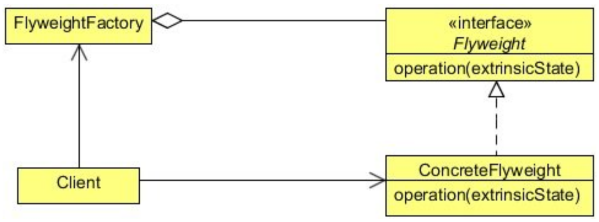

Figure 11. Flyweight UML.

- We can reduce the memory footprint of our application and improve its performance.
- Carefully consider the trade-off between the benefits of memory savings and the added complexity of implementing the pattern.

3.7. Proxy

Provides an intermediary object between a client and a real subject.

The proxy pattern can be used to:

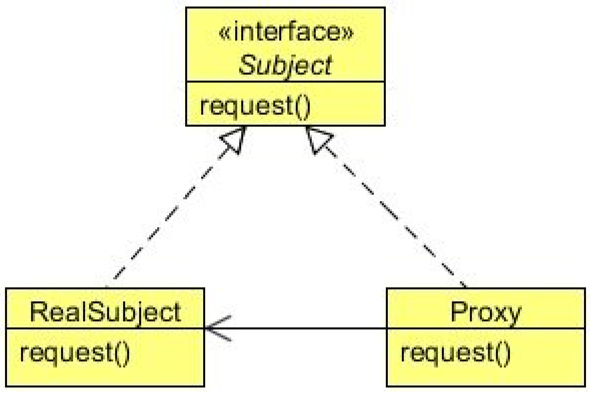

Figure 12. Proxy UML.

- Provide a placeholder for a potentially expensive or resource-intensive object. The proxy can be used to create the real object only when it is needed, rather than creating it upfront.
- Control access to the real subject. The proxy can be used to enforce access restrictions or implement authentication and authorization checks.
- Add additional functionality to the real subject. The proxy can be used to intercept requests to the real subject and perform additional tasks before or after forwarding the request.

4. Behavioral patterns

4.1. Chain of Responsibility

Allows an object to send a request to a chain of objects in order to handle the request.

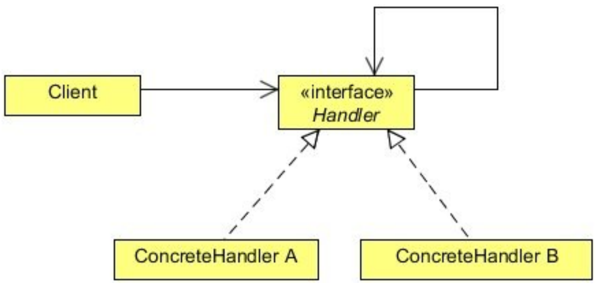

Figure 13. Chain of Responsibility UML.

- Useful for situations where multiple objects may be able to handle a request, and the specific object that should handle the request is not known in advance.
- Allows for the easy addition or removal of objects from the chain without disrupting the overall functionality.

4.2. Command

Allows for the encapsulation of a request as an object, which can then be passed to a receiver to be executed.

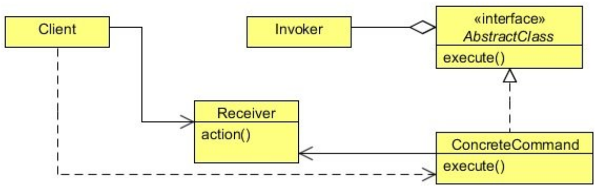

Figure 14. Command UML.

- Allows for the separation of the sender and receiver of a request.
- Ability to queue or log requests, and to support undo/redo functionality.

4.3. Iterator

Allows clients to access elements of an aggregate object sequentially without exposing the object's underlying representation.

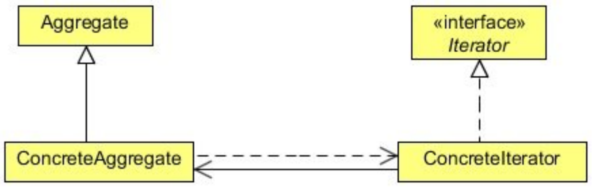

Figure 15. Iterator UML.

- Allows clients to traverse a collection of objects in a consistent, uniform manner, regardless of the specific implementation of the collection.

4.4. Mediator

Allows multiple objects to communicate with each other without knowing the details of their implementation.

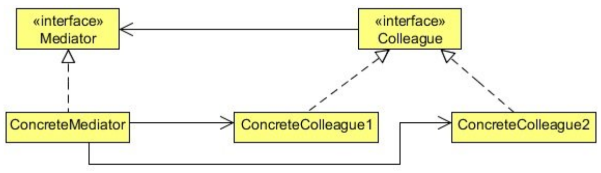

Figure 16. Mediator UML.

- It provides a central point of communication, known as the mediator, which acts as an intermediary between the objects.
- Useful in cases where there are a large number of objects that need to communicate with each other, as it reduces the complexity of the system by separating the communication logic from the objects themselves.

4.5. Observer

Allows an object (the subject) to notify a set of objects (the observers) when its state changes. The observer pattern is also known as the publish-subscribe pattern.

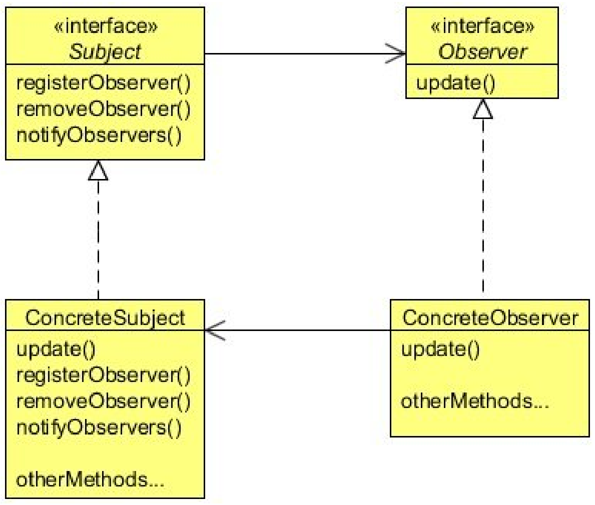

Figure 17. Observer UML.

- Useful when you want to ensure that various objects are kept in sync with each other, or when you want to be able to reuse subjects and observers independently of each other.

4.6. Strategy

Allows an object to change its behavior or strategy at runtime by switching to a different strategy object.

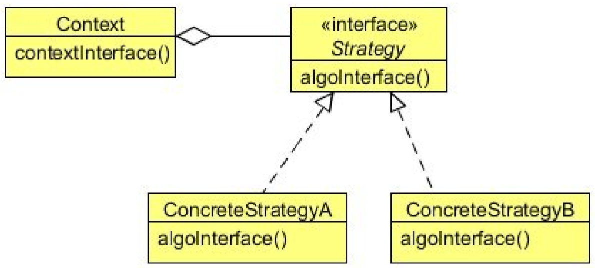

Figure 18. Strategy UML.

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

4.7. Template Method

Defines the steps of an algorithm and allows subclasses to override certain steps, while still preserving the overall structure of the algorithm.

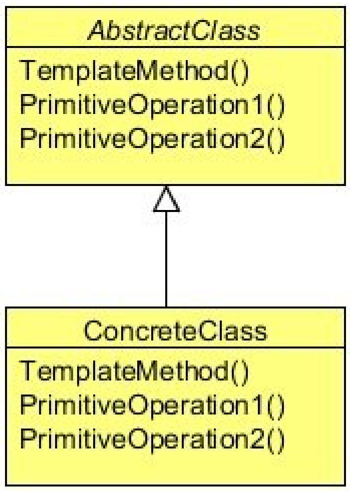

Figure 19. Template Method UML.

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most, or all subclasses need to implement the behavior.