Lecture Notes: SOLID Design Principles and Design Patterns

1. SOLID Design Principles

1.1 Introduction to SOLID Design Principles

The structural components of software engineering design are organised and arranged using design principles. The ways in which these design principles are implemented affect the expressive content and the working process of any software product. Design principles help designers reach a common understanding of architectural knowledge, assisting people with large-scale software engineering and assisting newcomers in avoiding traps and errors identified through previous experiences. The acronym SOLID is made from the names of five design principles that focus on improved code design, maintainability and extensibility. In his paper, *Design Principles and Design Patterns*, presented in 2000, Robert Martin (also known as Uncle Bob in the developer community) established these principles for the first time. Michael Feathers later named the principles and rearranged their order to create the acronym.

Uncle Bob argued that without a good design, an application may suffer from one of the following drawbacks:

- Rigidity: Everything is really set in stone. You cannot move or adjust something without influencing something else. Still, it is obvious what would break if you do alter anything.
- Fragility: Things are easy to move and adjust, however it is unclear what else could break as a result.
- Code is immobile, which implies that you cannot reuse it without duplicating or replicating it.
- When you make a change, everything falls apart, but you quickly put it back together and make your change operational . When someone else makes a change, the same problem arises.

In addition, you must have gone through a situation where you spend hours on end on a single bug because the code is not readable enough. Design principles recommend approaches that can help avoid these situations.

People may argue that even after applying design principles, there are chances of inclusion of bugs while writing code. It is equivalent to claiming that you can still drive badly even if you arrange your mirrors before leaving the house so you can see around and behind you. That is not an excuse for not adjusting your mirrors. Similarly, applying design principles is one of the many points that you must inculcate in your code to design a better product.

You learnt the following 5 design principles in this session:

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

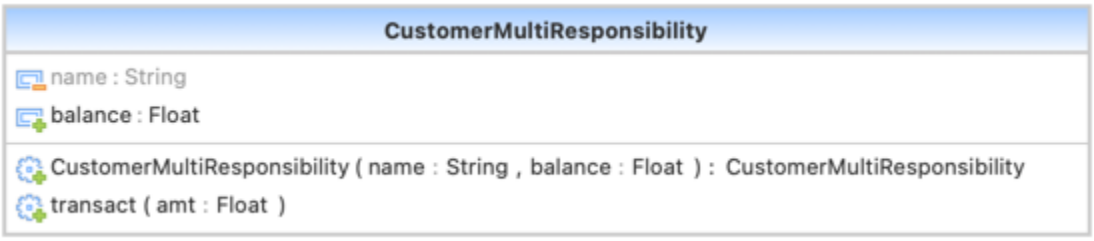
1.2 Understanding Single Responsibility Principle (SRP)

SRP can be summarised as follows:

- As the name of the principle suggests, one class should have only one responsibility.
- Do not try to accomplish too much in the same class or method. This notion is broken if you implement reading from a text file, writing to a database, making a service call and writing to a log all in the same procedure.
- If your test suite has a lot of tests for a single class or method, for example, if a single method writes to the database and outputs a text file, this is a code smell for breaking the SRP.

You further learnt the concept of SRP through the example of the database of a customer at a bank. The UML of the example before and after the application of the pattern are as shown below:

- Before applying SRP to the customer database at a bank



- After applying SRP to the customer database at a bank

You can see how the responsibilities were divided into two different classes instead of only one.
The code for this example can be found here:
<code>

1.3 Understanding Open/Closed Principle (OCP)

Open/Closed Principle can be summarised as follows:

- OCP states that classes should be open for extension, but closed for modification.
- You should be able to extend rather than modify an existing class to add or change functionality, that is, inherit from the class in a subclass that extends the behaviour of the superclass with the new functionality.
- If you want the class to perform additional functions, the best approach is to add to the existing functions rather than changing them.

You further learnt the concept of OCP through the example of the food order class. The UML of the example before and after the application of the pattern are as shown below:

- Before applying OCP to the food ordering class

- After applying OCP to the food ordering class

You can see how functionalities are added to the existing class diagram and not modified.

The code for this example can be found here:
<code>

1.4 Understanding Liskov Substitution Principle (LSP)

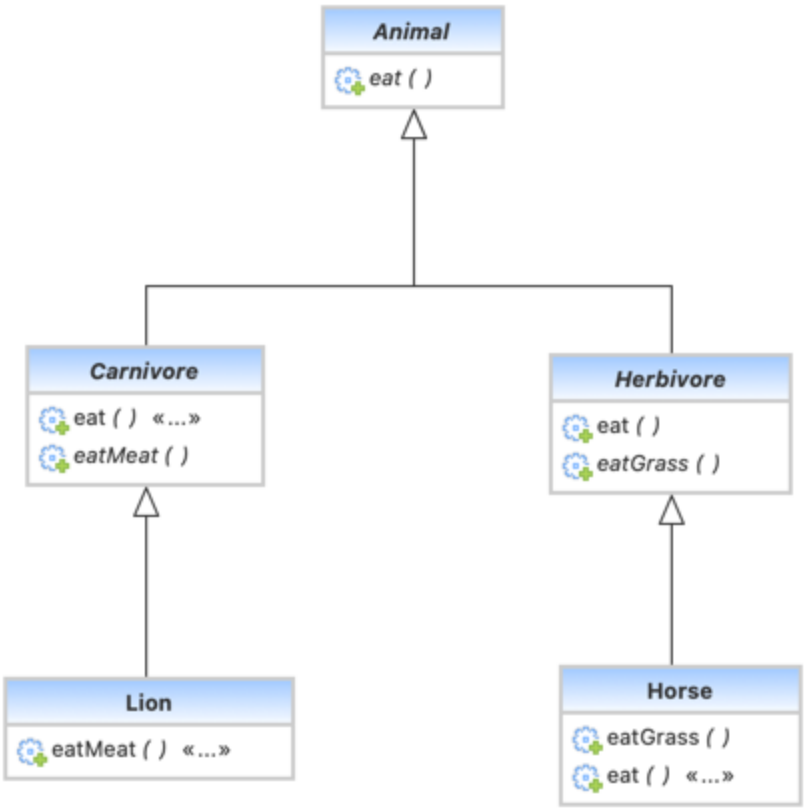
Liskov Substitution Principle can be summarised as follows:

- LSP states that subclasses should be substitutable for their base classes.
- If A is a subtype of B, then B objects in a program can be substituted with A objects without affecting any of the program's desired qualities.
- Bugs can occur when a child class is unable to perform the same behaviour as its parent class.
- The child class should be able to process the same requests as the parent class and deliver the same result or a result of the same type.

You further learnt the concept of LSP through the example of the animal hierarchy class. The UML of the example before and after the application of the pattern are as shown below:

- Before applying LSP to the animal hierarchy class

- After applying LSP to the animal hierarchy class



1.5 Understanding Interface Segregation Principle (ISP)

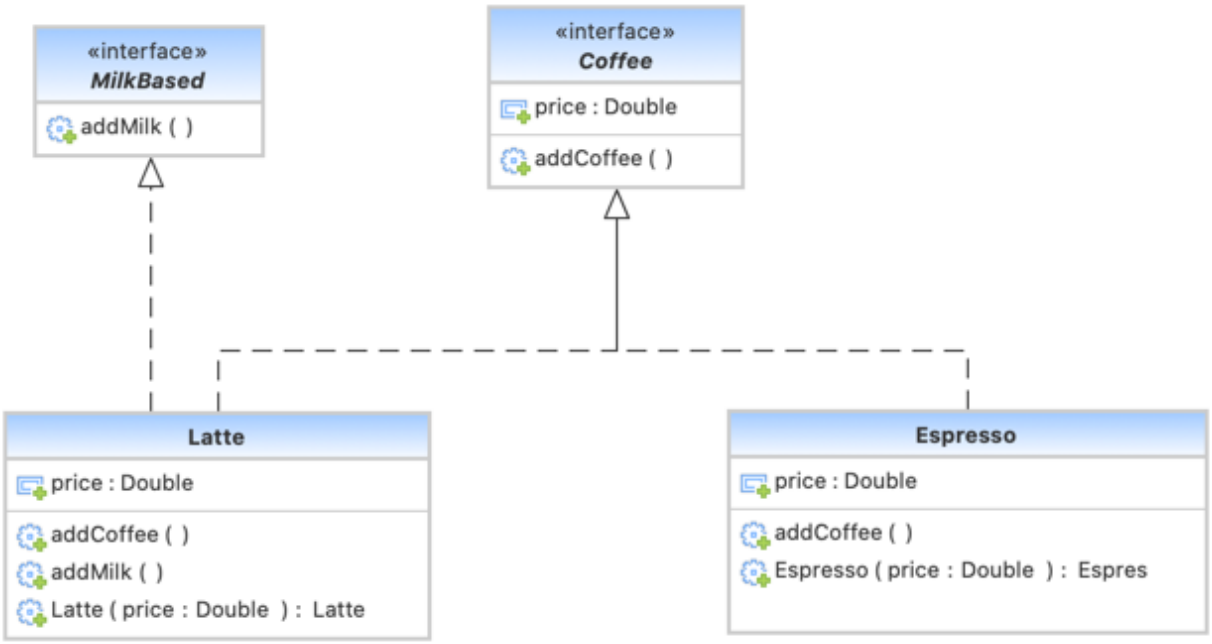
Interface Segregation Principle can be summarised as follows:

- ISP states that many client-specific interfaces are better than one general-purpose interface.
- Interfaces should be split down into small groups of related members that are implemented by all the classes that implement the interface.
- When you update an interface, you must also update everything that uses it. Since a single large interface is implemented by more classes rather than a series of smaller interfaces, changes to a large interface typically necessitate modifications to a larger number of classes.

You further learnt the concept of ISP through the example of the coffee maker class. The UML of the example before and after the application of the pattern are as shown below:

- Before applying ISP to the coffee maker class

- After applying ISP to the coffee maker class



1.6 Understanding Dependency Inversion Principle (DIP)

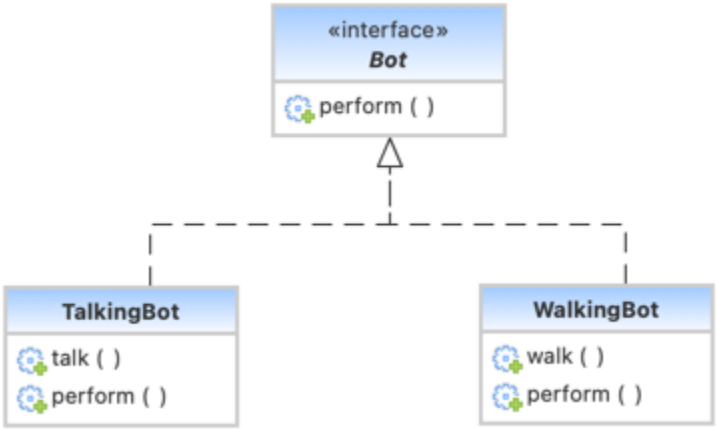
Dependency Inversion Principle can be summarised as follows:

- DIP states that you must depend upon abstractions. Do not depend upon concretions.
- According to this principle, a class should not be fused with the tool it uses to perform an operation. Rather, it should be merged with the tool's interface to allow it to connect to the class.
- This principle further specifies that neither the class nor the interface should be aware of how the tool operates. However, the tool must adhere to the interface's specifications.

You further learnt the concept of DIP through the example of the walking and talking robot class. The UML of the example before and after the application of the pattern are as shown below:

- Before applying DIP to the talking and walking robot class

- After applying DIP to the talking and walking robot class



Here, you can see how the responsibilities were divided into two different classes instead of only one.

2. Introduction to Design Patterns

2.1 Understanding Design Patterns

In the previous session, you learnt some of the good practices that you can apply in your applications using the SOLID Design Principles. In this session, you will learn how some of these recommended design principles can be implemented using the concept of design patterns.
Briefly, design patterns can be considered as conventional answers to common software design issues. Each pattern is similar to a blueprint that you can modify to tackle a specific design problem in your code.
You may ask what is the difference between an algorithm and a design pattern. Both notions explain typical solutions to well-known issues, however patterns and algorithms are sometimes misconstrued. A pattern is a higher-level description of a solution, whereas an algorithm always describes a clear set of activities that can be performed to attain a goal. When the same pattern is applied to two separate programs, the code may differ.

But, why do you even need to understand the different design patterns? The answer to this question is as follows:

- Patterns are tried and tested solutions to a specific problem that has previously occurred repeatedly.
- You can use these design patterns to communicate with your teammates more efficiently as it gives a common language that you have agreed upon.

The next step is to understand the classification of design patterns. They are broadly categorised as follows:

1. **Creational design patterns**:
    a. These patterns define HOW classes and objects will be created. Different classes of objects will follow different mechanisms to create objects.
    b. Another major function of the creational design patterns is to hide the actual implementation of the classes from their usage.
    c. These patterns are useful if the user is not primarily concerned with the actual implementation of the classes and is only focussed on the final created product.
2. **Structural design patterns**:
    a. These patterns define the relationships among different classes. They are mainly concerned with how the different classes and subclasses are

organised amongst each other and how classes and objects are composed to form larger objects and structures.

b. They differ from the creational patterns in the way that while the creational patterns focus on how classes and objects are created, the structural patterns focus on how these different classes are related to each other.

3. **Behavioural design patterns**:
   a. These patterns help define the relationship between different classes and objects and how these classes and objects will communicate with each other.
   b. These patterns are concerned with how different responsibilities are shared amongst different classes.
   c. They help define and streamline the complex flow of information among different classes.

3. Creational Design Patterns

3.1 Understanding Factory Design Pattern

The factory design pattern can be summarised as follows:

- The factory design pattern is a creational design pattern that provides an interface to produce objects in a superclass while allowing the subclasses to choose the type of objects created.
- In this pattern, you use a special Factory method instead of using a new operator to generate an object.
- The new operator is still used to generate the objects, but it is now invoked from within the Factory function.
- The objects returned by a Factory method are known as a product.

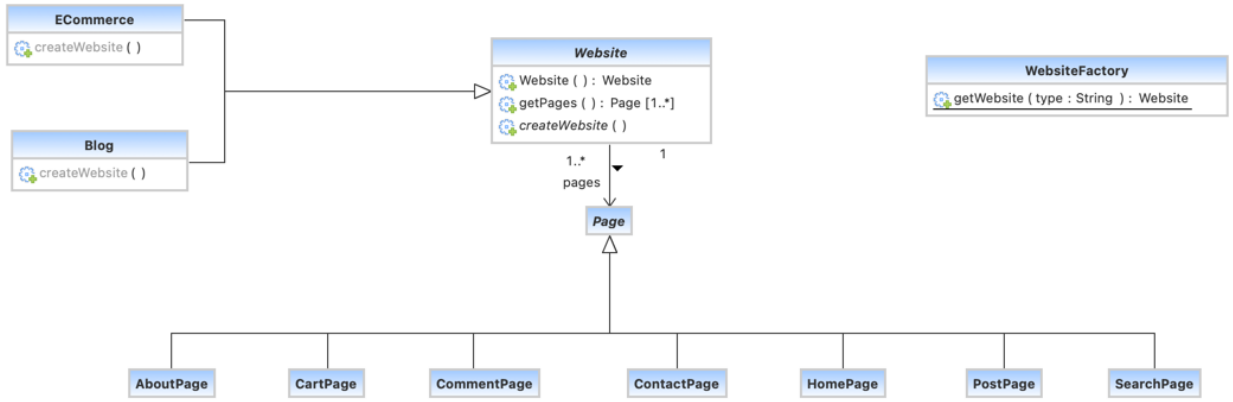
Now that you know what factory pattern is, let's understand where it is applicable:

- When you do not know the exact types and dependencies of the objects your code should interact with ahead of time, use the factory pattern.
- When you want to give your users the libraries or frameworks to extend the application's internal components, use the factory method.
- When you wish to save system resources by reusing the existing objects rather than reconstructing them each time they are needed, use the factory method.

You further learnt the concept of factory design pattern through the example of e-commerce. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the factory pattern

- Application design after applying the factory pattern



3.2 Understanding Builder Design Pattern

The builder design pattern can be summarised as follows:

- The builder design pattern allows you to build complicated objects in stages. Using the same building code, you can create different types and representations of the same object.

- The builder pattern recommends separating the object creation code from its own class and moving it to distinct objects known as Builders.

Now that you know what builder pattern is, let's take a look at where it is applicable:

- To get rid of a 'telescopic constructor', use the builder pattern.
- When you want your code to be able to produce several representations of a product, utilise the builder pattern.
- You can build composite trees and other complicated objects using the builder pattern.
- During the construction process, a builder does not expose the unfinished result. This prohibits the client from retrieving a partial result.

You further learnt the concept of builder design pattern through the example of the Pizza dressing. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the builder pattern

- Application design after applying the builder pattern

3.3 Understanding Prototype Design Pattern

The prototype design pattern can be summarised as follows:

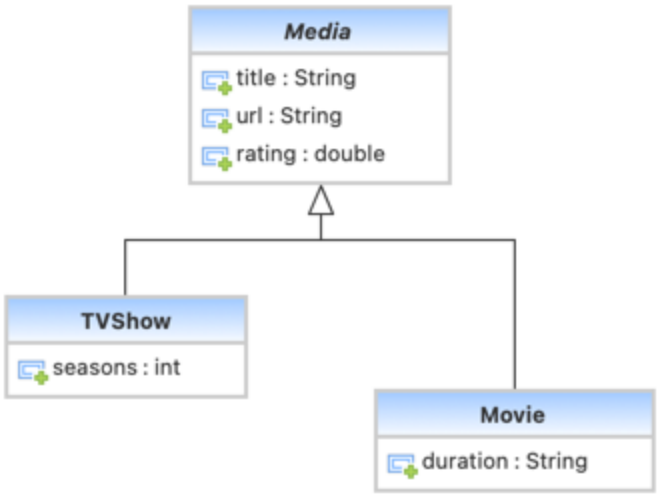
- The prototype design pattern allows you to clone the existing objects in your code without having to rely on their classes.
- The problem with the implementation of this pattern is that if the fields of an object are private and not viewable from outside the object, that object cannot be cloned through this process.
- The cloning operation is delegated to objects using the prototype pattern.
- This pattern declares a standard interface for all cloning-capable objects.
- This interface allows you to clone an object without having to tie your code to the object's class. Typically, such an interface just has one cloning method.

Now that you know what prototype pattern is, let's take a look at where it is applicable:

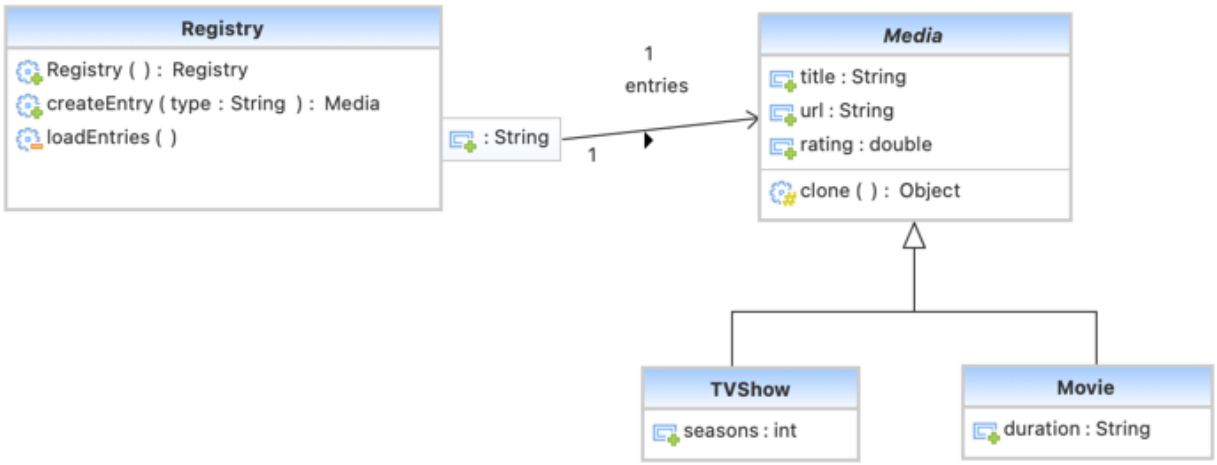
- When you do not want your code to be dependent on the concrete classes of the objects that you are copying, use the prototype pattern.
- When you need to limit the number of subclasses that merely differ in terms of how they initialise their objects, use the prototype pattern.
- Instead of creating a subclass that matches a specific configuration, the client can simply find and clone an acceptable prototype.

You further learnt the concept of prototype design pattern through the example of Media. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the prototype pattern



- Application design after applying the prototype pattern



3.4 Understanding Singleton Design Pattern

The singleton design pattern can be summarised as follows:

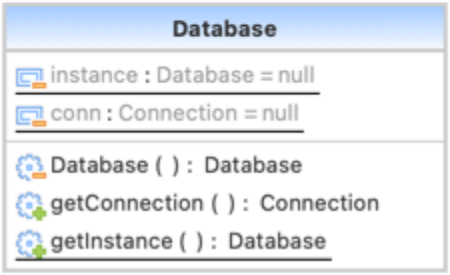
- The singleton design pattern ensures that a class has only one instance while also giving a global access point to that instance.
- To prevent other objects from using the new operator with the Singleton class, make the default constructor private.
- The singleton pattern is well-exemplified by the structure of the government. There can only be one official government in a country. The title 'The Government of X' is a global point of access that identifies the group of people in command, regardless of the personal identities of the individuals that compose the governments.

Now that you know what singleton pattern is, let's take a look at where it is applicable:

- When a class in your software should only have one instance that is available to all clients, such as a single database item shared by all components of the programme, use the singleton pattern.
- When you require tighter control over global variables, use the singleton design.
- One of the challenges of the singleton pattern is that it violates the Single Responsibility Principle (SRP), as the latter involves performing two tasks instead of only one.

You further learnt the concept of singleton design pattern through the example of database connection. The UML of the example before and after the application of the pattern are as shown below:

- Application design after applying the singleton pattern



3.5 Understanding Abstract Factory Design Pattern

The abstract factory design pattern can be summarised as follows:

- The abstract factory design pattern allows you to create families of linked items without having to declare their particular classes.
- The first suggestion made by the abstract factory pattern is to create interfaces for each individual product in the product family.
- Each class in a well-designed software is solely responsible for one thing. Isolating a class's factory methods into a stand-alone factory class or a full-blown abstract

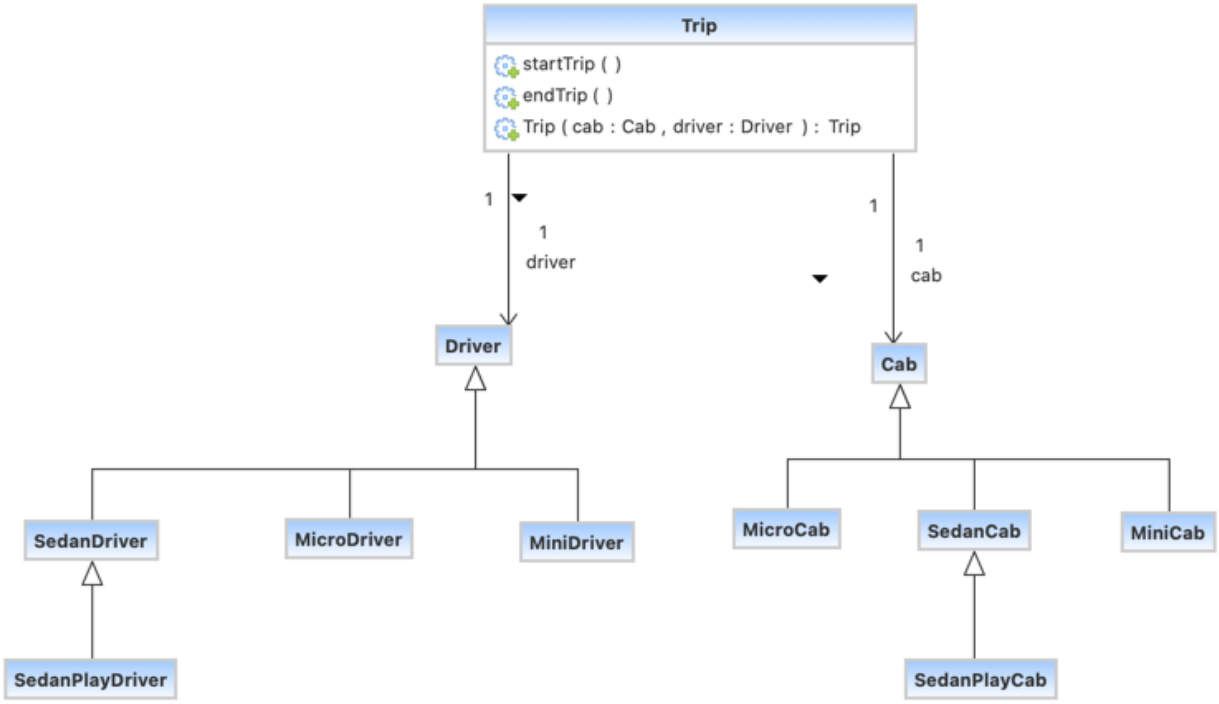
factory implementation when it deals with various product types might be useful in many cases.

Now that you know what abstract factory pattern is, let's take a look at where it is applicable:

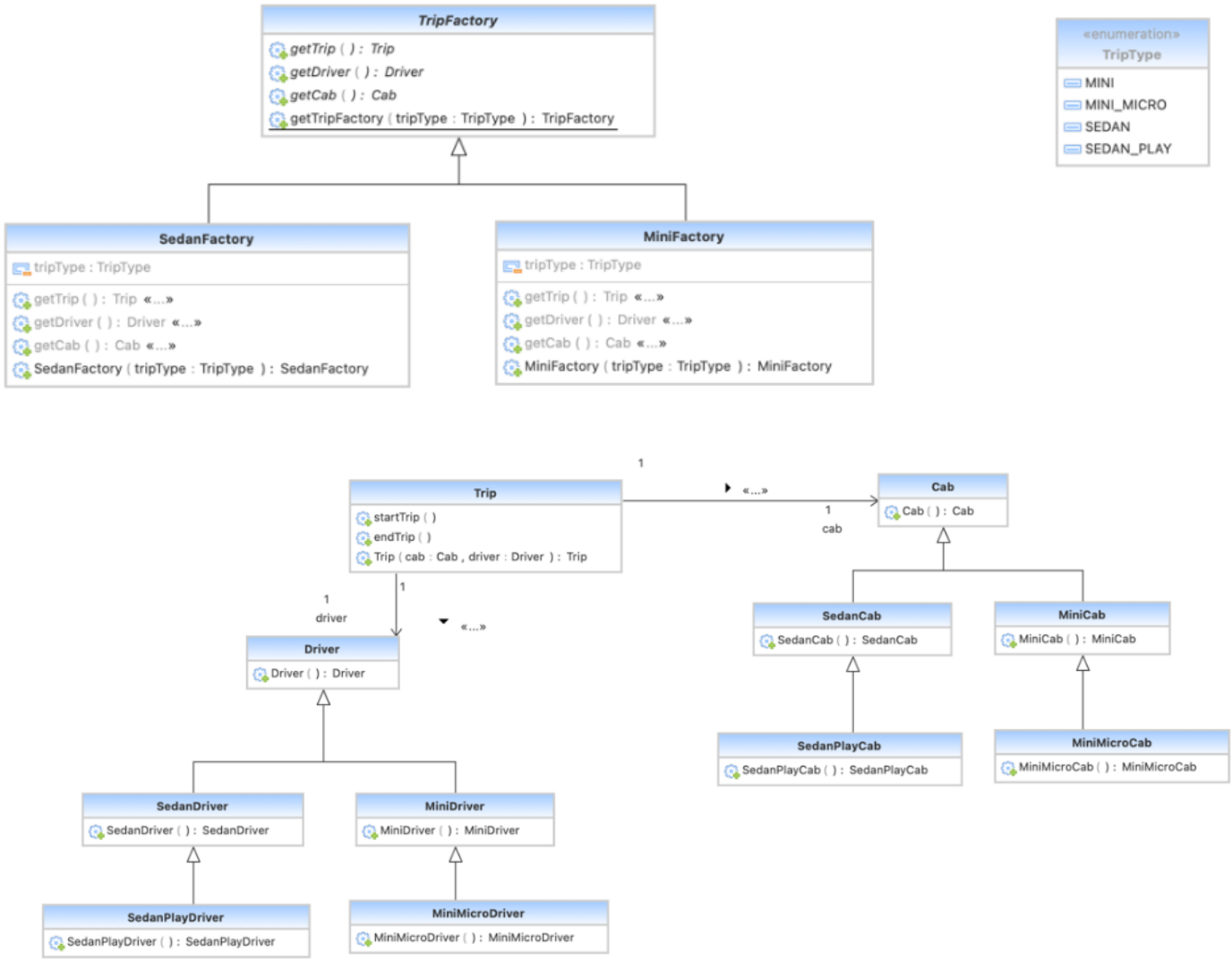
- When your code needs to deal with multiple families of related products, but you do not want the code to be dependent on the concrete classes of those products either because you do not know what they are or because you want to allow for future extensibility, use the abstract factory pattern.

You further learnt the concept of abstract factory design pattern through the example of cab rides. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the abstract factory pattern



- Application design after applying the abstract factory pattern



4. Structural Design Patterns

4.1 Understanding Adapter Design Pattern

The adapter design pattern can be summarised as follows:

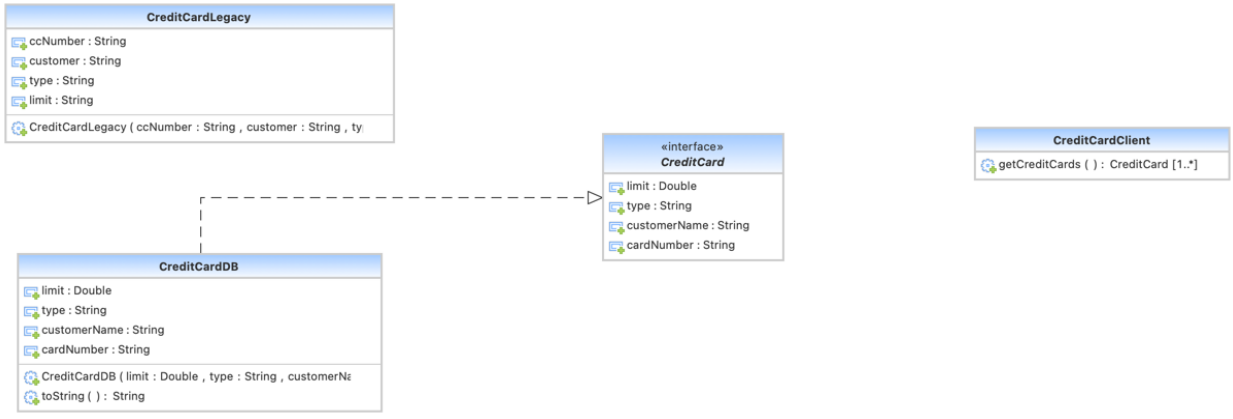
- The adapter design pattern is a structural design pattern that allows items with mismatched interfaces to work together.
- You can make your own Adapter, which is a unique object that changes an object's interface so that it can be understood by another object.
- Adapters can let objects with different interfaces communicate as well as translate their data as needed.

Now that we know what adapter pattern is, let's take a look at it is applicable:

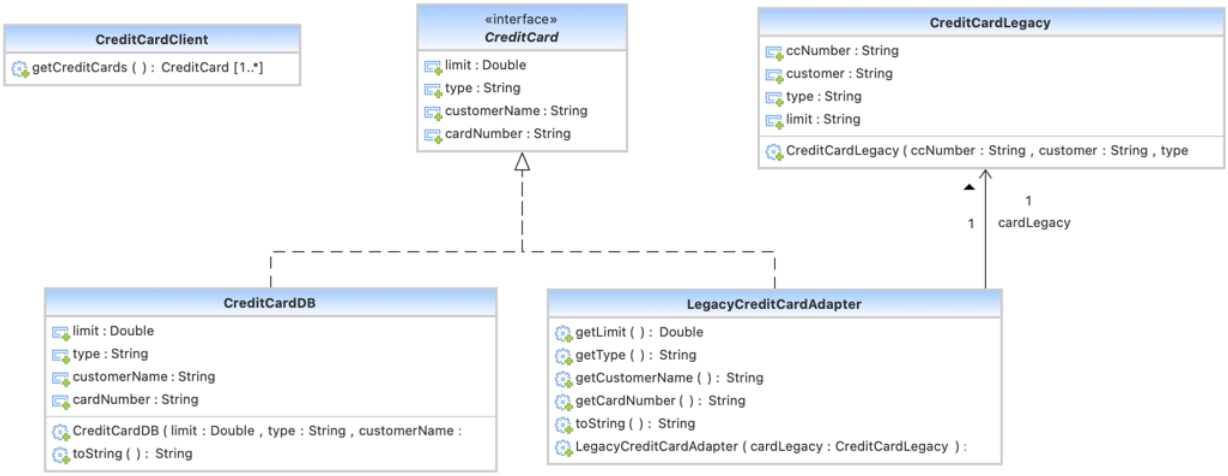
- When you wish to use an existing class but its interface is not consistent with the rest of your code, use the adapter pattern.
- When you wish to reuse numerous existing subclasses that lack some common functionality that cannot be added to the superclass, use the adapter pattern.
- The adapter pattern should be used by clients that use their own interface. This allows you to change or extend the Adapters without having to worry about the client code.

You further learnt the concept of adaptor design pattern through the example of credit cards. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the adapter pattern



- Application design after applying the adapter pattern



4.2 Understanding Bridge Design Pattern

The bridge design pattern can be summarised as follows:

- The bridge pattern is a structural design pattern that allows you to divide a large class or a group of closely related classes into two different hierarchies—abstraction and implementation—that can be developed independently.
- A high-level control layer for an entity is termed as the abstraction layer (also known as interface). This layer is not designed to perform any function by itself. The work should be delegated to the implementation layer (also called platform).
- Making even a little update to a monolithic codebase is difficult since you must possess a thorough understanding of the entire system. It is considerably easier to make changes to smaller, well-defined modules.

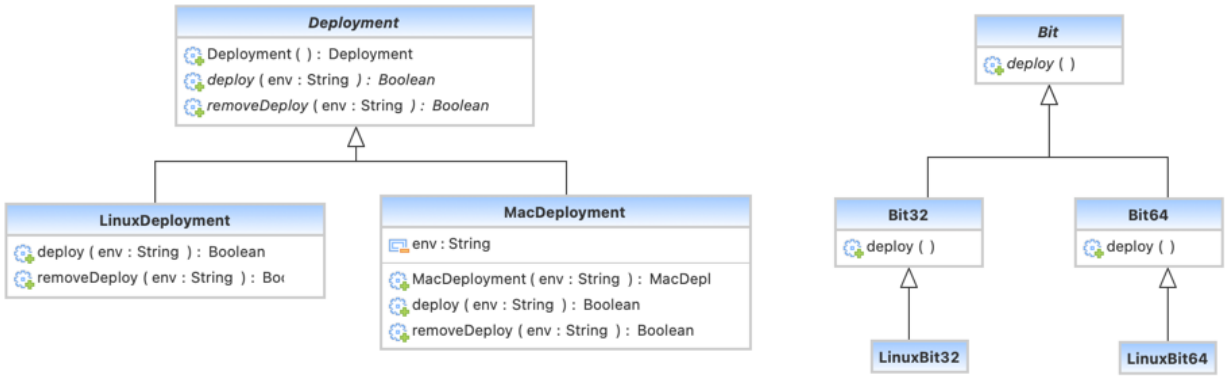
Now that you know what the bridge pattern is, let's take a look at where it is applicable:

- When you want to separate and arrange a monolithic class with numerous variants of certain functionality, use the bridge design pattern (for example, if a class can work with various database servers).

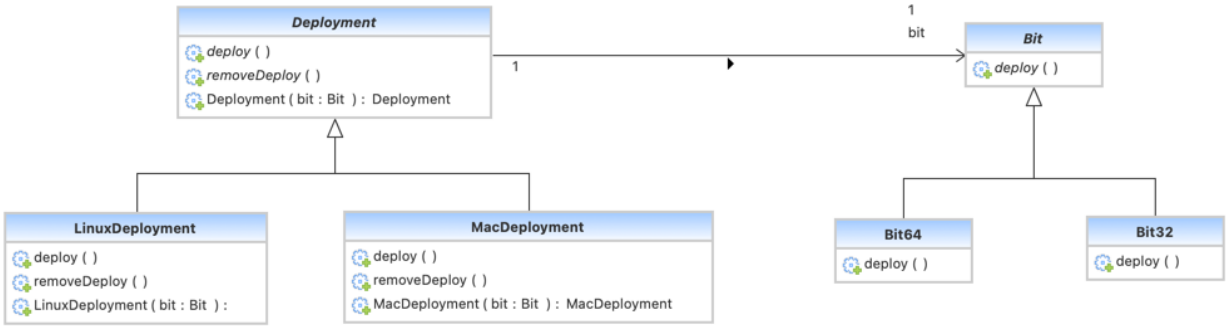
- When you need to extend a class in numerous orthogonal (independent) dimensions, use the bridge pattern.
- If you need to swap implementations in the middle of a project, use the bridge pattern.

You further learnt the concept of bridge design pattern through the example of code deployment. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the bridge pattern



- Application design after applying the bridge pattern



4.3 Understanding Decorator Design Pattern

The decorator design pattern can be summarised as follows:

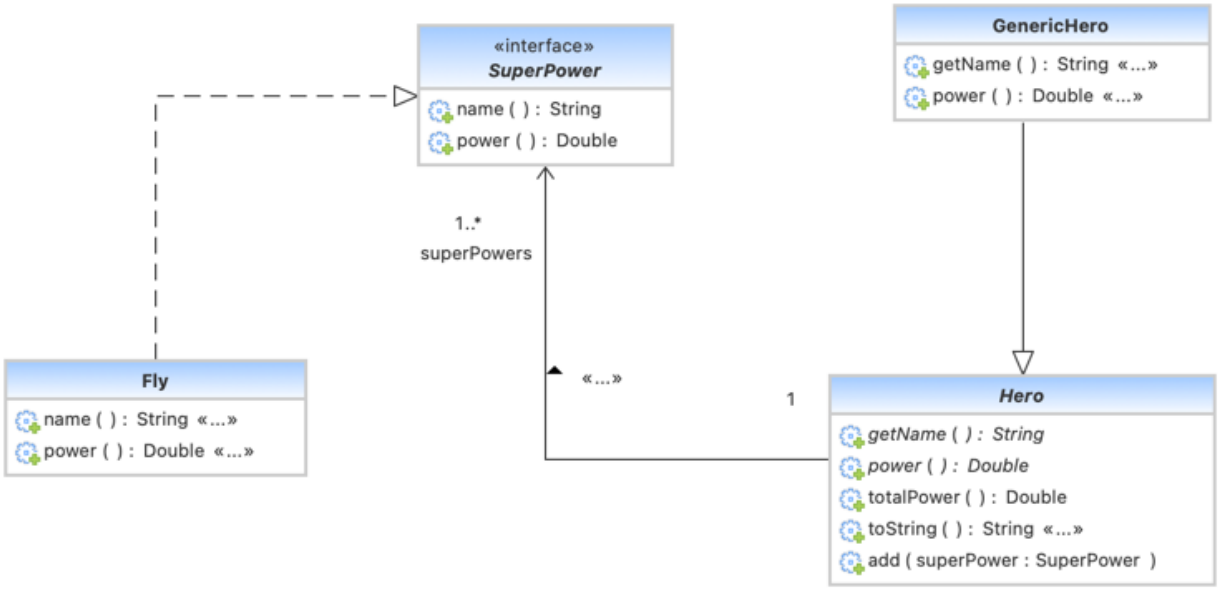
- Decorator design pattern allows you to attach additional behaviours to objects by enclosing them in special wrapper objects that contain the behaviours.
- The alternate name for the decorator pattern is 'Wrapper', which clearly describes the pattern's core notion.
- A wrapper is an object that can be associated with a specific target.
- All requests are delegated to the wrapper, which has the same set of methods as the target.
- When does a simple wrapper start being a true decorator? The wrapper, as previously stated, implements the same interface as the wrapped object. As a result, these objects appear to be identical to the client. Allow any object that follows that interface to be referenced in the wrapper's reference field. This allows you to wrap an object in multiple wrappers and apply the wrapper's aggregate functionality to the object.

Now that we know what the decorator pattern is, let's take a look at where it is applicable:

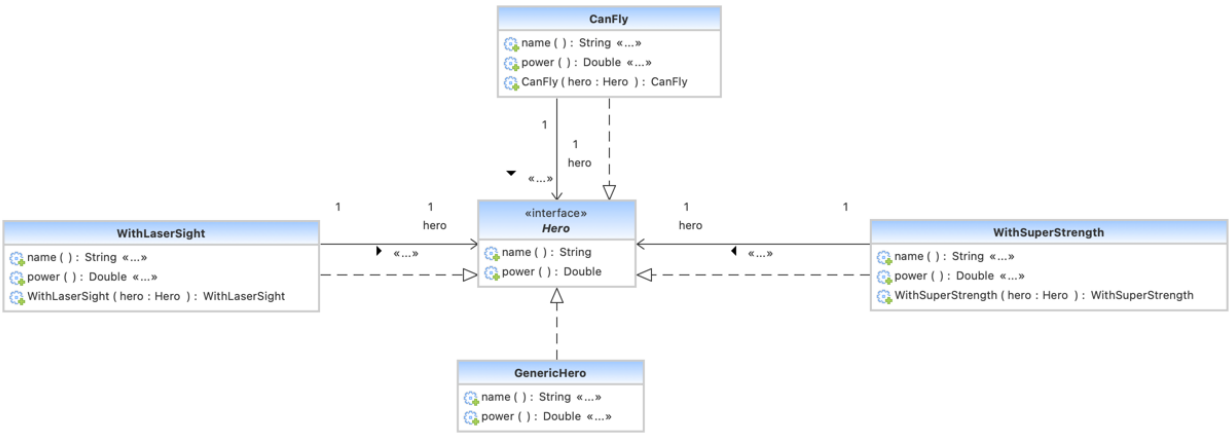
- When you need to be able to add extra behaviours to objects at runtime without disturbing the code that uses them, utilise the decorator approach.
- When it is inconvenient or impossible to extend an object's behaviour using inheritance, use the decorator pattern.

You further learnt the concept of decorator design pattern through the example of a superhero. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the decorator pattern



- Application design after applying the decorator pattern



4.4 Understanding Composite Design Pattern

The composite design pattern can be summarised as follows:

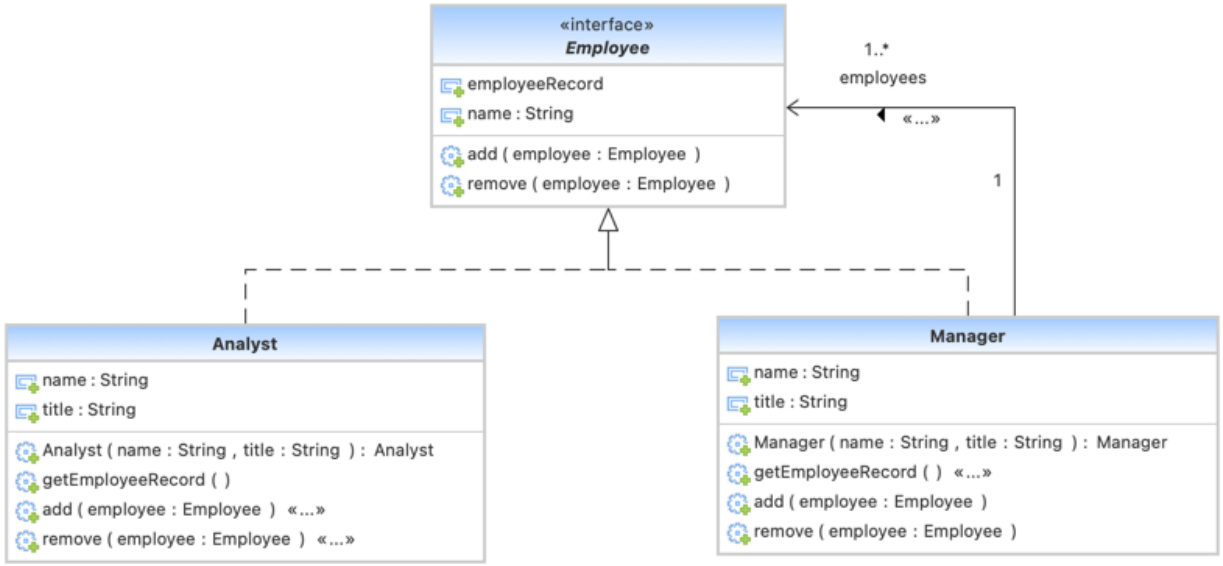
- The composite pattern is a structural design pattern that allows you to group elements into tree structures and manipulate them as if they were distinct objects.
- It is not necessary to know if an object is a simple product or a complex box. Using the shared interface, you may treat them all as the same. When you call a method, the request is passed down the tree by the objects themselves.

Now that you know what the composite pattern is, let's take a look at where it is applicable:

- When you need to create a tree-like object structure, use the composite pattern.
- When you want the client code to treat both simple and complicated items the same way, use the composite pattern.

You further learnt the concept of composite design pattern through the example of the employee structure. The UML of the example before and after the application of the pattern are as shown below:

- Application design after applying the composite pattern



4.5 Understanding Facade Design Pattern

The facade design pattern can be summarised as follows:

- The facade pattern is a structural design pattern that simplifies the interface to a library, framework or any other complex set of classes.
- A Facade is a class that gives a straightforward interface to a complicated subsystem with numerous moving pieces. Compared to direct interaction with the subsystem, a Facade might have limited capabilities. It does, however, only include the features that clients really require.

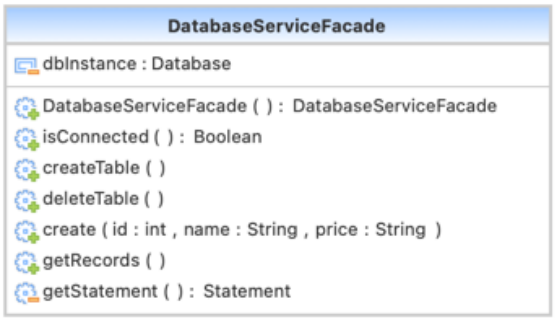
Now that you know what the facade pattern is, let's take a look at where it is applicable:

- When you require a simple but limited interface for a complicated subsystem, use the facade pattern.
- When you want to divide a subsystem into layers, use the facade design pattern.
- To establish entry points to each level of a subsystem, create Facades. By implementing communication among the various subsystems solely through Facades, you can reduce entanglement.

You further learnt the concept of facade design pattern through the example of database service. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the facade pattern

- Application design after applying the facade pattern



4.6 Understanding Proxy Design Pattern

The proxy design pattern can be summarised as follows:

- The proxy pattern is a structural design pattern that allows you to use another object as a substitute or placeholder.
- A Proxy manages access to the actual object, allowing you to do something before or after the request reaches the real object.
- Proxy allows you to run something before or after the primary logic of the class without modifying the class. Proxy can be supplied to any client that expects a real service object because it implements the same interface as the original class.

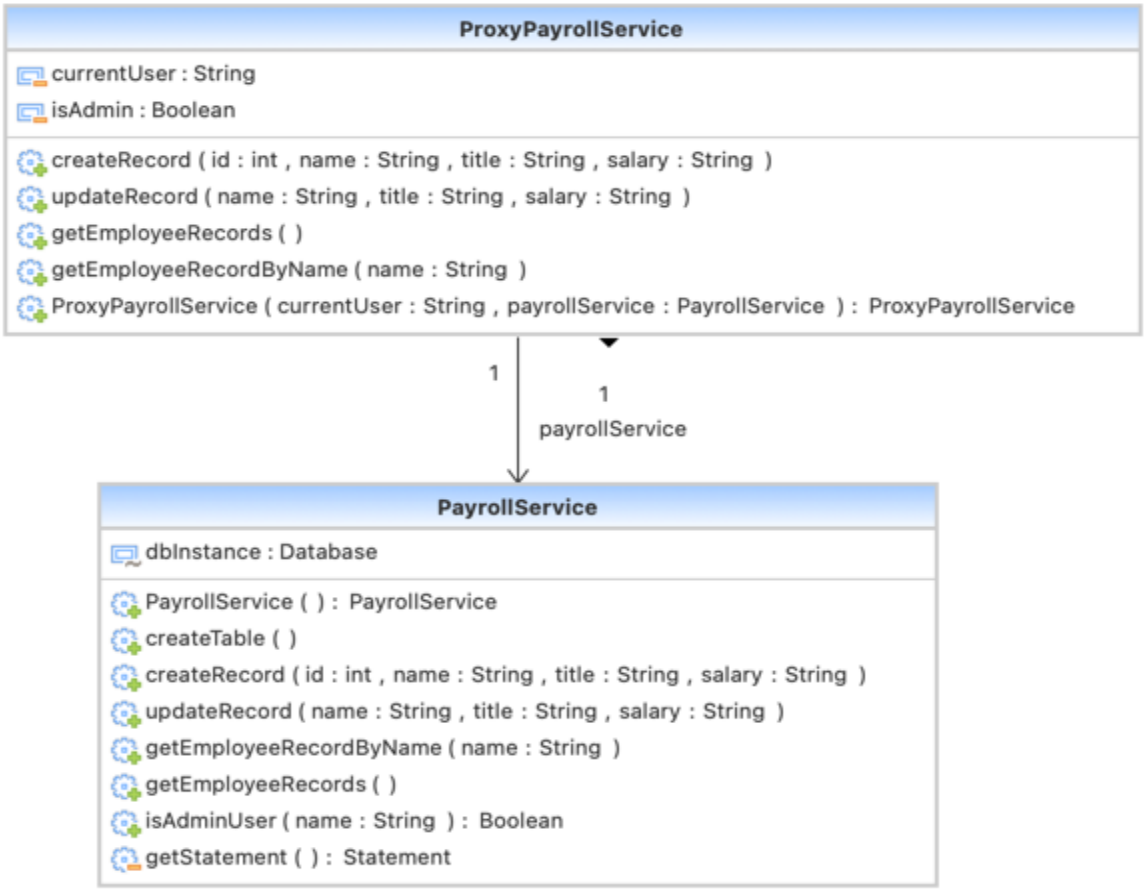
Now that you know what the proxy pattern is, let's take a look at where it is applicable:

- This pattern is used in cases where lazy initialisation is required (virtual Proxy). This case occurs when you have a heavyweight service object that wastes system resources by running all the time, even if you only use it occasionally.
- This pattern is used in cases where a distant service on a local machine needs to be executed (remote Proxy). This case occurs when the service object is on a remote server.
- This pattern is used in cases where requests need to be logged (logging Proxy). When you wish to preserve a history of requests to the service object, use this method.
- This pattern is used in cases where request results need to be cached (caching Proxy). This case occurs when you need to cache client request results and manage the cache's life cycle, especially if the results are enormous.

You further learnt the concept of proxy design pattern through the example of payroll service. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the proxy pattern

- Application design after applying the proxy pattern



4.7 Understanding Front Controller Design Pattern

The front controller design pattern can be summarised as follows:

- In the front controller design pattern, all requests for a resource in an application will be handled by a single handler before being forwarded to the proper handler for that type of request.
- Other helpers may be used by the Front controller to complete the dispatching procedure. Front controller comprises the following components:
    - Controller: The controller is the system's first point of contact for all requests. It can delegate the completion of a user's authentication and authorisation to a helper, along with the start of the contact retrieval.

- View: A view is a representation of and display of information for the client. A model's data is retrieved by the view. Views, in turn, are supported by helpers, who encapsulate and adapt the underlying data model.
- Dispatcher: A dispatcher is in charge of view management and navigation, selecting the next view that would be shown to the user and providing a mechanism for vectoring control to the appropriate resource.
- Helper: A helper assists a view or the controller in completing its processing. Thus, helpers are responsible for a variety of tasks, including obtaining the data necessary for the view and storing the intermediate model (in this case, the helper is referred to as a value bean).

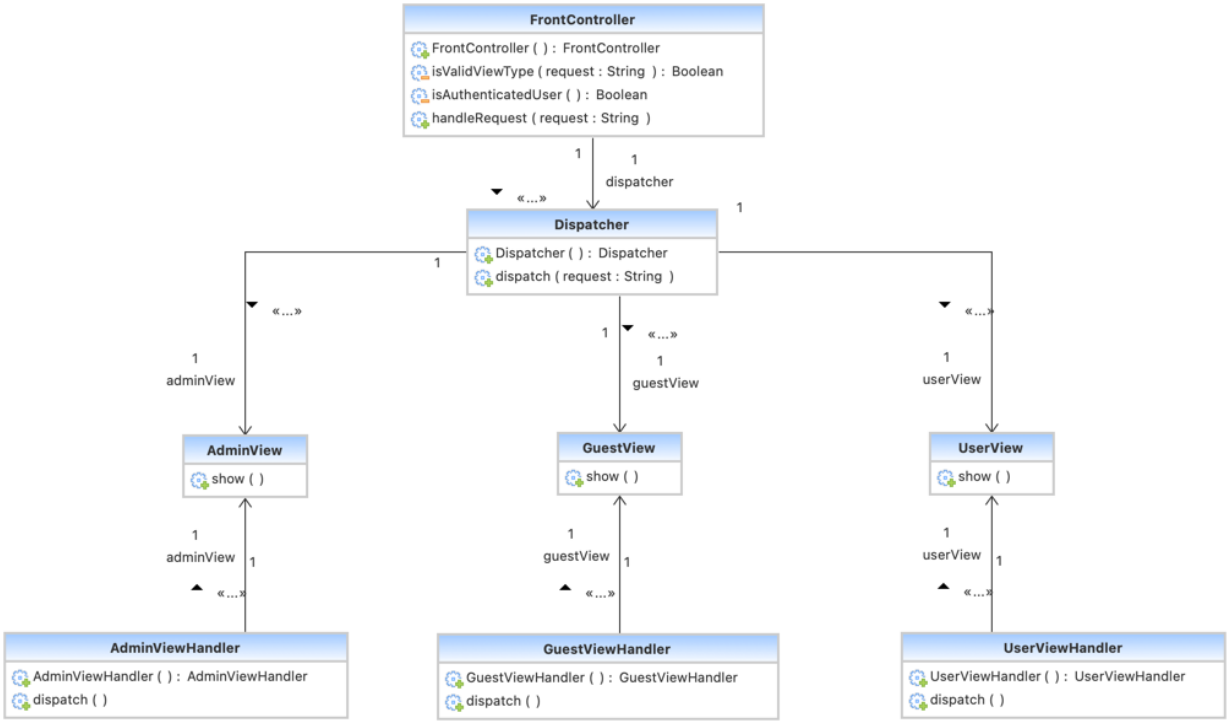
Now that you know what the front controller pattern is, let's try to list down the various advantages of using this pattern:

- All requests to the web application are handled by the Front Controller, which is centralised. This centralised control architecture, which avoids the use of many controllers, is ideal for enforcing application-wide policies, such as user tracking and security.
- When a new request is received, a new command object is created. Command objects are not designed to be thread-safe. As a result, it will be safe to use them in command classes. When threading issues accumulate, thread safety is not guaranteed. However, the codes that act with the command are thread-safe.

You further learnt the concept of front controller design pattern through the example of a user. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the front controller pattern

- Application design after applying the front controller pattern



5. Behavioral Design Patterns

5.1 Understanding Chain of Responsibility Design Pattern

The chain of responsibility design pattern can be summarised as follows:

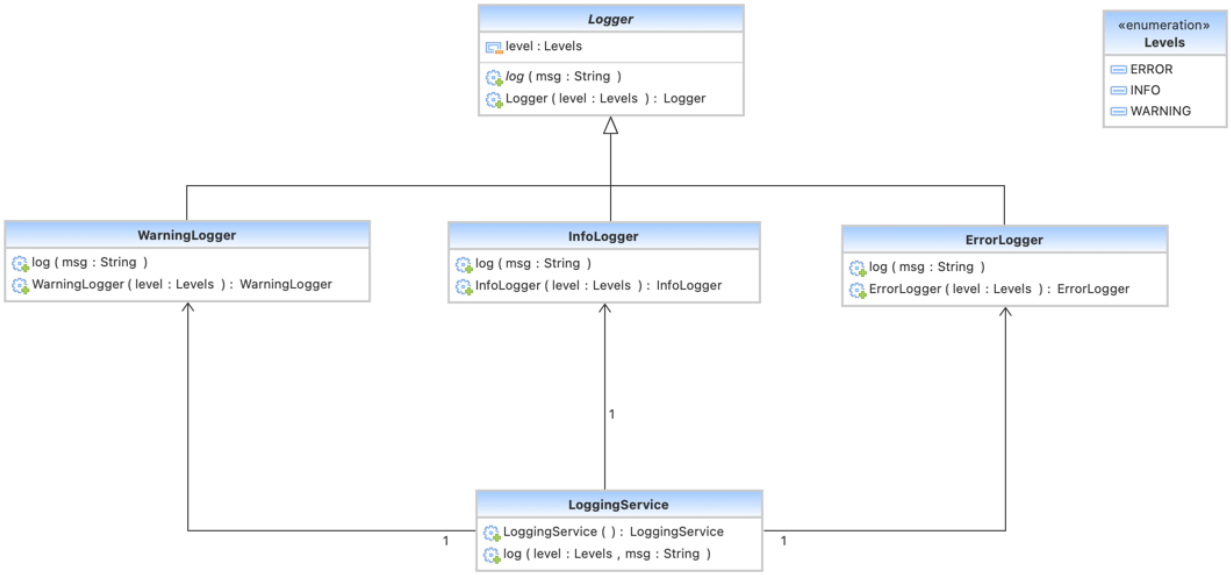
- The chain of responsibility pattern allows you to send requests through a chain of handlers. When a request is received, each handler determines whether to process it or send it on to the next handler in the chain.
- The chain of responsibility pattern, like many other behaviour design patterns, relies on the transformation of particular traits into stand-alone objects known as handlers.
- Each check should be moved to its own class with a single method to perform the check. This method receives the request, along with its data, as an argument.
- The finest feature of this pattern is that a handler can choose not to forward the request farther down the chain, thereby stopping all processing.

Now that you know what the chain of responsibility pattern is, let's take a look at where it is applicable:

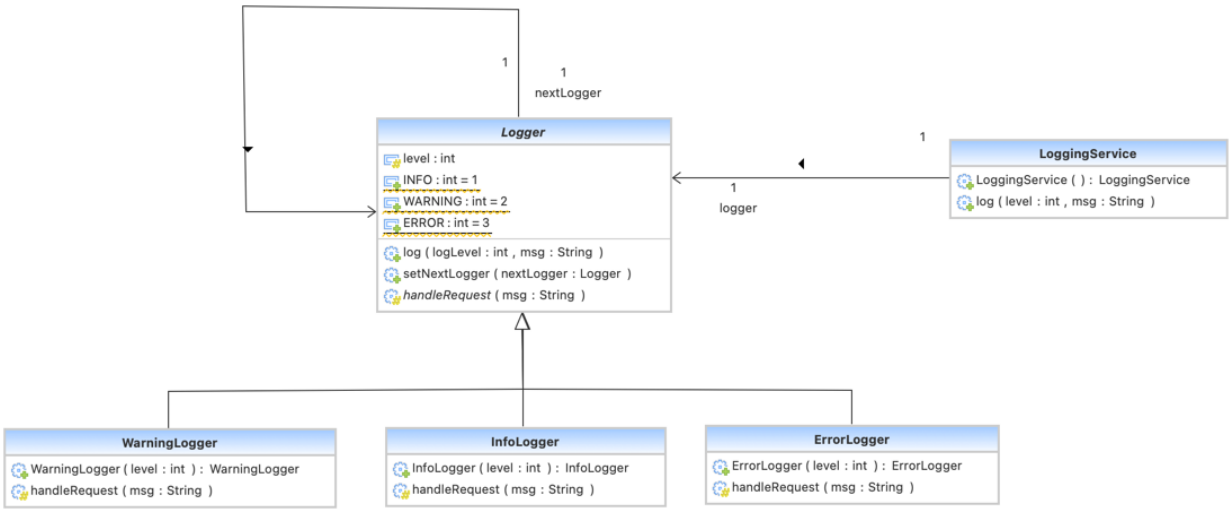
- When your software is expected to process multiple types of requests in various ways, but the specific types of requests and their sequences are unknown ahead of time, use the chain of responsibility design.
- When numerous handlers must be executed in a specific order, use the chain of responsibility pattern.
- When the set of handlers and their order are expected to change at runtime, use the chain of responsibility pattern.

You further learnt the concept of chain of responsibility design pattern through the example of logger. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the chain of responsibility pattern



- Application design after applying the chain of responsibility pattern



5.2 Understanding Observer Design Pattern

The observer design pattern can be summarised as follows:

- Observer pattern is a behavioural design pattern that allows you to establish a subscription mechanism to notify many objects of the events that occur on the object they are watching.
- The object that will notify other objects about changes to its status is known as the publisher. All other objects that keep track of changes to the publisher's state are known as subscribers.
- According to the Observer pattern, the publisher class should provide a subscription mechanism so that other objects can subscribe to or unsubscribe from the stream of events originating from the publisher.

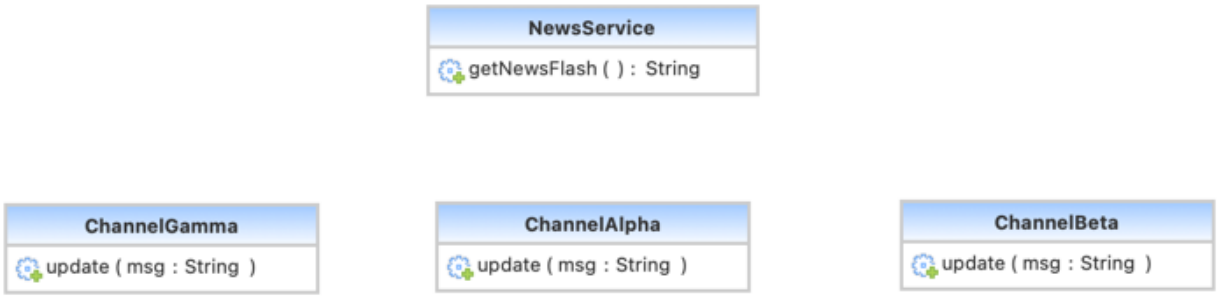
Now that we know what the observer pattern is, let's take a look at where it is applicable:

- When altering the state of one object results in a change in the state of other objects and the exact collection of objects is unknown or changes dynamically, use the observer design pattern.

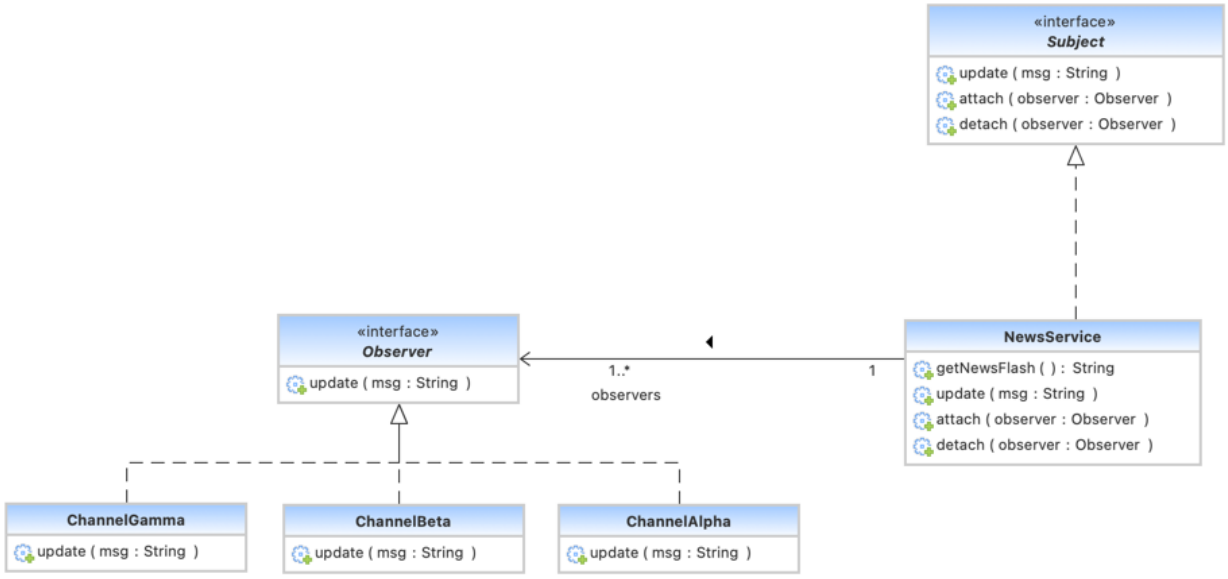
- When some objects in your program must monitor other objects for a limited time or under specific circumstances, use the observer pattern.

You further learnt the concept of observer design pattern through the example of a news service. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the observer pattern



- Application design after applying the observer pattern



5.1 Understanding Mediator Design Pattern

The mediator design pattern can be summarised as follows:

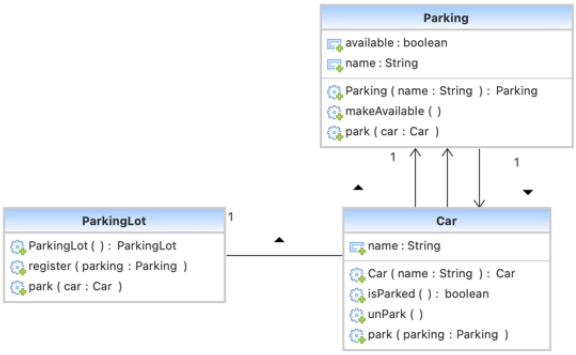
- Mediator pattern is a behavioural design pattern that allows you to decrease the chaos of object relationships. This pattern prevents the items from communicating directly with one another, forcing them to collaborate only through a Mediator object.
- In the mediator pattern, you should stop all direct contact among the components you want to be independent of one another. Instead, these components must work together through a specific Mediator object, which routes the calls to the proper components. As a result, instead of being tied to dozens of their colleague objects, the components just rely on a single Mediator class.
- A very good analogy of the mediator pattern can be aircrafts that do not talk to each other to decide who gets to land their plane first; instead, all the communication passes through the control tower.

Now that we know what the mediator pattern is, let's take a look at where it is applicable:

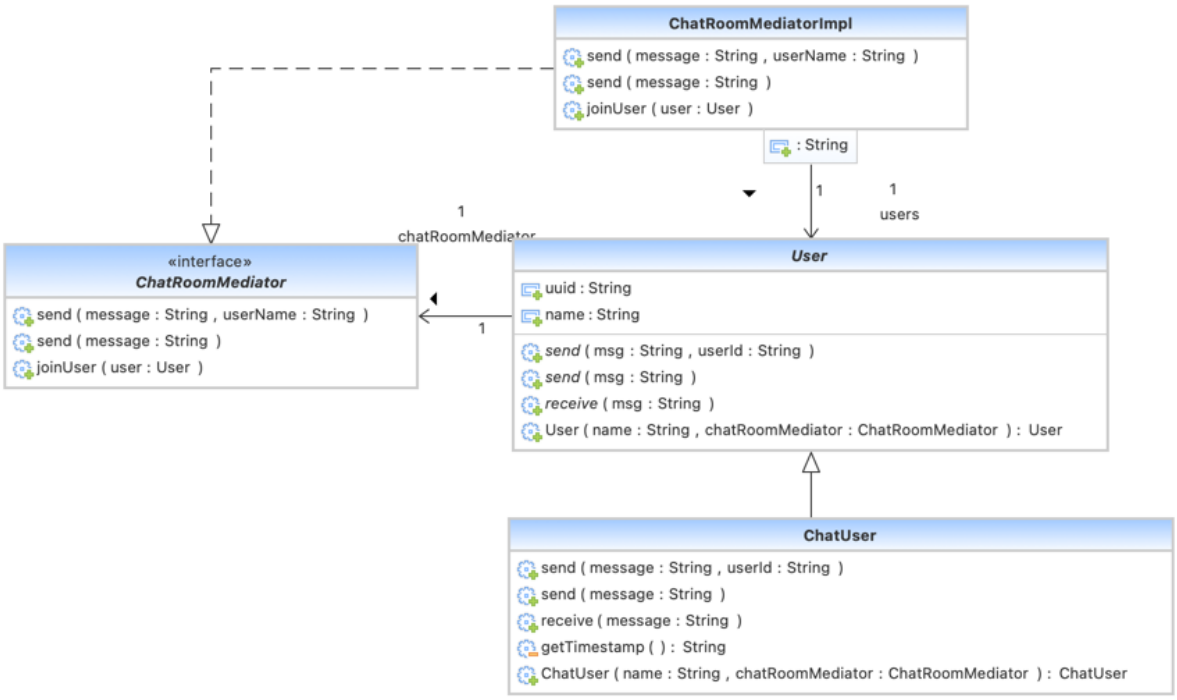
- When it is difficult to alter some classes because they are tightly related to a bunch of other classes, use the mediator design pattern.
- When you cannot reuse a component in another program because it is highly dependent on other components, utilise the mediator pattern.
- When you find yourself building a lot of component subclasses only to reuse some fundamental behaviour under different circumstances, use the mediator pattern.

You further learnt the concept of mediator design pattern through the example of a chatroom. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the mediator pattern



- Application design after applying the mediator pattern



5.1 Understanding Command Design Pattern

The command design pattern can be summarised as follows:

- The command pattern is a behavioural design pattern that turns a request into a stand-alone object along with all the request's details.
- You can use this transformation to pass requests as method arguments, postpone or queue the execution of a request and support undoable operations.

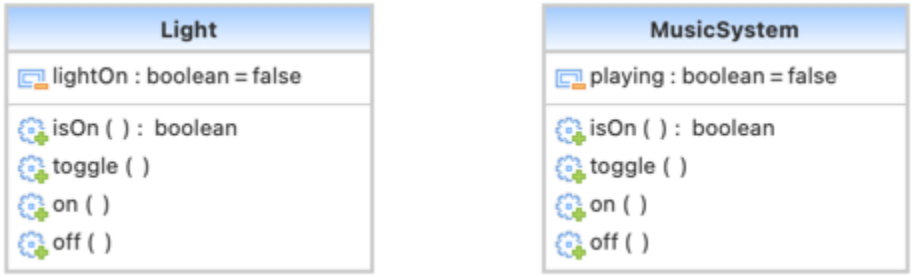
- The object that calls the action and the object that knows how to perform the action are separated by the command.
- The designer achieves this separation by creating an abstract base class that associates a receiver (an object) with an action. The execute() method in the base class just calls the action on the receiver.

Now that you know what the command pattern is, let's take a look at where it is applicable:

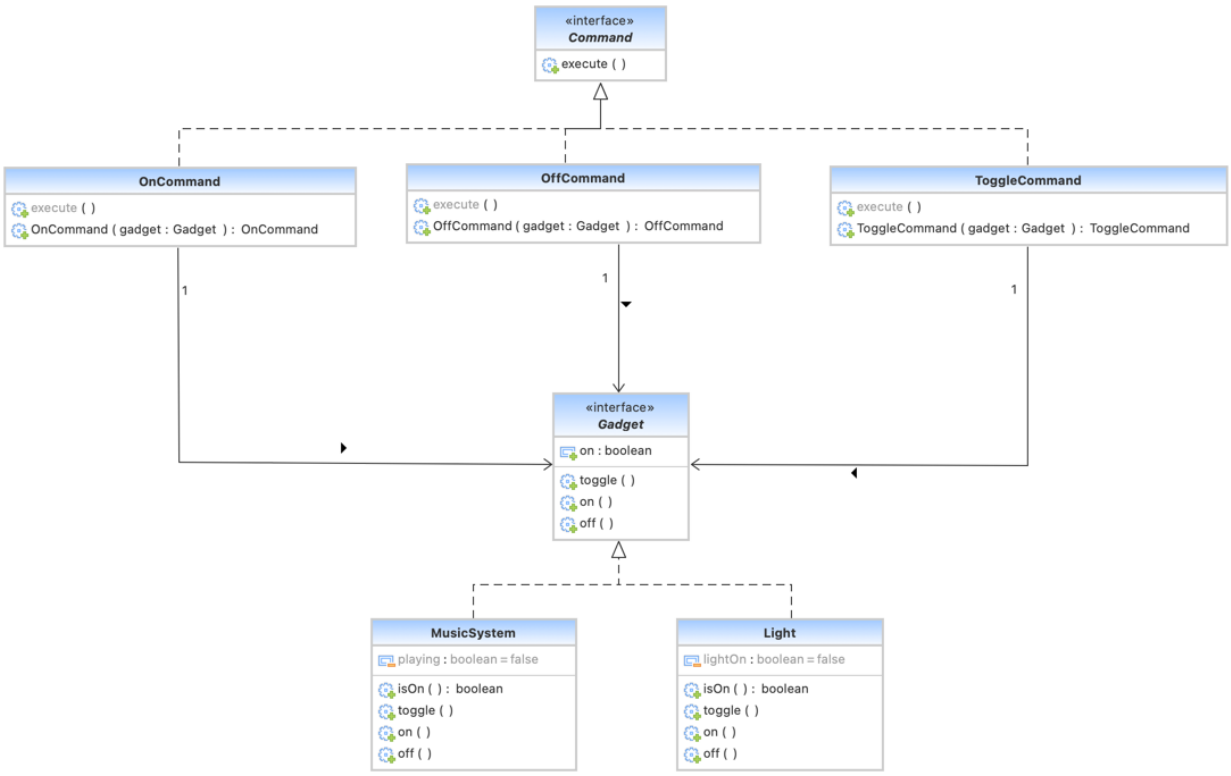
- When you want to parametrise objects with operations, use the command pattern.
- When you want to queue actions, schedule their execution or perform them remotely, use the command pattern.
- When you need to create reversible operations, use the command pattern.

You further learnt the concept of command design pattern through the example of electrical appliance control. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the command pattern



- Application design after applying the command pattern



5.1 Understanding Iterator Design Pattern

The iterator design pattern can be summarised as follows:

- Iterator pattern is a behavioural design pattern that allows you to traverse the components of a collection without revealing the underlying representation (such as list, stack, tree, etc.).
- The iterator pattern encapsulates the details of dealing with a complicated data structure and provides the client with a number of straightforward ways to retrieve collection elements.
- While this method is incredibly handy for the client, it also safeguards the collection from reckless or harmful activities that the client could perform if they worked directly with the collection.

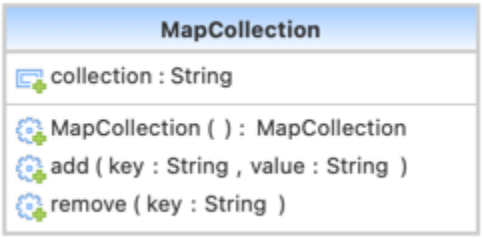
Now that you know what the iterator pattern is, let's take a look at where it is applicable:

- When your collection has a sophisticated underlying data structure that you want to hide from your clients, use the iterator pattern (either for convenience or for security reasons).

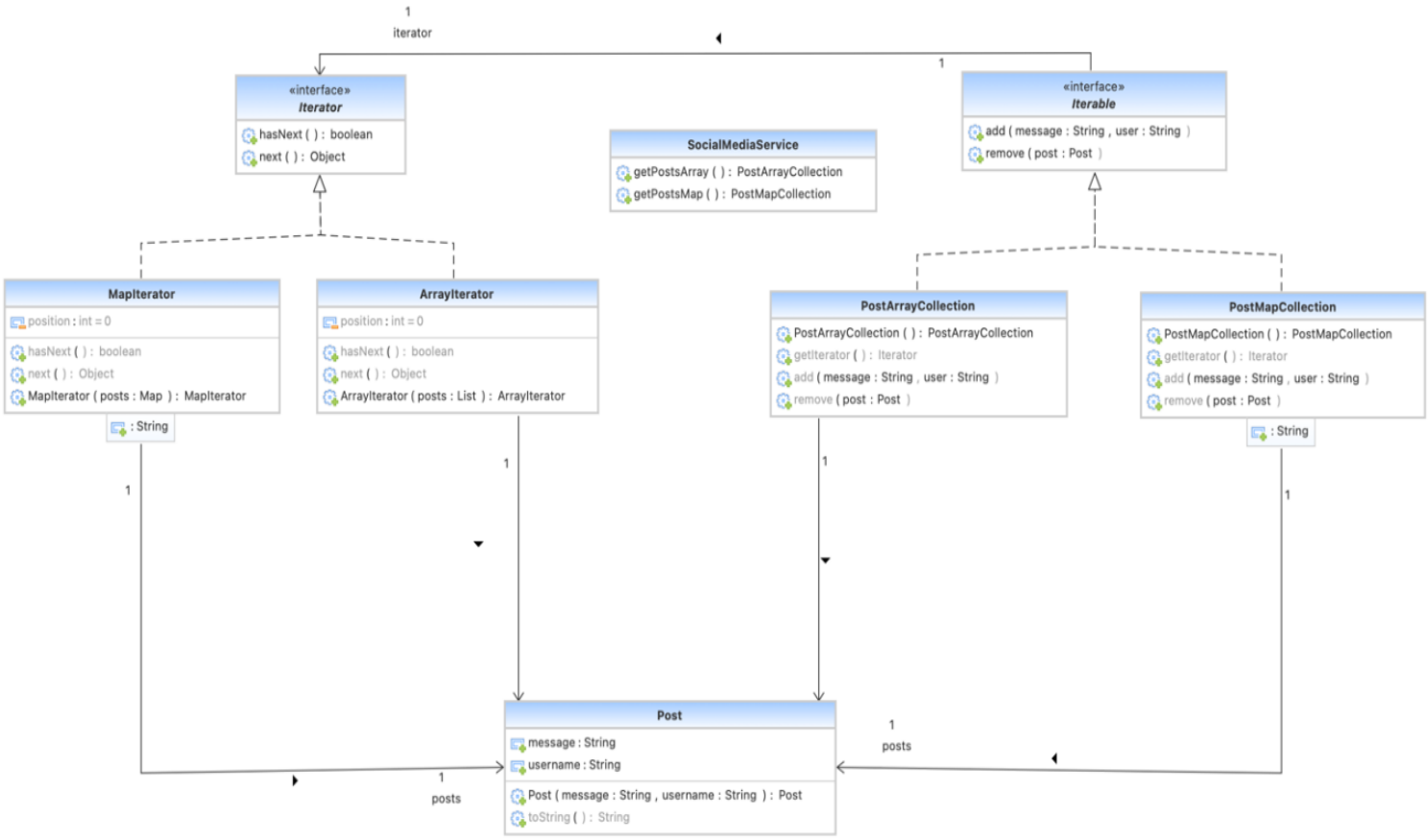
- Use the iterator pattern to avoid duplicating traversal code throughout your program.
- When you want your code to traverse different data structures or when the nature of these structures are unknown ahead of time, use the iterator pattern.

You further learnt the concept of iterator design pattern through the example of a social media service. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the iterator pattern



- Application design after applying the iterator pattern



5.1 Understanding Strategy Design Pattern

The strategy design pattern can be summarised as follows:

- Strategy pattern is a behavioural design pattern that allows you to construct a family of algorithms, segregate them into classes and interchange their objects.
- The strategy pattern advises extraction of all the algorithms from a class that accomplishes something specific in a variety of ways into distinct classes called strategies.

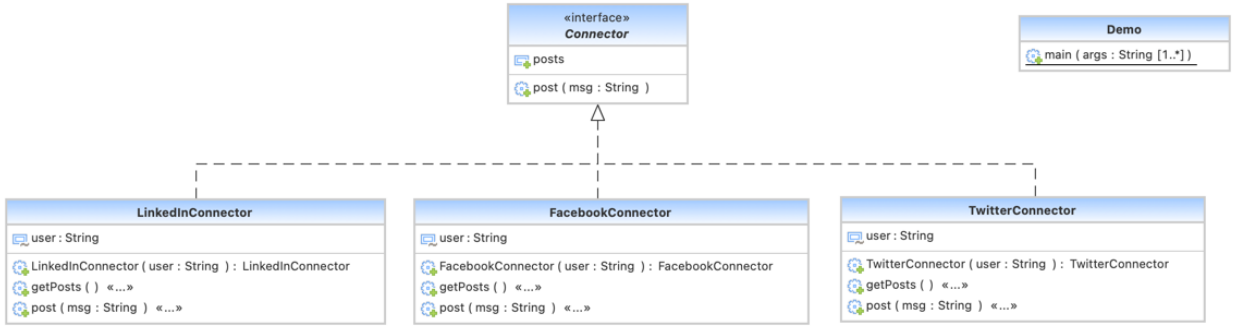
- A field in the original context class must be used to store a reference to one of the techniques. Instead of executing the task on its own, the context class delegates it to a linked strategy object.

Now that you know what the strategy pattern is, let's take a look at where it is applicable:

- When you wish to employ different variants of an algorithm within an object and switch between them during runtime, use the strategy pattern.
- When you have a group of similar classes that just differ in the way they perform some task, use the strategy pattern.
- Use the strategy pattern to separate a class's business logic from the implementation of algorithms.
- When your class contains a large conditional operator that varies between several variants of the same procedure, use the strategy pattern.

You further learnt the concept of strategy design pattern through the example of a social media platforms connector. The UML of the example before and after the application of the pattern are as shown below:

- Application design before applying the strategy pattern



- Application design after applying the strategy pattern