Design Patterns

Designing with Patterns

In software engineering, **design patterns** are the reusable solutions to commonly occurring problems in software design. These patterns started as best practices used in the industry that were applied multiple times to similar problems encountered in different contexts. Design patterns are used as templates or blueprints that any developer can customize to solve a recurring design problem on codes. Design patterns are not a specific piece of code, but a general concept for solving a particular problem. These are also used to design the architecture of an entire software system.

Design patterns are developed in the software industry. When a solution is repeatedly used in various projects, someone eventually puts a name to it and describes the solution in detail. That's basically how a pattern gets discovered.

Design patterns are said to have four (4) essential features:

- **Pattern Name** – This is the name of a pattern that can be used to describe a design problem, its solution, and consequences.
- **Problem** – This describes when to use the pattern. It explains the problem and its context.
- **Solution** – This describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of element solves it.
- **Consequences** – These are the results and interchanges of applying the pattern to the problem. They include time and space tradeoffs, but also flexibility, extensibility, and portability, among others.

Categories of Patterns

Design patterns differ by their complexity, level of detail, and scale of applicability to the entire software system being designed. Developers can implement design patterns in any language.

There are three (3) categories of design patterns in object-oriented design:

- **Creational Patterns** – These patterns deal with when and how objects are created. These provide object creation mechanisms that increase flexibility and reuse of existing codes.
- **Structural Patterns** – These describe how objects are composed into larger groups and explain how to assemble objects and classes into larger structures while keeping their structures flexible and efficient.
- **Behavioral Patterns** – These describe how responsibilities are distributed between objects in the design and how communication happens between objects.

Creational Patterns

Creational design patterns are all about class instantiation. These are patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to a certain situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

There are five (5) creational patterns:

- **Singleton Pattern** – This design pattern lets developers ensure that a class has only one (1) instance while providing a global access point to the instance.

  Problem:

  The Singleton pattern solves two (2) problems at the same time, therefore violating the Single Responsibility Principle:

  1. Ensure that a class has just a single instance. This is to control access to some shared resource—for example, a database or a file.
  2. Provide a global access point to that instance. While those global variables (which developers used to store some essential objects) are very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app. Just like a global variable, the Singleton pattern lets developers access some objects from anywhere in the program. However, it also protects that instance from being overwritten by other code.

  Solution:

  The Singleton pattern implementation has two (2) steps:

  1. Make the default constructor private to prevent other objects from using the `new` operator with a `Singleton` class.
  2. Create a static creation method that acts as a constructor. This method will call the private constructor to create an

object and saves it in a static field. All following calls to this method return the cached object.

If a code has access to a Singleton class, then it can call the Singleton's static method. Therefore, whenever that method is called, the same object is always returned.

Structure:

- In *Figure 1*, the `Singleton` class declares the static method `getInstance()` that returns the same instance of its own class. The Singleton's constructor should be hidden from the client code. Calling the `getInstance()` method should be the only way of getting the `Singleton` object.

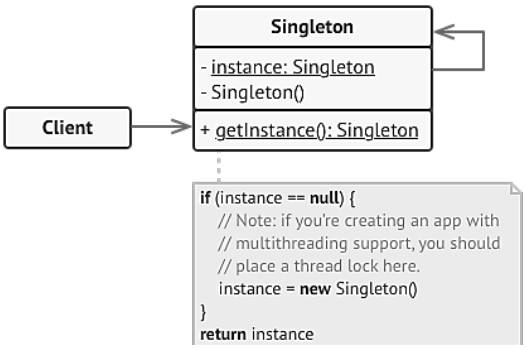

Figure 1. Singleton structure
Source: https://refactoring.guru/design-patterns/singleton

- **Factory Method Pattern** – This provides an interface for creating objects in a superclass but allows subclasses to alter the type of object that will be created.

Problem:

A developer is creating a logistics management application. The first version of his app can only handle transportation by trucks, so the bulk of his code lives inside the `Truck` class. After a while, his app becomes pretty popular. Each day he receives dozens of requests from sea transportation companies to incorporate sea logistics into the app.

At present, most of his code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Furthermore, if he decides to add another type of transportation to the app later, he will probably need to make all of these changes again.

As a result, he will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

Solution:

The Factory Method pattern suggests that he replace direct object construction using the `new` operator with calls to a special factory method. The objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as "products".

Structure:

- In *Figure 2*, `Product` declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
- The `ConcreteProduct` classes are different implementations of the product interface.
- The `Creator` class declares the factory method that returns new product objects. The return type of this method must match the product interface.
  Developers can declare the factory method as an abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.
- The `ConcreteCreator` classes override the base factory method, so it returns a different type of product.

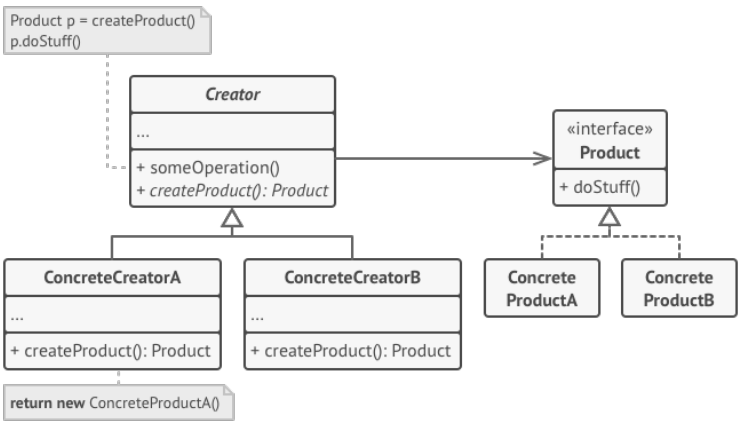

Figure 2. Factory method structure
Source: https://refactoring.guru/design-patterns/factory-method

- **Prototype Pattern** – This creational design pattern lets a developer copy existing objects without making his code dependent on their classes.

Problem:

When a developer wants to create an exact copy of an object, s/he has to create a new object of the same class. Then, s/he has to go through all the fields of the original object and copy their values over to the new object. The problem here is not all objects can be copied that way because some of the objects' field may be private and may not be visible from outside of the object itself. Another problem to this approach is that since s/he has to know the object's class to create a duplicate, his/her code becomes dependent on the class.

Solution:

The Prototype pattern delegates the cloning process to the actual objects being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets a developer clone an object without coupling the code to the class of that object. Usually, such an interface contains just a single `clone()` method.

The implementation of the `clone()` method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. Developers can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

An object that supports cloning is called a prototype. When a program's objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

When creating a set of objects that are configured in various ways, and a program needs an object like the one that is already configured, the developer just clones a prototype instead of constructing a new object from scratch.

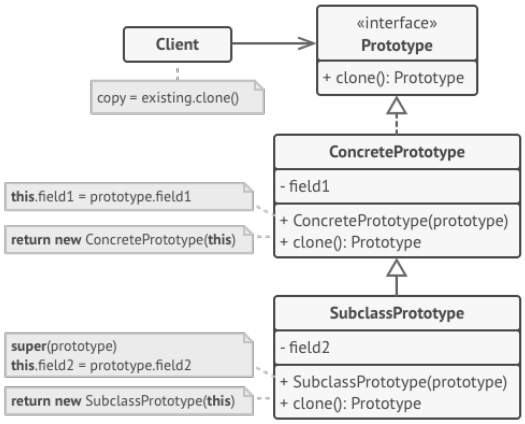

Figure 3. Prototype structure
Source: https://refactoring.guru/design-patterns/prototype

Structure:

Basic Implementation of Prototype Pattern

- In *Figure 3*, the `Prototype` interface declares the cloning methods. In most cases, it is a single clone method.
- The `ConcretePrototype` class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.
- The `Client` class can produce a copy of any object that follows the prototype interface.

Prototype Registry Implementation

- In *Figure 4*, the `PrototypeRegistry` class provides an easy way to access frequently used prototypes. It stores a set of pre-built objects that are ready to be copied.

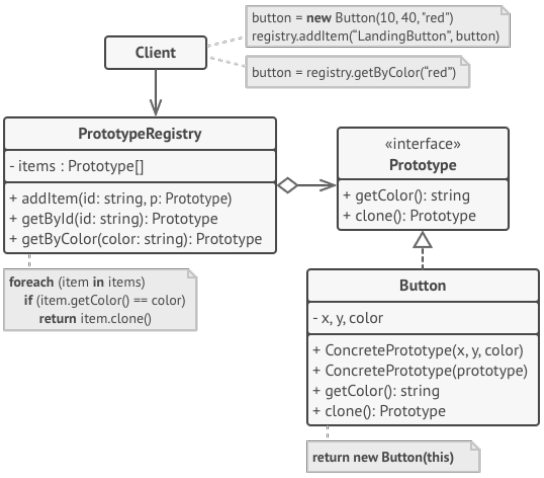

Figure 4. Prototype registry implementation structure
Source: https://refactoring.guru/design-patterns/prototype

- **Abstract Factory Pattern**
  - This allows developers to produce families of related objects without specifying their concrete classes.
  - A family of objects that are usually used together or a set of objects that are dependent on each other in some way.

Problem:

A developer is creating a furniture shop simulator. His code consists of classes that represent a family of related products; for example, a family of furniture with chair, sofa, and coffee table. The products are available in different variants of family such as modern, art deco, and Victorian. He needs a way to create individual furniture objects so that they match the other objects of the same family. The developer also doesn't want to change the existing code when adding new products or

families of products to the program. Furniture vendors update their catalogs very often, and he wouldn't want to change the core code each time it happens.

Solution:

These are the following steps to implement the Abstract Factory pattern:

1. Explicitly declare interfaces for each distinct product of the product family. Then developers can make all variants of products to follow those interfaces. For example, all chair variants can implement the `Chair` interface and all coffee table variants can implement the `CoffeeTable` interface.
2. Declare an Abstract Factory interface with a list of creation methods for all products that are part of the product family such as `createChair()`, `createSofa()`, and `createCoffeeTable()`. These methods must return abstract product types presented by the created interfaces of each product.
3. For each variant of a product family, create a separate factory class based on the Abstract Factory interface. A factory is a class that returns products of a particular kind. For example, a class named `ModernFurnitureFactory` class that implements the Abstract Factory interface can only create modern chair, modern sofa, and modern coffee table objects.

The client code has to work with both factories and products through their respective abstract interfaces. This lets developers change the type of factory that is passed to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

Structure:

- In *Figure 5*, the *Abstract Products* declare interfaces for a set of distinct but related products which make up a product family.
- `ConcreteProducts` are various implementations of abstract products grouped by variants. Each abstract product must be implemented in all given variants.
- The `AbstractFactory` interface declares a set of methods for creating each of the abstract products.
- `ConcreteFactory` classes implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and only creates those product variants.
- Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding abstract products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory. The `Client` can work with any concrete factory or product variant as long as it communicates with their objects via abstract interfaces.

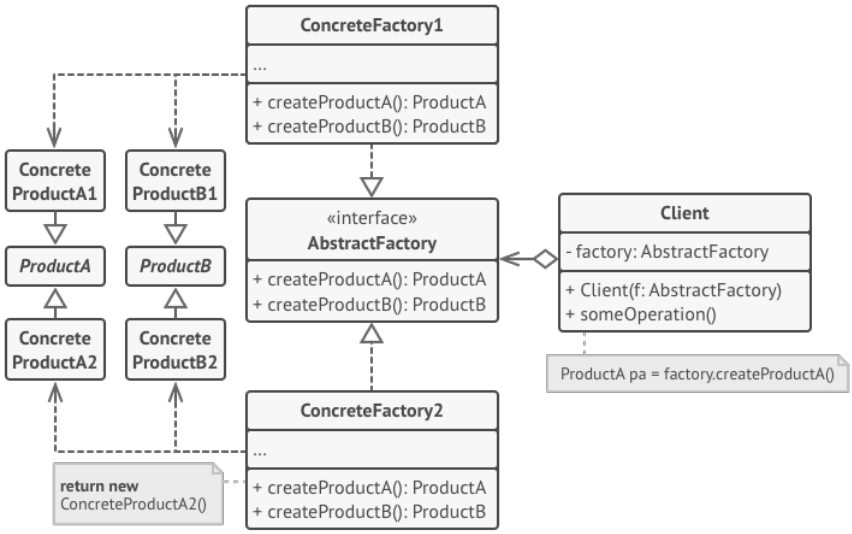

Figure 5. Abstract factory structure
Source: https://refactoring.guru/design-patterns/abstract-factory

- **Builder Pattern** – This creational design pattern allows developers to construct complex objects step by step. The pattern allows developers to produce different types and representations of an object using the same construction code.

Problem:

When creating an object that requires complex step-by-step initialization of many fields and nested objects, the initialization of an object will usually contain lots of parameters or will scatter all over the client code. For example, to build a `House` object, a developer needs to construct four (4) walls and a floor, install a door and a pair of windows, and build a roof. In the future, he might also need to add other features of the house such as garage, swimming pool, garden, etc.

The simplest solution he might implement is to extend the base `House` class and create a set of subclasses to cover the additional parameters. But the problem of this approach will create a considerable number of subclasses. Any additional

new parameter will require growing this hierarchy even more.

Another solution is to create a constructor right in the base `House` class with all possible parameters that control the `House` object. The problem on this approach is that in some situations, some parameters will be unused, thereby making the initialization ugly.

Solution:

The Builder design pattern suggests that developers must extract the object construction code out of its own class and move it to separate objects called builders. The pattern organizes object construction into a set of steps or methods such as `buildWalls()`, `buildDoor()`, `buildWindows()`, etc.

To create an object, execute a series of these steps on a builder object. The important part is that developers don't need to call all of the steps. They can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementation when the program needs to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone. In this case, a developer can create several different builder classes that implement the same set of building steps but differently. Then, s/he can use these builders in the construction process to produce different kinds of objects.

For example, a builder builds everything from wood and glass, a second builder builds everything with stone and iron, and a third builder uses gold and diamonds. By calling the same set of steps, developers get a regular house from the first builder, a small castle from the second builder, and a palace from the third builder. This will only work if the client code that calls the building steps can interact with builders using a common interface.

Director

A developer can extract a series of calls to the builder steps s/he used to construct a product into a separate class called `Director`. The `Director` class defines the order of executing the building steps, while the builder provides the implementation for those steps.

The `Director` class is a good place to put various construction routines so developers can use them across the program. It completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

Structure:

- In *Figure 6*, the `Builder` interface declares product construction steps that are common to all types of builders. `ConcreteBuilder` classes provide different implementations of the construction steps.
- `ConcreteBuilder` classes may produce products that do not follow the common interface.
- `Products` classes are the resulting objects. `Product` classes constructed by different builders do not have to belong to the same class hierarchy or interface.
- The `Director` class defines the order of calling construction steps so that developers can create and reuse specific configurations of products.
- The `Client` class must associate one of the builder objects with the director. Usually, it's done just once through the parameters of the director's constructor. Then, the director uses that builder object for all further construction. However, there is an alternative approach for when the client passes the builder object to the production method of the director. In this case, a developer can use a different builder each time s/he produces something with the director.

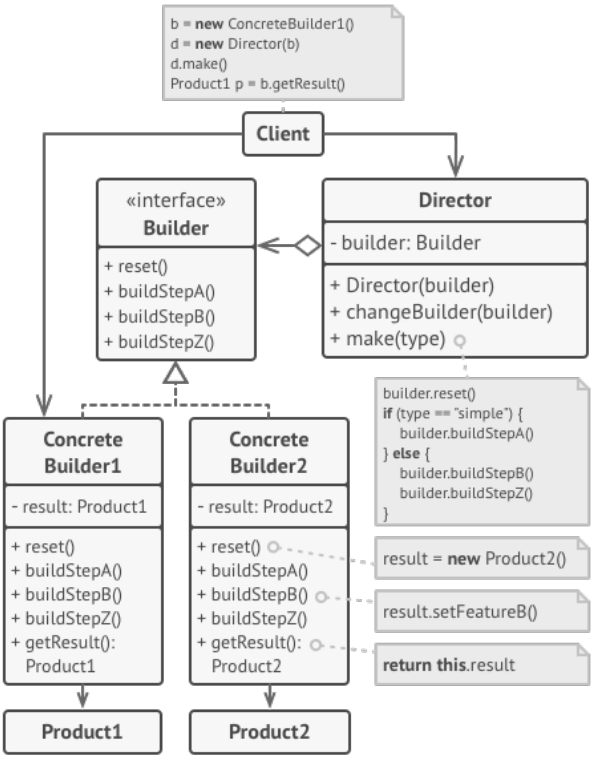

Figure 6. Builder pattern structure
Source: https://refactoring.guru/design-patterns/builder

Structural Patterns

Structural design patterns deal with the arrangement and relationship between the classes in the software system. These patterns are all about grouping classes together and providing ways for objects to coordinate to get work done. Structural patterns can also control and grant access to an object and add new functionalities to existing objects.

There are seven (7) design patterns in this category:

- **Adapter Pattern** – This structural design pattern allows objects with incompatible interfaces to collaborate. This pattern converts the interface of a class into another interface as expected by the client application.

  Problem:
  For example, a developer developed a stock monitoring software for a certain client. The software downloads the stock data from multiple sources in XML format and then displays the charts and diagrams for the user. In the future, the client decided to improve the software by integrating a smart third-party analytics library. But the problem is that this third-party library only works with data in JSON format. He can change the library to work with XML format. However, this might break some existing code that relies on the library. He might also not have access to the third-party library's source code, making this approach impossible.

  Solution:
  In the Adapter pattern, a developer can create a special object called *adapter* that converts the interface of one (1) object so that another object can understand it. An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. For example, developers can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

  Adapters can not only convert data into various formats but can also help objects with different interfaces to collaborate. The following steps show how an adapter works:

  1. The adapter gets an interface compatible with one of the existing objects.
  2. Using this interface, the existing object can safely call the adapter's methods.
  3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

  To solve the problem of incompatible formats from the previous example of stock monitoring software, the developer can create XML to JSON adapters for every class of the analytics library that works with the code directly. Then the developer can adjust the code to communicate with the library only through these adapters. When an adapter receives a call, it will translate the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

  Structure:
  Object Adapter
  This implementation uses this object composition principle: the adapter implements the interface of one (1) object and wraps the other object.
  - In *Figure 7*, the `Client` class contains the existing business logic of the program.
  - The interface `Client Interface` describes a protocol that other classes must follow to be able to collaborate with the client code.
  - The `Service` class is usually a third-party class. The client cannot use this class directly because it has an incompatible interface.
  - The `Adapter` class can work with both the client and the `Service` class. It implements the client interface while wrapping the service object. The adapter receives calls from the client through the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

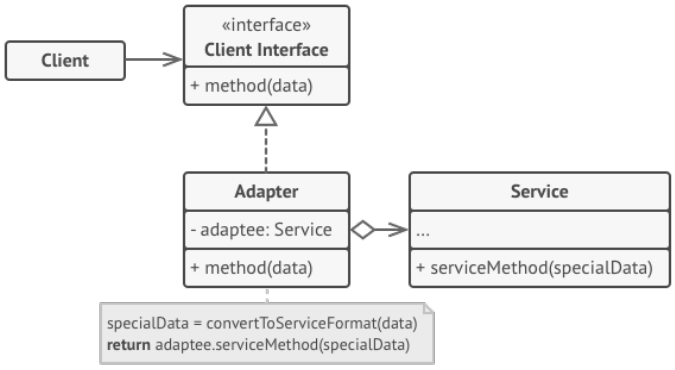

Figure 7. Adapter structure
Source: https://refactoring.guru/design-patterns/adapter

The client code does not get coupled to the concrete adapter class as long as it works with the adapter through the client interface. This will allow the client code to use new types of adapters without breaking the existing client code. This is also useful when the interface of the service class gets changed or replaced. The developers only need to create a new adapter class without changing the client code.

- **Bridge Pattern** – This allows the developers to split a large class or a set of closely related classes into two (2) separate hierarchies, which are abstraction and implementation, that can be developed independently of each other.

  Problem:
  For example, a developer created a Shape class with a pair of subclasses: `Circle` and `Square`. When the developer wants to extend this class hierarchy to incorporate colors, he will create `Red` and `Blue` shape subclasses. However, the program already has two (2) subclasses, so the developer needs to create four (4) class combinations such as `BlueCircle`, `RedCircle`, `BlueSquare`, and `RedSquare`.

  Adding new shape types and colors to the hierarchy will grow it exponentially. Adding new types would require creating several subclasses for each type.

  Solution:
  The given problem occurs because the developer is trying to extend the shape classes in two (2) different dimensions: by form and by color.

  The Bridge pattern solves this problem by switching from inheritance to the object composition. This means that the developer must extract one (1) of the dimensions into a separate class hierarchy so that the original classes will reference an object of the new hierarchy instead of having all of its states and behaviors within a single class.

  In the given problem, the developer can extract the color-related code into its own class with two (2) subclasses: `Red` and `Blue`. The Shape class then gets a reference field pointing to one (1) of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and `Color` classes. This design architecture allows the adding of new colors without changing the shape hierarchy, and vice versa.

  Structure:
  - In *Figure 8*, the `Abstraction` provides high-level control logic. It relies on the implementation object to do the actual low-level task.
  - The `Implementation` declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.
    The abstraction may list the same methods as the implementation, but usually, the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.
  - `Concrete Implementations` contain platform-specific code.
  - `Refined Abstractions` provide variants of control logic. Like their parent classes, they work with different implementations via the general implementation interface.
  - The `Client` application is only interested in working with the abstraction. However, it is the client's job to link the abstraction object with one of the implementation objects.

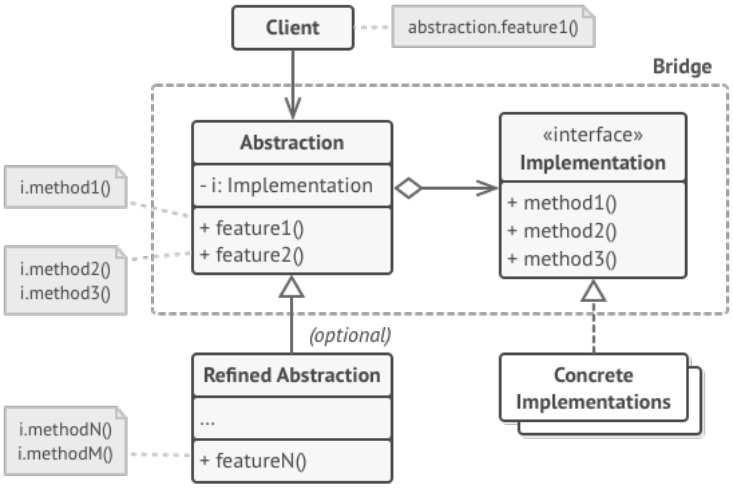

Figure 8. Bridge structure
Source: https://refactoring.guru/design-patterns/bridge

- **Composite Pattern** – This design pattern allows developers to compose objects into tree structures and then work with these structures treating them as individual objects.

  Problem:
  A developer created two (2) types of objects named `Products` and `Boxes`. The `Box` object contains several `Products` as well as a number of smaller `Boxes`. These `Boxes` can also hold some `Products` or even smaller `Boxes`, and so on. These classes will be used to create an ordering system. Orders will contain simple products without any wrapping, and boxes will contain products and boxes. To determine the total price of such an order, the developer will need to unwrap all the boxes and calculate the total price. This approach can be hard or even impossible.

  Solution:
  The Composite pattern suggests that a developer must work with `Products` and `Boxes` through a common interface which declares a method for calculating the total price. This can be done by simply making a product to return its price. For a box, it will go over each item the box contains, ask its price, and then returns the total for this box. If a box contains smaller boxes, those boxes will also start going over their contents and so on, until the prices of all inner components are calculated. A box could even add some extra cost to the final price, such as packaging cost.

  The benefit of this approach is that developers do not need to care about the concrete classes of objects that compose the tree. The objects can all be treated through the common interface. When the developer calls a method, the objects themselves will pass the request down the tree.

  Structure:
  - In *Figure 9*, the `Component` interface describes operations that are common to both simple and complex elements of the tree.
  - The `Leaf` is a basic element of a tree that does not have sub-elements. Usually, leaf components end up doing most of the real work since they do not have anyone to delegate the work to.
  - The `Composite` is an element that has sub-elements: leaves or other containers. A container does not know the concrete classes of its children. It works with all sub-elements only via the component interface.
  - Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results, and returns the final result to the client.
  - The `Client` works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

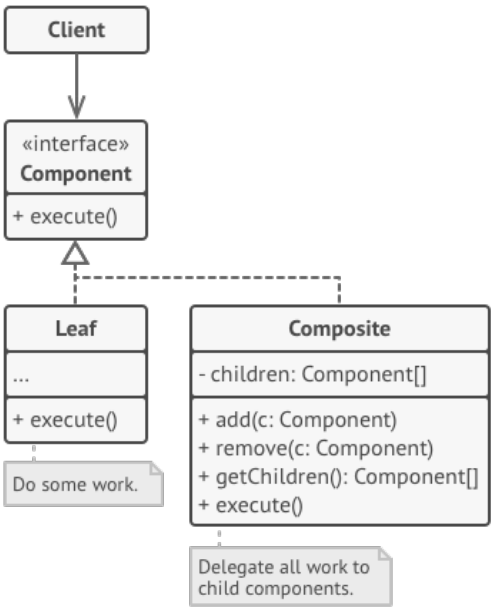

Figure 9. Composite structure
Source: https://refactoring.guru/design-patterns/composite

- **Decorator Pattern** – This allows developers to attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

  Problem:
  A developer is working on a notification library that lets other programs notify their users about important events. The initial version of the library was based on the `Notifier` class that only has a few fields, a constructor, and a single `send()` method. The method can accept a message argument from a client and send the message to a list of e-mails that were passed to the `Notifier` object through its constructor. A third-party app that acts as a client is supposed to create and configure the `Notifier` object once, and then use it each time something important happened.

  The requirements changed. Some of the clients wanted to receive an SMS notification, while the others through social media. This can be hard since the developer needs to extend the `Notifier` class and put additional notification methods into new subclasses.

  The requirements changed again. Some clients wanted to add a new type of notification, and some wanted a combination of some of types of notification. The developer can address this problem by creating special subclasses which combine several notification methods within a single class. However, this approach will make the code bigger, not only the library code but the client code as well.

Solution:

The Decorator pattern suggests creating a wrapper object that can be linked with some target object. The wrapper contains the same set of methods as the target and delegates it to all requests it receives. The wrapper implements the same interface as the wrapped object then makes the wrapper's reference accept any object that follows that interface. This will cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

In the given problem, the simple e-mail notification behavior will stay inside the base `Notifier` class but will turn all other notification methods into decorators. The client code would need to wrap a basic `Notifier` object into a set of decorators that matches the client's preferences. The resulting objects will be structured as a stack. The last decorator in the stack would be the object that the client actually works with.

This approach can be applied to other behaviors, such as formatting messages. The client can decorate the object with any custom decorators as long as these follow the same interface as the others.

Structure:

- In *Figure 10*, the `Component` declares the common interface for both wrappers and wrapped objects.
- `Concrete Component` is a class of objects being wrapped. It defines the basic behavior which can be altered by decorators.
- The `Base Decorator` class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
- `Concrete Decorators` define extra behaviors that can be added to components dynamically. These override methods of the base decorator and execute their behavior either before or after calling the parent method.
- The `Client` can wrap components in multiple layers of decorators as long as it works with all objects via the component interface.

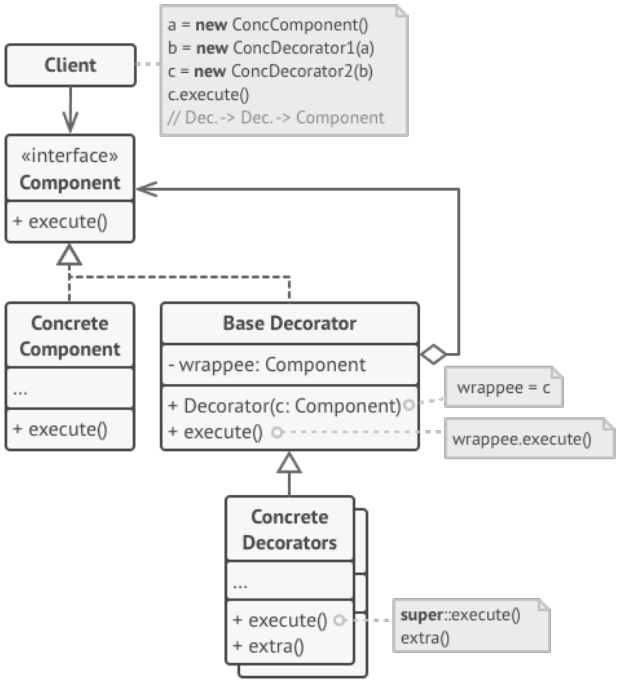

Figure 10. Decorator structure
Source: https://refactoring.guru/design-patterns/decorator

- **Facade Pattern** – This provides a simplified interface to a library, a framework, or any other complex set of classes.

Problem:

A developer is creating a code that works with a broad set of objects that belong to a sophisticated library or framework. He will simply need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on. As a result, the business logic of his classes would become tightly coupled to the implementation details of third-party classes, making it hard to comprehend and maintain.

Solution:

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about. Using a facade is convenient when the program needs to integrate with a library that has dozens of features, but the program only needs its few functionalities.

Structure:

- In *Figure 11*, the `Facade` provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.
- An `Additional Facade` class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Both clients and other facades can use additional facades.
- The `Complex Subsystem` consists of dozens of various objects. To make them all do something meaningful, developers

have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.
Subsystem classes are not aware of the facade's existence. They operate within the system and work with each other directly.

- The `Client` uses the facade instead of calling the subsystem objects directly.

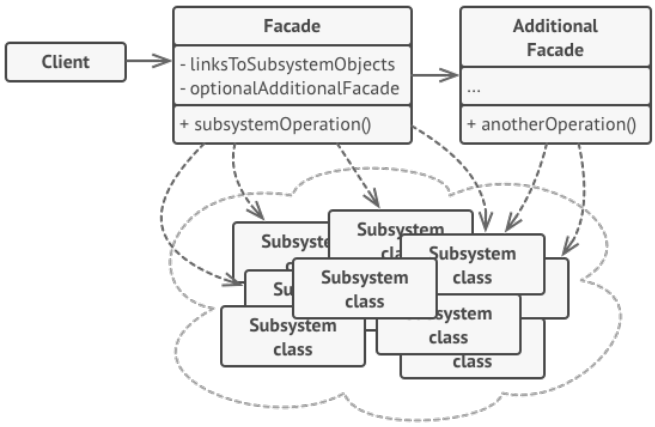

Figure 11. Facade pattern structure
Source: https://refactoring.guru/design-patterns/facade

- **Flyweight Pattern** – This allows developers to fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each other.

Problem:
A developer creates a video game with realistic particle system of multiple bullets, missiles, and explosions. When the game is built and tested on his computer, the game is always crashing during gameplay. The developer discovers that the game keeps on crashing because of an insufficient amount of memory. This problem is caused by the realistic particle system. Each particle, such as the bullet, is represented by a separate object containing plenty of data. When the program creates multiple particles, some of these particles may not fit into the remaining memory—hence the crash.

Solution:
The Flyweight pattern suggests creating an object that is divided into two (2) parts: the state-dependent part and the state-independent part.

In the given problem, the particle system objects shared some fields, such as color and sprit; these are state-independents and their values remain constant. While other fields such as coordinates, movement, and speed are unique to each particle object, these are state-dependents and their values may always change.

The state-independent is stored in the Flyweight object. The state-dependent is stored or computed by client objects and is passed to the Flyweight object when its operations are invoked.

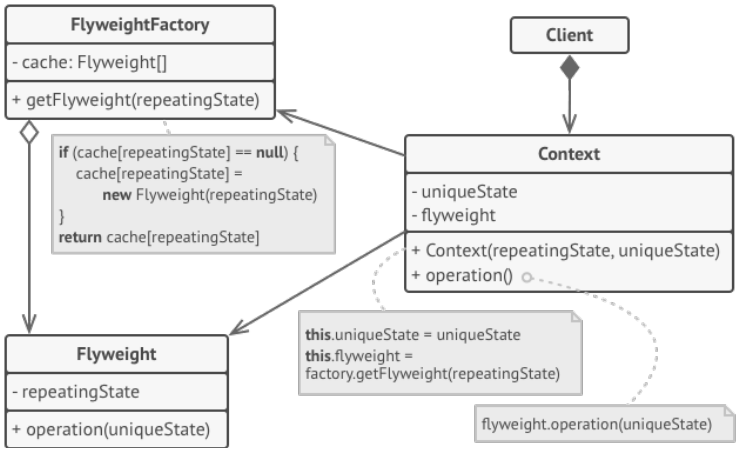

Figure 12. Flyweight pattern structure
Source: https://refactoring.guru/design-patterns/flyweight

Structure:

- In *Figure 12*, the Flyweight pattern is merely an optimization. Before applying it, make sure that a program does have the RAM consumption problem related to having a massive number of similar objects in memory at the same time. Make sure that this problem cannot be solved in any other meaningful way.
- The `Flyweight` class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called "intrinsic."

The state passed to the flyweight's methods is called "extrinsic."

- Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.
- The Client calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.
- The Flyweight Factory manages a pool of existing flyweights. With the factory, clients do not create flyweights directly. Instead, they call the factory, passing bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.

- **Proxy Pattern** – This design pattern allows developers to provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing developers to perform something before or after the requests gets through to the original object.

Problem:

Developers want to control access to an object. For example, a developer has a large object that consumes a tremendous amount of system resources. The client program needs it from time to time. The developer can do this by creating the object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. However, this will cause a lot of code duplication.

Solution:

The Proxy pattern suggests creating a new proxy class with the same interface as an original service object and updating the application so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

The benefit of this approach is that when the program needs to execute something before or after the primary logic of the class, the proxy lets the program do this without changing that class. Since proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

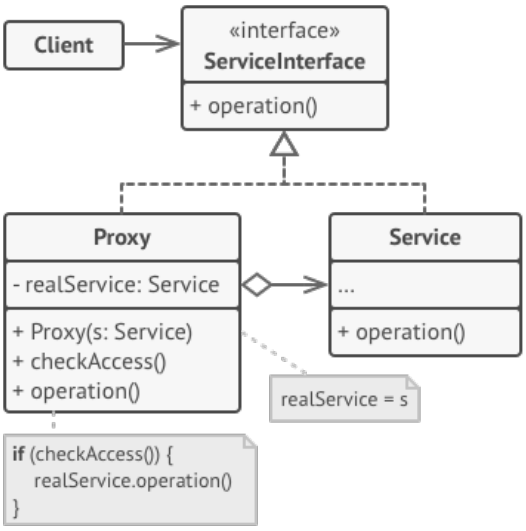

Figure 13. Proxy structure
Source: https://refactoring.guru/design-patterns/proxy

Structure:

- In *Figure 13*, the `Service Interface` declares the interface of the `Service`. The proxy must follow this interface to be able to disguise itself as a service object.
- The `Service` is a class that provides useful business logic.
- The `Proxy` class has a reference field that points to a service object. After the proxy finishes its processing, it passes the request to the service object.
- The `Client` should work with both services and proxies via the same interface. This way developers can pass a proxy into any code that expects a service object.

Behavioral Patterns

Behavioral design patterns are concerned with how classes and objects behave in a system software and how objects communicate with each other. Some of the design patterns in this category are iterator, observer, and strategy design patterns. These patterns describe how to assign behavioral responsibilities to classes.

- **Iterator Pattern** – This design pattern allows developers to traverse elements of a collection without exposing its underlying representation, such as list, stack, and tree. This pattern is used for sequentially iterating and accessing items from a collection of items.

  Problem:
  A developer is creating a system with a search feature. The system must allow clients to search an item from a tree data structure, which can be done by implementing a certain search algorithm. If different clients want to perform a different type of search, adding several search algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Since collections provide different ways of accessing their elements, developers need to couple their code to the specific collection classes.

  Solution:
  The main idea of the Iterator pattern is to extract the search behavior of a collection into a separate object called an *iterator*. An iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection—independently of each other—at the same time.

  Iterators provide one (1) primary method for fetching elements of the collection. The client can keep running this method until it does not return anything, which means that the iterator has traversed all of the elements.

  All iterators must implement the same interface. This makes the client code compatible with any collection type or any search algorithm as long as there is a proper iterator. If a program needs a special way to traverse a collection, developers only need to create a new iterator class without changing the collection or the client code.

  Structure:
  - In *Figure 14*, the `Iterator` interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.
  - The `Concrete Iterators` implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.
  - The `Collection` interface declares one (1) or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.
  - The `Concrete Collections` return new instances of a particular concrete iterator class each time the client requests one.
  - The `Client` works with both collections and iterators via their interfaces. This way, the client isn't coupled to concrete classes, allowing developers to use various collections and iterators with the same client code.

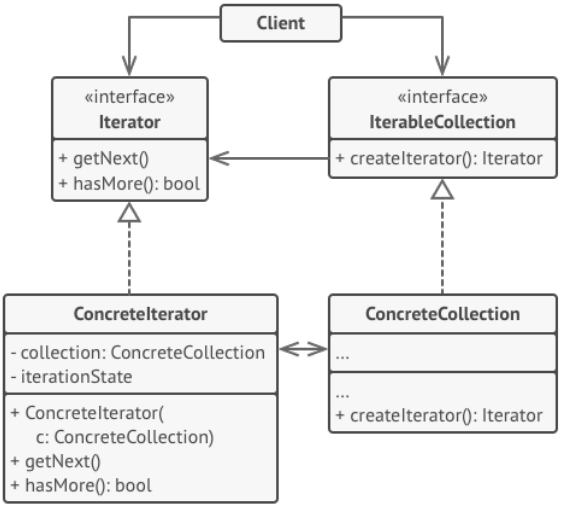

Figure 14. Iterator structure
Source: https://refactoring.guru/design-patterns/iterator

- **Observer Pattern** – This design pattern allows developers to define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing. This means that when an object changes state, all of its dependents are notified and updated automatically.

  Problem:
  A developer created two (2) types of objects named `Customer` and `Store`. A customer is interested in a particular brand of

product which should become available in the store sooner. The customer can visit the store's website and check the product's availability. The store can send lots of e-mails to all customers each time a new product becomes available. This would save time for some customers to visit the website. But this will upset other customers who are not interested in new products.

Solution:

The object that notifies other objects about the changes to its state is called *publisher*. All other objects that want to track changes to the publisher's state are called *subscribers*. The Observer pattern suggests adding a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. This mechanism consists of an array of field for storing a list of references to subscriber objects and several public methods which allow adding subscribers to and removing them from the list.

Whenever an important event happens to the publisher, it goes over its subscribers and calls the notification method on their objects. All subscribers must implement the same interface and that the publisher communicates with them only through that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification. The publisher will allow subscribers to observe publisher's state without coupling to their concrete classes.

Structure:

- In *Figure 15*, the `Publisher` class issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.
- When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.
- The `Subscriber` interface declares the notification interface. In most cases, it consists of a single `update()` method. The method may have several parameters that let the publisher pass some event details along with the update.
- The `Concrete Subscribers` perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface, so the publisher is not coupled to concrete classes.

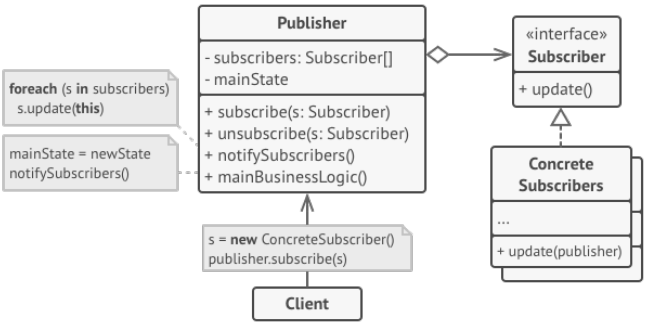

Figure 15. Observer structure
Source: https://refactoring.guru/design-patterns/observer

- Subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.
- The `Client` class creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

- **Strategy Pattern** – This allows developers to define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem:

A developer created a navigation application for travelers. The application is centered around a beautiful map which helps users orient themselves in any city quickly. The clients requested a new application feature that could automate route planning where a user can enter an address and see the fastest route to the destination displayed on the map. The developer created the application to build the routes for roads, walking routes, public transportation routes, and so on. While the application is working successfully, the main class of the navigator doubled in size every time a new routing algorithm is added. This makes the application hard to maintain; any changes to the algorithms affect the whole class, increasing the chance of an error to occur in already-working code.

Solution:

The Strategy pattern suggests defining the class that does a specific task in several different methods and extracts all of

these algorithms into separate classes called *strategies*. The original class called *context* must have a field of storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context will not be responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. The context will work with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy. This approach makes the context class become independent of concrete strategies, so developers can add new algorithms or modify existing ones without changing the code of the context or other strategies.

In the given problem, each routing algorithm can be extracted to its own class with a single method. This method accepts an origin and destination and returns a collection of the routes' checkpoints.

Structure:

- In *Figure 16*, the `Context` class maintains a reference to one of the concrete strategies and communicates with this object only through the `Strategy` interface.
- The `Strategy` interface is common to all concrete strategies. It declares a method that the context uses to execute a strategy.
- The `Concrete Strategies` implement different algorithms that the context uses.
- The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context will not know what type of strategy it works with or how the algorithm is executed.
- The `Client` class creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

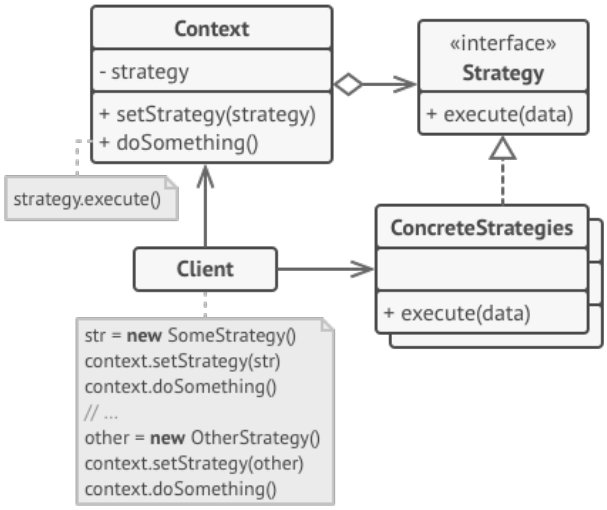

Figure 16. Strategy structure
Source: https://refactoring.guru/design-patterns/strategy

REFERENCES:

Classification of patterns (n.d.). In *Refactoring.guru*. Retrieved from https://refactoring.guru/design-patterns/classification

Design Patterns (n.d.). In *Sourcemaking*. Retrieved from https://sourcemaking.com/design_patterns

Dooley J. (2017). *Software development, design and coding: With patterns, debugging, unit testing, and refactoring* (2nd ed.). Retrieved from https://books.google.com.ph/books?id=LGRADwAAQBAJ&dq=Software+Development,+Design+and+Coding:+With+Patterns,+Debugging,+Unit+Testing,+and+Refactoring

Joshi, B. (2016). *Beginning SOLID principles and design patterns for asp.net developers.* California: Apress Media, LLC.

What's a design pattern (n.d.). In *Refactoring.guru*. Retrieved from https://refactoring.guru/design-patterns/what-is-pattern