**What is a strategy pattern?**

A strategy pattern enables an algorithm to be selected at runtime. Each of these algorithms can be used interchangeably.

Behavioral Pattern – Patterns that are concerned with algorithms and the assignment of responsibilities between objects.

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

The strategy lets the algorithm vary independent from client that use it.

**Story Problem:**

One day you decided to create a navigation app for casual travelers.

The app was centered around a beautiful UI map that helped users quickly orient themselves around any city.

One of the most requested features for the app was automatic route planning.

A user should be able to enter an address and see the fastest route to a destination displayed on the map.

The first version of the map could only build routes over roads.

Those who traveled by car were happy with the app.

However, not everyone drives a car.

So in the next update - you added an option for walking routes.

Then the update after that - you added another option for public transportation

And then in the next update you added an option specifically for cyclists.

The business for the app was successful, but the technical part began causing headaches.

Each time a new routing algorithm was added, the main class of the navigator would double in size

It became a beast that was too difficult to maintain

Any change to one of the algorithms, regardless if it was a simple bug fix affected the entire class

This increased the change of creating an error in an already working code

Teamwork became time consuming and difficult

Your teammates would often complain that they spend too much time resolving any merge conflicts.

Implementing a new feature required you to make a change in a large class

This caused conflicts with the code produced by other teammates touching the same class.

**Solution:**

The purpose of a strategy pattern suggests that you take a class that does something specific and extracting them into smaller algorithms called strategies.

The original class (call context) must have a field for storing the references to one of the strategies 🚘🚴‍♂️🚶‍♀️

The context delegates the work one of the strategy objects instead of executing it on its own

The context class isn’t responsible for selecting an algorithm

The algorithm is selected when the client passes their desired path

The context doesn’t know much about the strategies.

It works with all strategies through the same generic interface

Where it only exposes a single method for triggering the algorithm that is encapsulated within the selected strategy

In our navigation app example, each routing algorithm 🚘🚴‍♂️🚶‍♀️strategy can be extracted to its own class.

Each class would have a method called build route that accepts origin and destination

Then returns a collection of the route’s checkpoints.

Though the parameter in the method are the same, each class would build a different route.

The navigator class doesn’t care about the algorithm selected since its only job is to render a set of checkpoints on a map.

The class also has a method for switching the active routing strategy where the client selects their preferred route type.

And can replace their selected routing behavior with another on the UI.

**Structure:**

Context maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.

Strategy interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

Concrete Strategies implement different variations of an algorithm the context uses.

The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn’t know what type of strategy it works with or how the algorithm is executed.

Client 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.

**Applicable:**

Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

* The Strategy pattern lets you indirectly alter the object’s behavior at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.

Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.

* The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.

* The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.

Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.

* The Strategy pattern lets you do away with such a conditional by extracting all algorithms into separate classes, all of which implement the same interface. The original object delegates execution to one of these objects, instead of implementing all variants of the algorithm.

**Implement:**

1. In the context class, identify an algorithm that’s prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.

2. Declare the strategy interface common to all variants of the algorithm.

3. One by one, extract all algorithms into their own classes. They should all implement the strategy interface.

4. In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.

5. Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.