Strategy Design Pattern

Strategy is a behaviour design pattern that is used to encapsulate algorithms designed for a specific task with the ability to change during runtime. While using an application that adopts this pattern, you will need to know the behaviour of each strategy and choose the one that fits closest to your need. One example of using strategy is that a player in a game can switch from conservative strategy to aggressive strategy if it gains more resources and becomes more powerful.

Strategy lets the algorithm vary independently from clients that use it, which means it decouples the algorithms from the one using the algorithm. Consider the implementation of a list or a collection. If you have a sorting algorithm built into it, you can never change the sorting algorithm during the runtime; suppose you have a new, faster sorting algorithm, you have to make changes to the list implementation; however, if you use strategy pattern to decouple them, inject different sorting algorithms without modifying the list object/class itself.

Motivation:

1. Want to support the open/close principle, which is *"open for extension, but closed for modification".* It is very easy to create additional behaviours but modification to main strategy interface is prohibited.
2. Different algorithms may be appropriate at different times. Want to switch between different strategies dynamically.
3. Make strategy code reuseable. Different clients may share and use some common strategies.

Applicability:

- When you want to be able to replace a behaviour at runtime (strategy reference can be dynamically altered).
- When you want to have a family of behaviours that might not be applicable for the client class.

Introduction [1]

Vocabulary:

Context:

- Is configured with a ConcreteStrategy object
- Contains reference to chosen strategy and invokes algorithm
- May define an interface that lets Strategy access its data

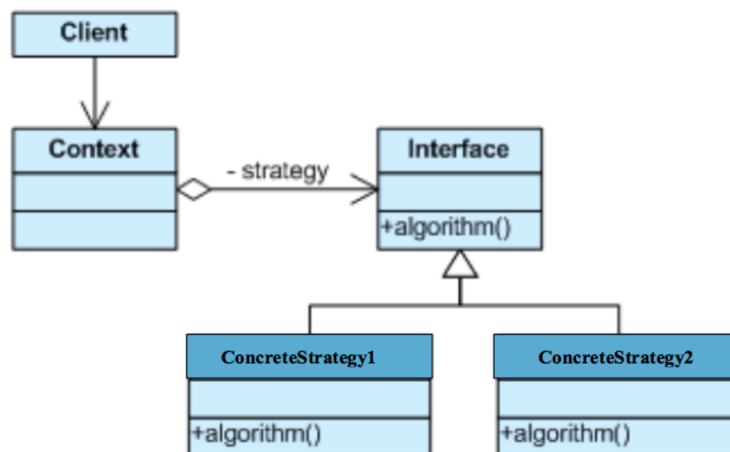
Strategy interface:

Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy:
implements algorithm. Context does not use any methods not defined in the Strategy interface.

Implementation:
Define the strategy interface and context.

1. Create strategy interface and reference in context.
2. Interact only through interface
3. Allow data access if necessary



[2]

Real Life Example

A real-life scenario where the strategy design pattern can be used is creating a recipe book to make burgers. First, we need to determine if our task will be enhanced by using this design.
We ask, Is there a specific task we need to accomplish?
Making burgers is the specific task, and each strategy we implement will be to accommodate a different variety of burger.

Our main strategy interface will have separate parts including customizeBuns, customizeMeat, customizeDressing, customizeVeggies, and customizeExtras.

My favorite type of burger is a chicken burger, and here is how to create the recipe strategy for a chicken burger.

ChickenBurgerStrategy

-override customizeBuns -> add Sesame buns
-override customizeMeats -> add Chicken
-override customizeDressing -> Mayonnaise, ketchup
-override customizeVeggies -> add Lettuce, Tomatoes
This strategy will now be used by a customer whenever they wish to order a chicken burger.

Advantages:

1. Allows you to have more options. Clients will have more choices at their disposal.
    - Relating to our burger joint, having more strategies will equate to having a bigger menu and more order choices for customers.
2. Encapsulating the algorithms into separate classes means algorithms can be easily changed, or new algorithms can be easily introduced.
    - Relating to our burger joint, it's very easy to add new burger toppings or change the recipe for one of the sauces without affecting the other ingredients.
3. Strategy allows you to switch strategies at run-time
    - Relating to the burger joint, you can switch tomatoes for pickles while serving customers.
    - We can easily modify our strategies for customizing burgers as well as add a new ones simply by inheriting from the main strategy interface.
4. Avoids duplicated code.
5. Strategies eliminates conditional statements by being able to choose the required algorithm without using "switch" or "if-else" statements.
6. Data structures used to implement the algorithms are encapsulated in the Strategy classes so you can change an algorithm's implementation without affecting the context class.

Disadvantages:

1. Prior knowledge about each individual strategy is needed before a client can select which one to use.
    - The customers must know the menu well to be able to choose an item they will like.
2. Clients must be aware of different Strategies. This means the client must understand how Strategies differ before it can select the appropriate one.
3. Communication overhead between Strategy and Context. This means the Strategy base class must expose the interface for all required behaviours.
4. Increased number of objects. Strategies increase the number of objects in an application which will increase the memory requirement and may have a performance impact.
5. The application needs to create and maintain two objects in the place of one since the Context and Strategy object need to be configured together.

Advantages/Disadvantages source: [3]

NFPs:

1. Testability
    - The alternative of the strategy pattern is using if/else conditional statements in code, which adds extra test complexity as additional branches are needed for extra logic conditions. Using the strategy pattern allows each new algorithm to only require one additional test class that tests only the modularized functionality that it provides.

1. Extensibility
   - When new algorithms are added for additional runtime cases, compared to if statements, it's simpler/easier to create an additional module/class that includes the algorithm for that specific case.
2. Maintainability
   - With the strategy pattern, it's easier to modify/debug code inside a self-contained module/class as opposed to a large amount of conditional branches in the same function for example.
3. Readability
   - More readable as each separate module/classes has its specific name and documentation
4. Reusability
   - Algorithm modules/classes can be reused in other parts of the program as its functionality is modularized.

Similar patterns:

Template, Decorator, State, Command

The state pattern share lots of similarities with the strategy design pattern. Although they solve different problems, they have similar structures but differ in intents. If you still remember, the State pattern allows an object to change its behaviours when its internal state changes. (Will annotate the graph on the board). Here, our strategy pattern define a family of algorithms, encapsulate each one, and make them interchangeable. Various algorithms are actually completing one common task.

Strategy pattern also shares some similarities with template and decorator. The template pattern find invariant part of different algorithms, put them in an abstract base class and the variant parts are supplied in concrete derived classes. The decorator attaches additional properties to core objects, it changes the skin while strategy change the core.

References:
[1] Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, *49*(120), p.11.
[2] https://cs.uwaterloo.ca/~rtholmes/teaching/2015winter/cs446/slides/SE2_17_design-patterns.pdf
[3] http://bobcash20.blogspot.ca/2007/05/strategy-pattern.html