Strategy Design Pattern

Purpose and Motivation

The strategy pattern is a behavioural software design pattern that allows selection of a specific algorithm at runtime. The basic idea here is that instead of duplicating code and writing the same algorithm all over the place, the code implements a specific strategy based on each client and based on this strategy, it executes one algorithm from a family of algorithms.
The purpose of the strategy pattern is to make the task of adding extra features and functionality to existing code very easy and convenient. This is done while ensuring that there is minimum modification done to the current code. The strategy pattern also allows the code to interchangeably pick and choose specific behaviours at runtime based on the pre-existing requirements.
The motivation behind using the strategy pattern is that it supports the “open/close” principle which states that code should be open for extension but closed for modification. Similarly, in the strategy pattern, it is very easy to extend and add extra strategies without modifying the strategy interface.

Intended Use Cases

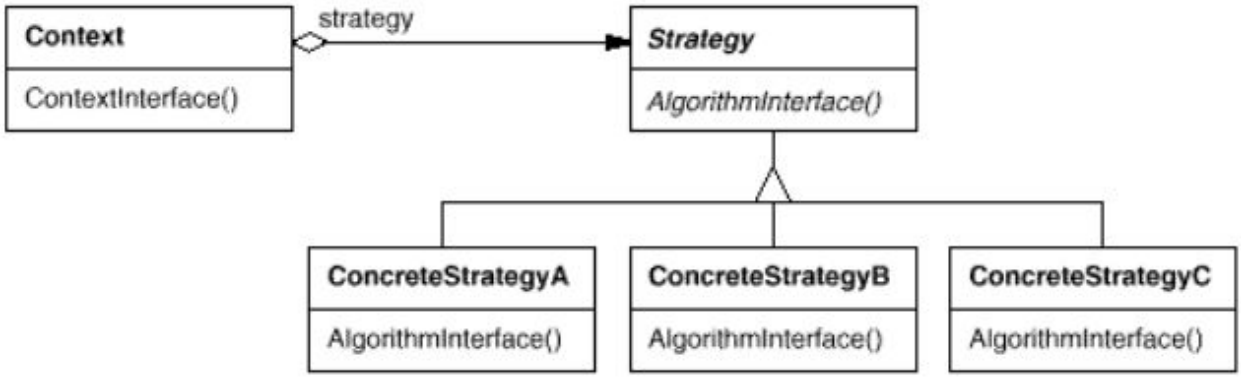
Use the Strategy pattern when

- many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviours, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Vocabulary

- **Strategy Interface:** An interface that provides a link between a concrete class and a context object. The interface usually consists of definitions of functions that are common to many concrete class objects.
- **Concrete Strategy classes:** These classes implement the strategy interface and provide an actual implementation to the algorithm in question. These implementations are very likely to differ from one another in a unique way.
- **Context objects:** The context objects provide a link between the client code or function and the concrete strategy class.

Structure and Runtime Behaviour



Known Consequences

Positive Consequences

- ***Strategies eliminate conditional statements***. The Strategy pattern offers an alternative to conditional statements for selecting desired behaviour.
- ***An alternative to subclassing***. Inheritance offers another way to support a variety of algorithms or behaviours. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behaviour they employ. Encapsulating

the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

- ***A choice of implementations***. Strategies can provide different implementations of the same behavior. The client can choose between strategies with different time and space trade-offs.

Negative Consequences

- ***Clients must be aware of different Strategies***. The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behaviour is relevant to clients.
- ***Communication overhead between Strategy and Context***. The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; simple ConcreteStrategies may use none of it! That means there will be times when the context creates and initializes parameters that never get used. If this is an issue, then you'll need tighter coupling between Strategy and Context.
- ***Increased number of objects***. Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object. Shared strategies should not maintain state across invocations.

NFPs

Improved NFPs

Reusability
Since the strategies are modularized, they can be used anywhere in the codebase that has the required context.

Readability
By eliminating if/else-if and switch blocks and separating the strategies into different classes we can greatly increase readability.

Extensibility

We can easily add new algorithms to an existing family of algorithms. Other algorithms in this family are not affected by the addition of a new algorithm. The algorithms can then be used interchangeably to alter application behaviour without changing its architecture.

Inhibited NFPs

Performance:
Each strategy requires us to create new classes which exponentially increases the number of objects defined in code. This negatively impacts application performance.

Credits

Students:

HoodieSzn (Ravindu Kandana, Anmol Shah, Tirman Sidhu, Hamza Usmani)

TEAM GOOSE (Anurag Joshi, Gunin Khanna, Xinjue Lu, Daniel Weisberg)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal