Design Pattern Research Homework

# Strategy DP

1. Intent of the Strategy DP

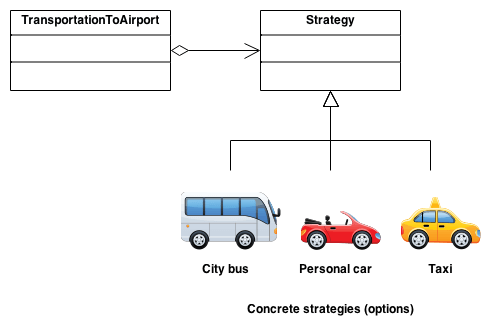
The intent is to define a set of algorithms and let them vary depending on the needs of the client that use it.

1. Explanation (more detailed)

From what I understood, Strategy DP is focused on two things: One is to define a set of algorithms (= strategies) and to make them available depending on the client’s needs. Second is to promote programming to an interface, not an implementation; i.e. it encourages the client to use an abstraction and bury the implementation details in derived classes.

We can see this example in practice with the interfaces List, Map & Set. Depending on your needs, you can either use ArrayList, LinkedList etc. for lists or TreeSet, HashSet for sets, depending on which implementation is more convenient to you.

In this case, the different implementations would be called strategies and they are interchange-able i.e. you can easily switch from a HashSet to a TreeSet if need be.



<https://sourcemaking.com/design_patterns/strategy>

https://www.tutorialspoint.com/design\_pattern/strategy\_pattern.htm

<https://en.wikipedia.org/wiki/Strategy_pattern>

https://dzone.com/articles/design-patterns-strategy

1. Related patterns

* Similar to Decorator pattern.
* Strategy objects often make good Flyweights.

1. Common situations of use

For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known until run-time and may require radically different validation to be performed. The validation algorithms (strategies), encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without [code duplication](https://en.wikipedia.org/wiki/Duplicate_code).

As for **extensibility**, imagine you are writing a framework, where users are supposed to be able to inject their own behavior. For example, you want to create some kind of tax calculations framework and want to support tax systems of different countries. Instead of implementing all of them, you just want to give framework users a chance to provide an implementation of how to calculate some particular taxes.

1. Can be mistaken with

* Decorator

# Façade DP

1. Intent of the Façade DP

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

1. Explanation (more detailed)

A facade is an object that provides a simplified interface to a larger body of code, such as a [class library](https://en.wikipedia.org/wiki/Class_library). A facade can:

* make a [software library](https://en.wikipedia.org/wiki/Software_library) easier to use, understand, and test, since the facade has convenient methods for common tasks
* make the library more readable, for the same reason
* reduce [dependencies](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system
* wrap a, subjectively, poorly-designed collection of [APIs](https://en.wikipedia.org/wiki/Application_programming_interface) with a single well-designed API

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

<https://www.geeksforgeeks.org/facade-design-pattern-introduction/>

<http://www.dofactory.com/net/facade-design-pattern>

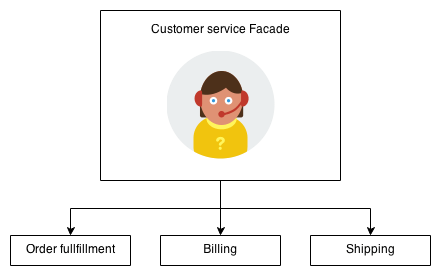
https://www.tutorialspoint.com/design\_pattern/facade\_pattern.htm

1. Related patterns

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

Adapter and Facade are both wrappers; but they are different kinds of wrappers. The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface.

1. Common situations of use



A real world example can be when you call the customer support number of any given company, because you will be transferred to an operator which acts as a façade between you and your desired objective (paying bills, making contract changes etc)

1. Can be mistaken with

Flighweight pattern. Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.