**Strategy Pattern**

**The Pattern Problem**

Given a context in which an algorithm is used, a programmer wants to use different algorithms depending upon the context and avoiding having to change the context (client).

* Reusability

This pattern is reusable because you can change the used strategy on run time. Additionally, this pattern is very useful when we must make objects that must be able to do the same task in lot of different ways or in a different behavior.

* Extensibility

This Pattern allows the programmer to extend the algorithm to his liking. This Pattern is extensible because he can change whichever algorithm, of course which implement the interface, to his liking or make minor adjustments, without hindering with the functionality of the others. Every algorithm is independent from the others, by making changes to one algorithm won’t change the others. They only share a signature.

* Maintainability

It makes some effort to maintain the code because of the reason that the programmer must have extensive knowledge of the classes. Additionally, the number of the classes may increase. However, because the classes are independent (encapsulated) and initiated in the context through the interface, it makes it easy to control and change an algorithm without destroying the integrity of the other strategies.

**Design Pattern Solution**

**Consequences**

* Positive consequences

1. Allows hot swapping algorithms at runtime.
2. Isolates the code and data of the algorithms from the other classes.
3. Replaces inheritance with delegation.
4. Follows the Open/Closed Principle.

* Negative consequences

1. Increases overall code complexity by creating multiple additional classes.
2. Client must be aware of the differences between strategies to pick a proper one.
3. Communication overhead