Used to manage algorithms, relationships and responsibilities between objects.

**GOF Definition:**

**Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior**

**Essentially, the strategy pattern allows us to change the behavior of an algorithm at runtime.**

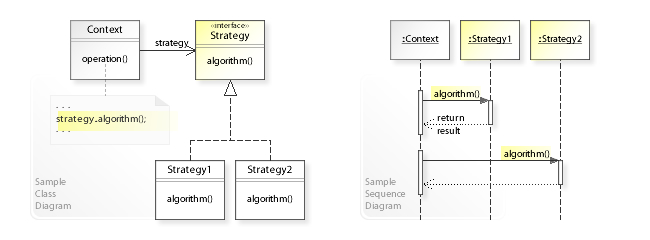
**Context**is composed of a **Strategy**. The context could be anything that would require changing behaviors - a class that provides sorting functionality perhaps. The **Strategy** is simply implemented as an interface, so that we can swap **ConcreteStrategy**s in and out without affecting our **Context.**

**Where Would I Use This Pattern?**

The Strategy pattern is to be used where you want to choose the algorithm to use at runtime. A good use of the Strategy pattern would be saving files in different formats, running various sorting algorithms, or file compression.

The Strategy pattern provides a way to define a family of algorithms, encapsulate each one as an object, and make them interchangeable.

UML Class and Sequence Diagram:



**Strategy Vs Template Method:**

The key **difference** is that **Strategy Pattern** is about modifying a behavior of a context in runtime using **strategies**, while **Template Method Pattern** is about following a skeleton implementation of an algorithm and modifying its behavior by overriding methods of the skeleton class **in the** subclasses.

The main difference between the two is when the concrete algorithm is chosen.

With the **Template method pattern** this happens at compile-time by subclassing the template. Each subclass provides a different concrete algorithm by implementing the template's abstract methods. When a client invokes methods of the template's external interface the template calls its abstract methods (its internal interface) as required to invoke the algorithm.

class ConcreteAlgorithm : AbstractTemplate

{

void DoAlgorithm(int datum) {...}

}

class AbstractTemplate

{

void run(int datum) { DoAlgorithm(datum); }

virtual void DoAlgorithm() = 0; // abstract

}

In contrast, the **Strategy pattern** allows an algorithm to be chosen at runtime by containment. The concrete algorithms are implemented by separate classes or functions which are passed to the strategy as a parameter to its constructor or to a setter method. Which algorithm is chosen for this parameter can vary dynamically based on the program's state or inputs.

class ConcreteAlgorithm : IAlgorithm

{

void DoAlgorithm(int datum) {...}

}

class Strategy

{

Strategy(IAlgorithm algo) {...}

void run(int datum) { this->algo.DoAlgorithm(datum); }

}

## In summary:

* Template method pattern: **compile-time** algorithm selection by **subclassing**
* Strategy pattern: **run-time algorithm** selection by **containment**

**Strategy Vs Bridge:**

The Bridge Pattern makes a distinction between an abstraction and an implementation in such a way that the two can vary independently.

The Strategy on the other hand is concerned with changing the behavior of an object at run time

These two patterns solve different problems. The strategy is concerned with making algorithms interchangeable while the Bridge is concerned with decoupling the abstraction from the implementation so that you can provide multiple implementations for the same abstraction. That is, the bridge is concerned with entire structures.

**Strategy Vs State:**

1. **Intent of the Pattern**: Intent or purpose of Strategy Pattern is to have a family of interchangeable algorithms which can be chosen based on the context and/or client needs. On the other hand, State Pattern’s intent is to manage states of the object along with object’s behavior which changes with its state.
2. **Client Awareness of the strategy/state**: In a Strategy Pattern implementation the strategy chosen is client-dependent and hence the client is aware which strategy is being used. However, in State Pattern implementation client interacts with the context to act on the object but does not decide on which State to chose. The Object itself appears to change its State Class based on the interaction the Client has via the Context.
3. **Reference back to the Context**: Every state in the State Pattern holds a reference back to the Context. However, each strategy does not hold the handle back to the Context in the Strategy Pattern.
4. **Relation between individual states/individual strategies**: Different states in the State Pattern are related to one another, say as a successor or predecessor etc. This is because there is a flow among the states like a Finite State Machine. Strategy Pattern, however, just chooses one of the strategies from the multiple strategies available. There is no successor/predecessor relationship between strategies.
5. **How v/s What & When**: Multiple strategies define multiple ways of *how*to do something.On the other hand, multiple states define *what*is to be done and based on the relationship between states *when*it is to be done.