**Factory Method**

**Intent**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Also known as**

Virtual Constructor

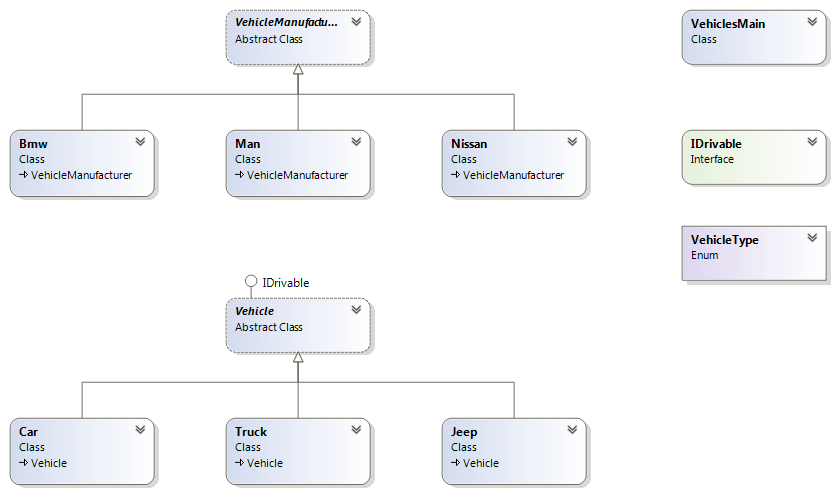
**Motivation**

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

**Applicability**

1. a class can't anticipate the class of objects it must create.
2. a class wants its subclasses to specify the objects it creates.
3. classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

**Class Diagram**



**Collaborations**

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.

**Consequences**

1. Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.
2. Provides hooks for subclasses. Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.
3. clients can find factory methods useful, especially in the case of parallel class hierarchies. It localizes knowledge of which classes belong together.

**Related Patterns**

* Abstract Factory is often implemented with factory methods.
* Factory methods are usually called within Template Methods.
* Prototypes don't require subclassing Creator, but they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. FactoryMethod doesn't require such an operation.

# Proxy Pattern

## Intent

Provide a surrogate or placeholder for another object to control access to it.

## Also known as

Surrogate

## Motivation

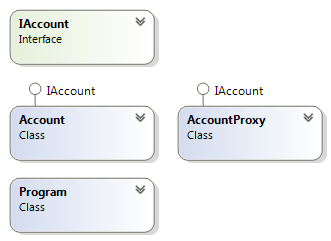
One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.

## Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.   
Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space.
2. A **virtual proxy** creates expensive objects on demand.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.
4. A **smart reference** is a replacement for a bare reference that performs additional actions when an object is accessed.

## Class Diagram



## Collaborations

Proxy forwards requests to Real Subject when appropriate, depending on the kind of proxy.

## Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

## Related Patterns

* Adapter: An adapter provides a different interface to the object it adapts.
* Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

# ****Strategy**** Design Pattern

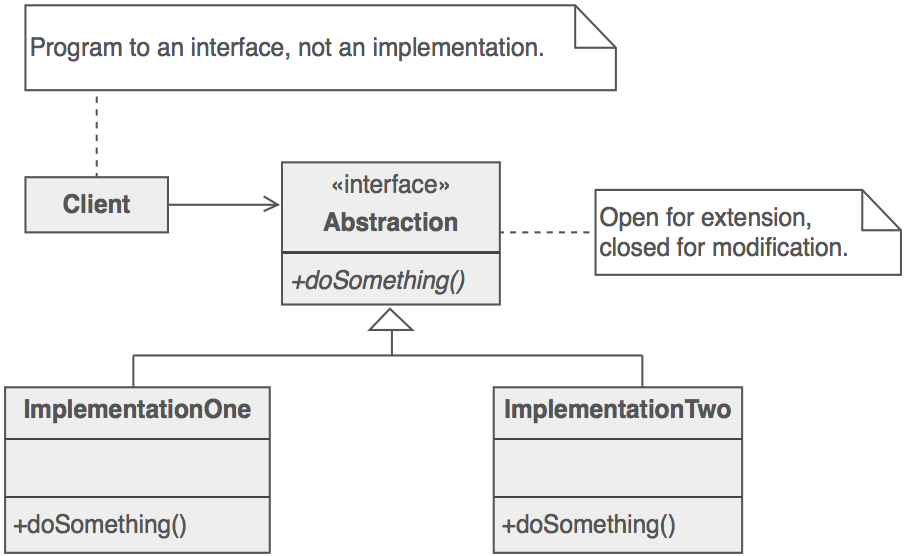
### Intent

* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
* Capture the abstraction in an interface, bury implementation details in derived classes.

### Problem

One of the dominant strategies of object-oriented design is the "open-closed principle".

Figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.



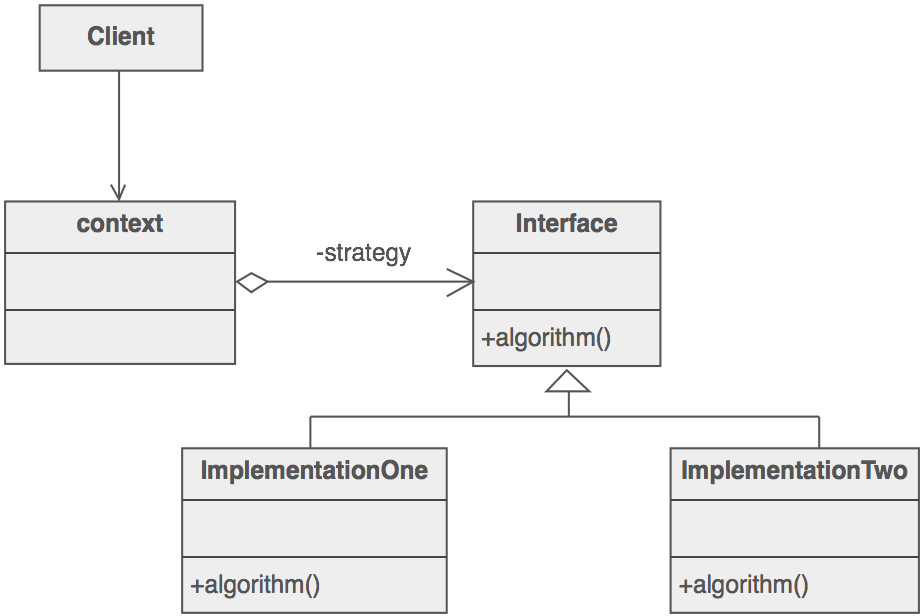
A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

A more popular characterization of this "abstract coupling" principle is "Program to an interface, not an implementation".

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

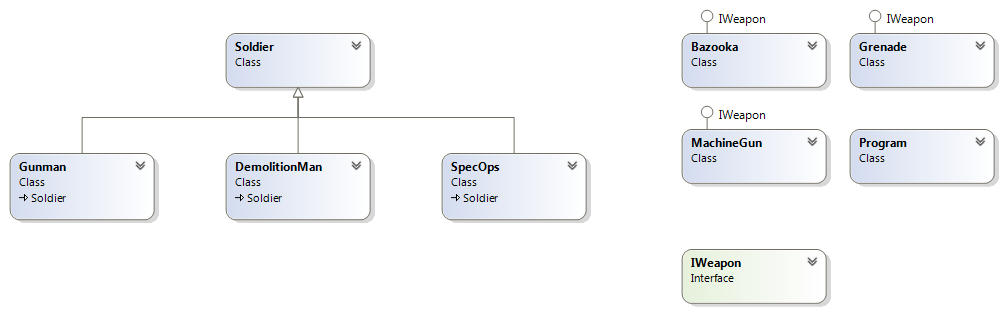
### Structure

The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.



### Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.



### Check list

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.

### Rules of thumb

* Strategy is like Template Method except in its granularity.
* State is like Strategy except in its intent.
* Strategy lets you change the guts of an object. Decorator lets you change the skin.
* State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.
* Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
* Strategy objects often make good Flyweights.