Design Patterns

Command Design Pattern

Intent

> Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
>
> Promote “invocation of a method on an object” to full object status
>
> An object-oriented callback

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Discussion

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an `execute()` method that simply calls the action on the receiver.

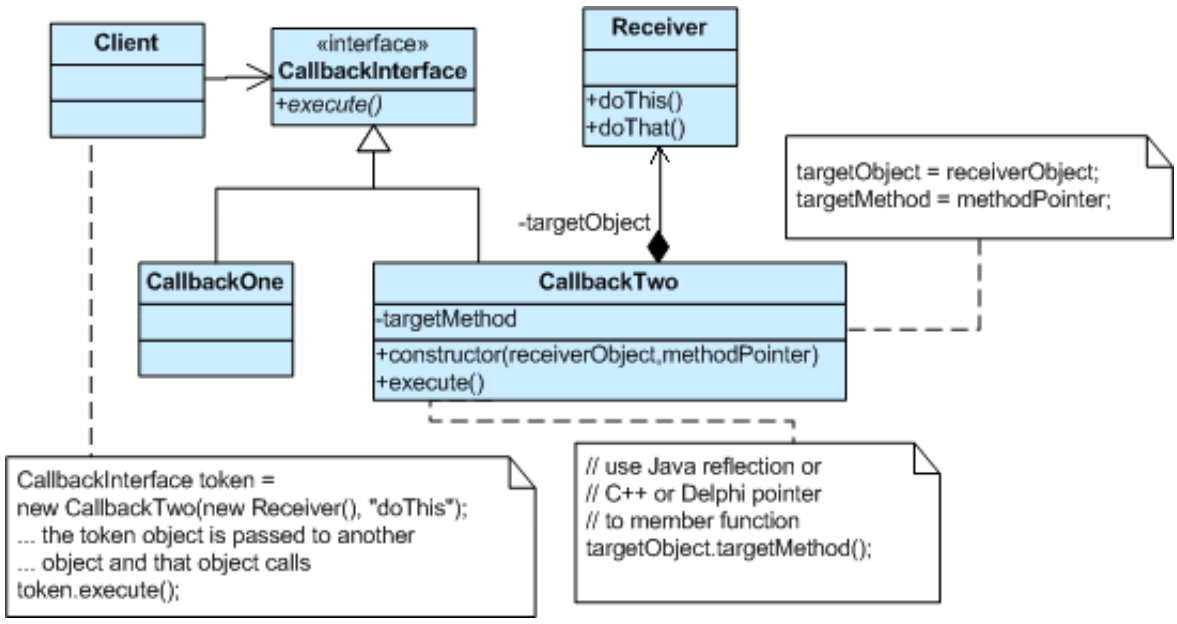
All clients of Command objects treat each object as a “black box” by simply invoking the object’s virtual `execute()` method whenever the client requires the object’s “service”.

A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command’s “execute” method then causes the pieces to come together.

Sequences of Command objects can be assembled into composite (or macro) commands.

Structure

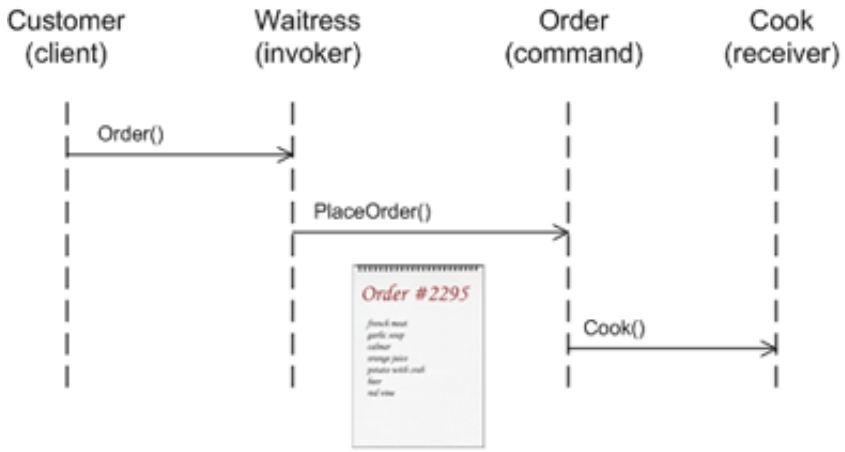
The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.



Command objects can be thought of as “tokens” that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The “check” at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of “checks” used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



Decorator Design Pattern

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Client-specified embellishment of a core object by recursively wrapping it.

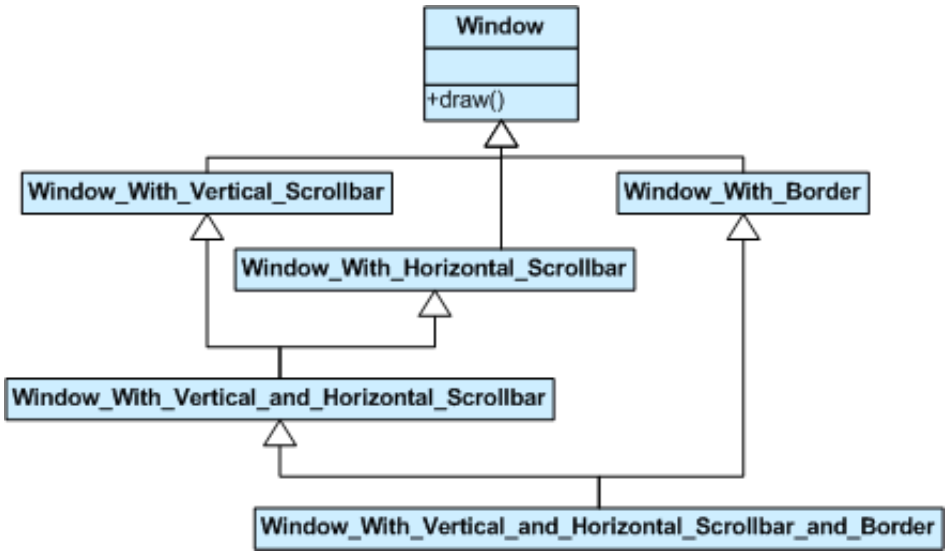
Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

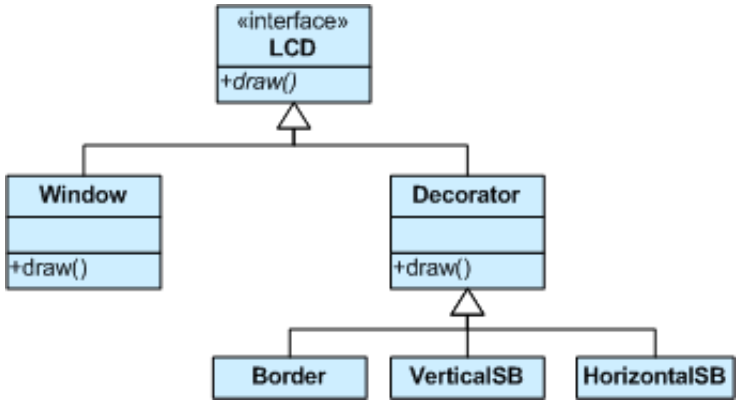
Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like ...



But the Decorator pattern suggests giving the client the ability to specify whatever combination of “features” is desired.

```
Widget*  aWidget = new BorderDecorator(
                     new HorizontalScrollBarDecorator(
                       new VerticalScrollBarDecorator(
                         new Window( 80, 24 ))));
aWidget->draw();
```

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like ...

```
Stream*  aStream = new CompressingStream(
                     new ASCII7Stream(
                       new FileStream( "fileName.dat" )));
aStream->putString( "Hello world" );
```

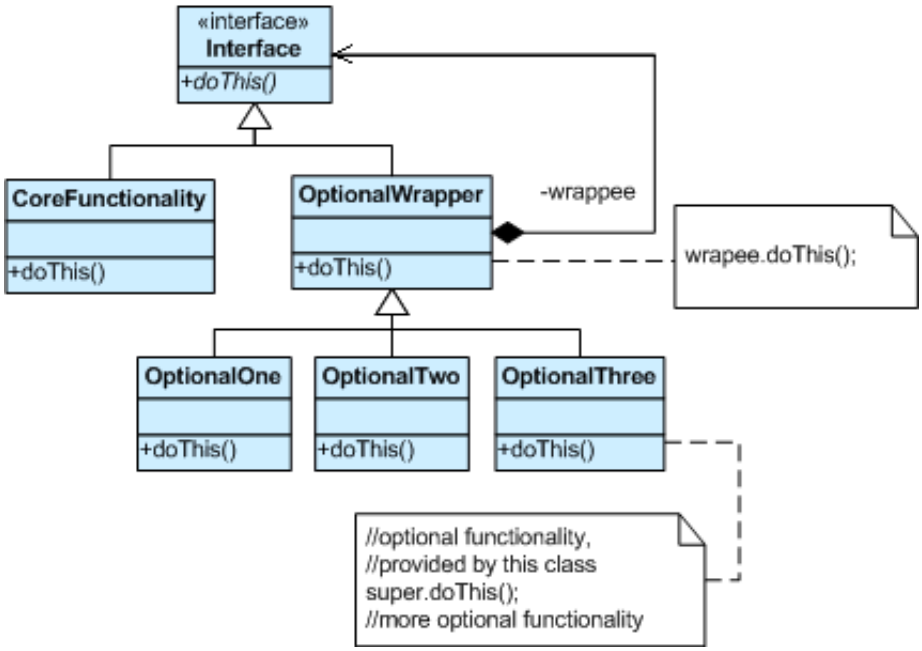
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Structure

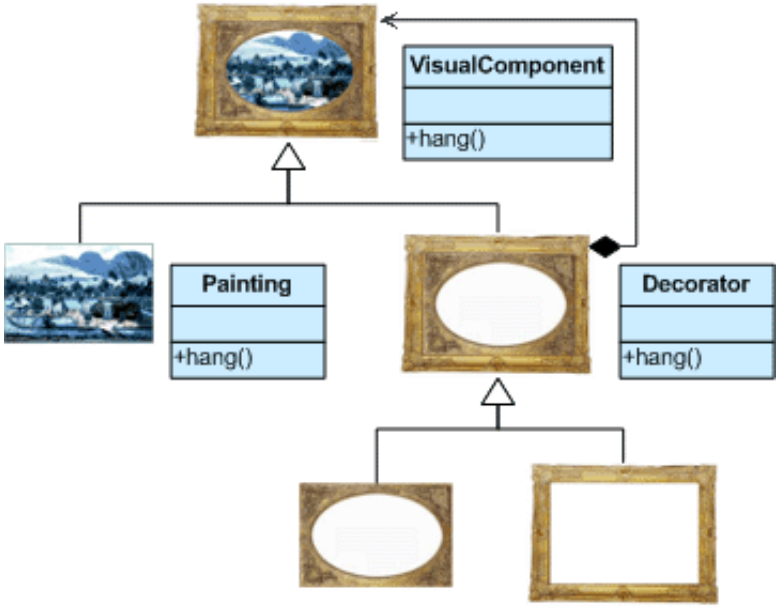
The client is always interested in `CoreFunctionality.doThis()`. The client may, or may not, be interested in `OptionalOne.doThis()` and `OptionalTwo.doThis()`. Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.


VisualComponent
+hang()
Painting
+hang()
Decorator
+hang()

Factory Method Design Pattern

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.
>
> Defining a “virtual” constructor.
>
> The `new` operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual “placeholders” for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

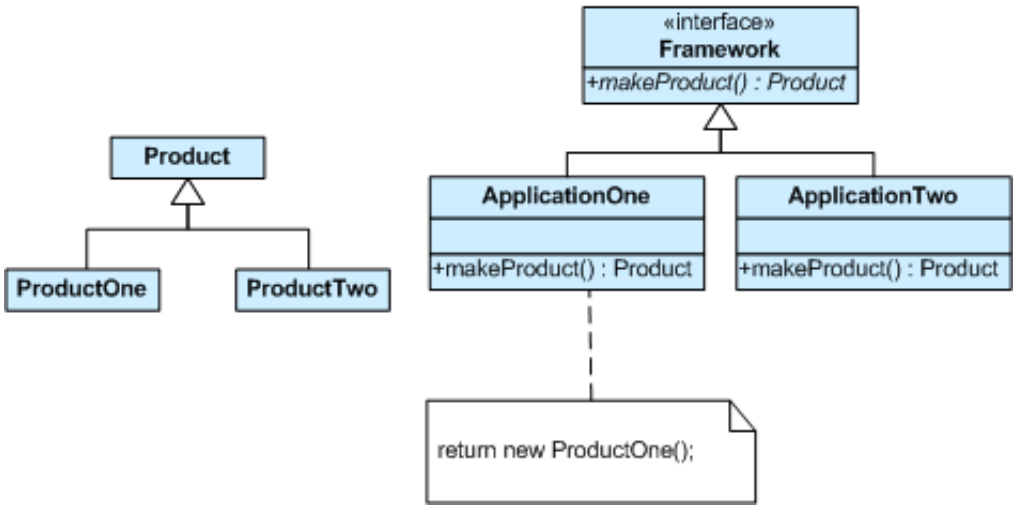
People often use Factory Method as the standard way to create objects; but it isn’t necessary if: the class that’s instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

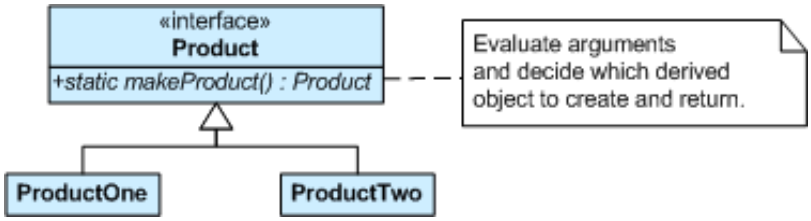
Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Structure

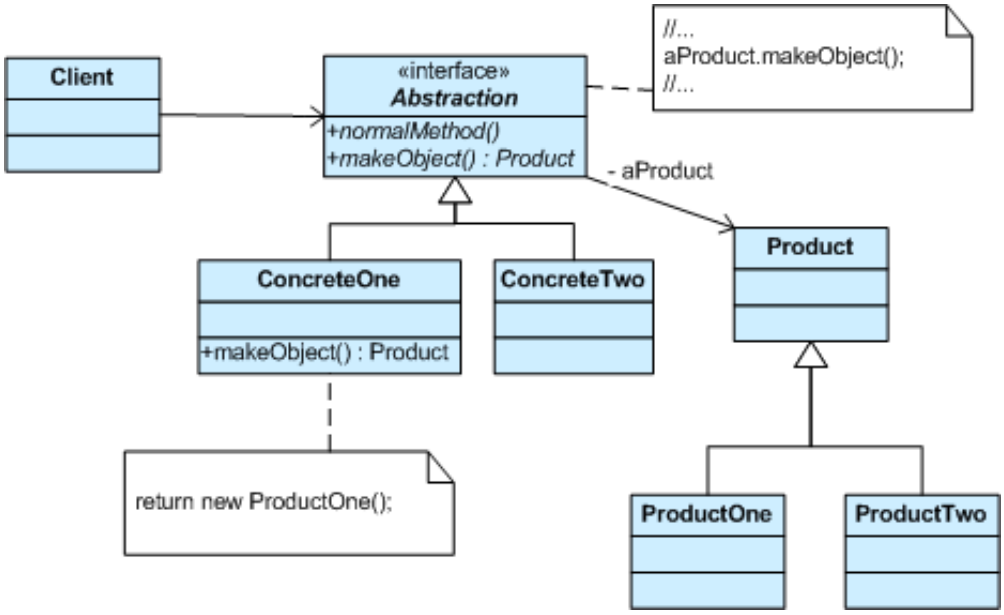
The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.



An increasingly popular definition of factory method is: a `static` method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g. `Color.make_RGB_color(float red, float green, float blue)` and `Color.make_HSB_color(float hue, float saturation, float brightness)`



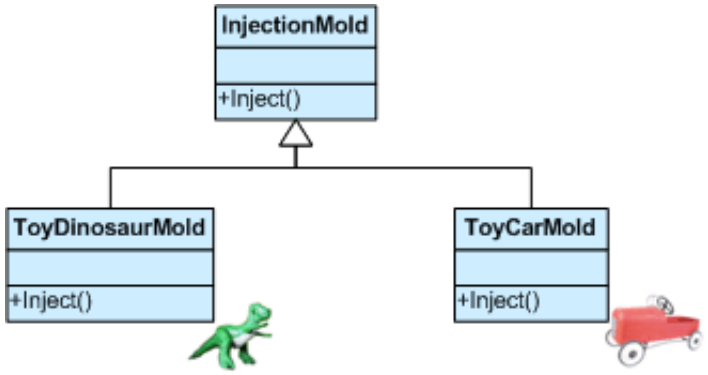
The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



Example

[11]

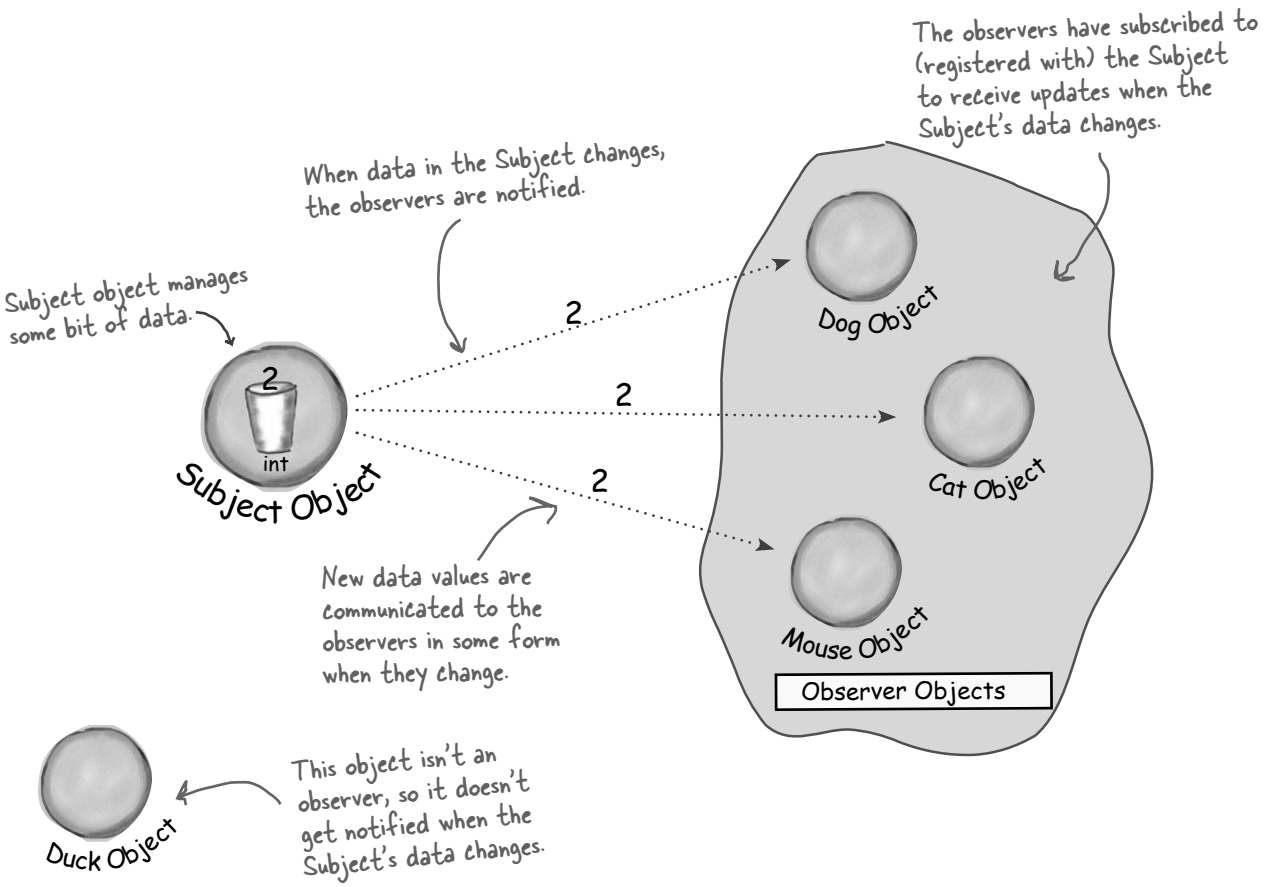
The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Publishers + Subscribers = Observer Pattern

If you understand newspaper subscriptions, you pretty much understand the Observer Pattern, only we call the publisher the SUBJECT and the subscribers the OBSERVERS.

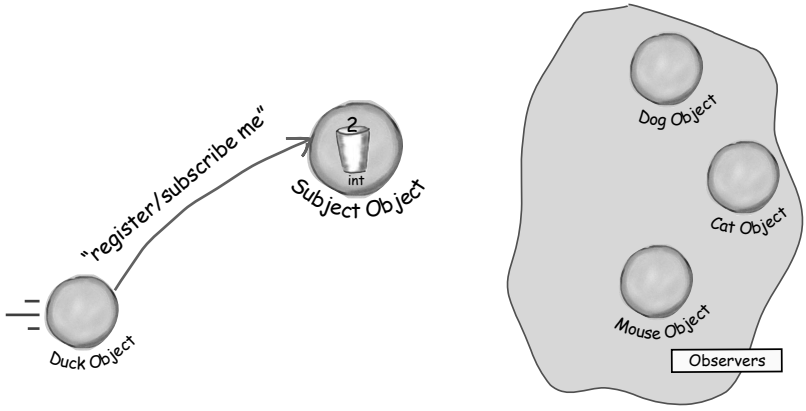
Let's take a closer look:



A day in the life of the Observer Pattern

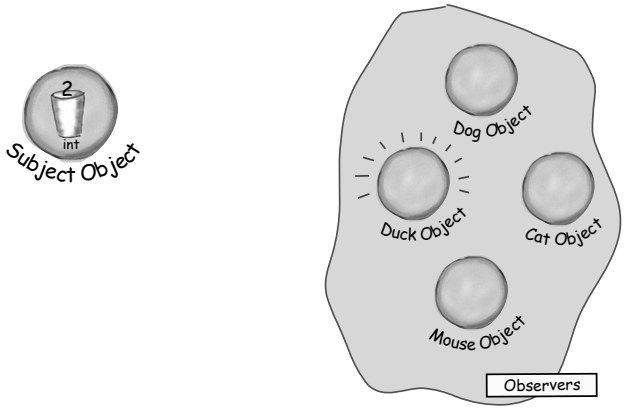
A Duck object comes along and tells the Subject that it wants to become an observer.

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...



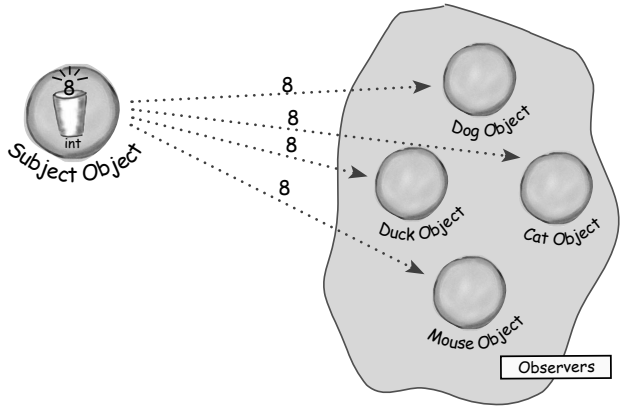
The Duck object is now an official observer.

Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.



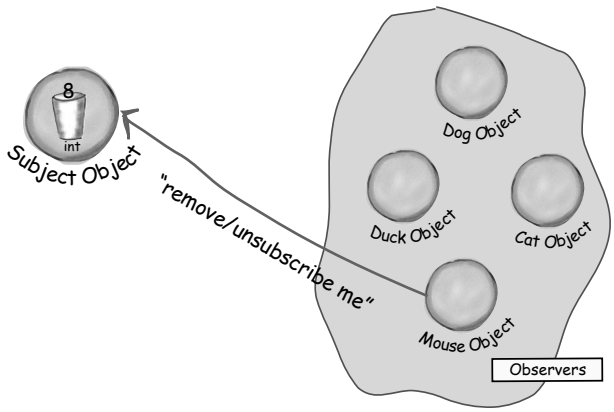
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



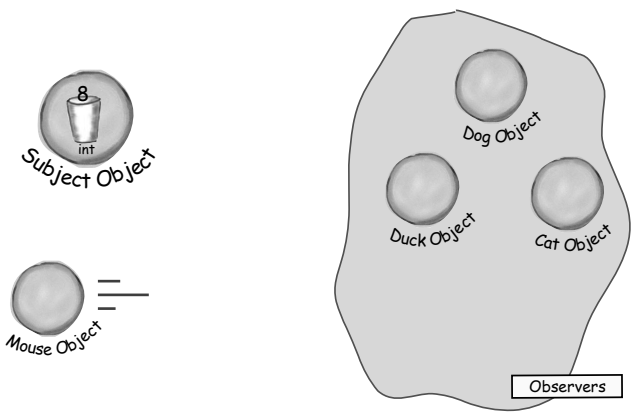
The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.



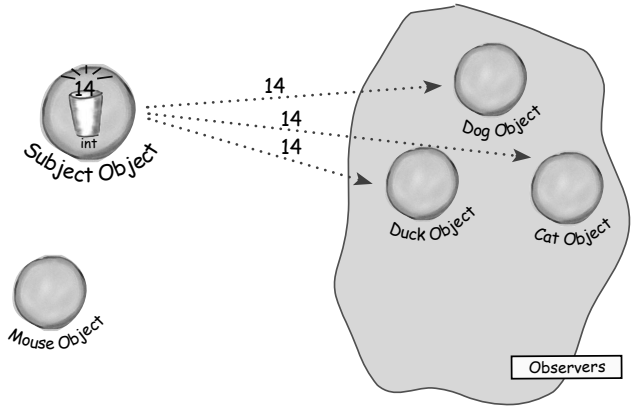
Mouse is outta here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.



The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.



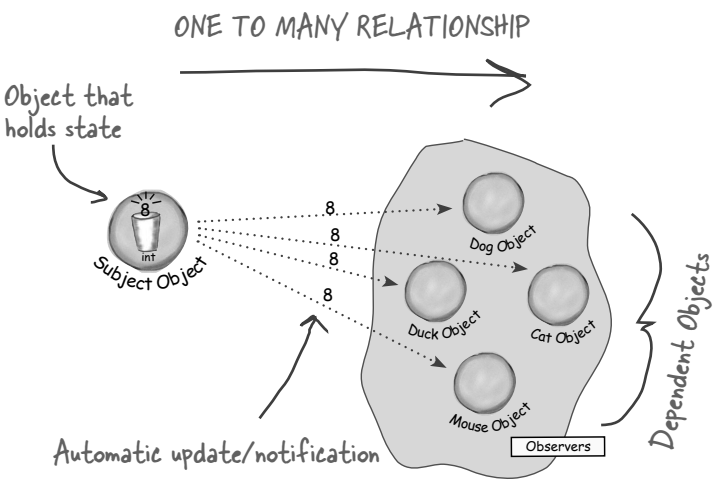
The Observer Pattern defined

When you're trying to picture the Observer Pattern, a newspaper subscription service with its publisher and subscribers is a good way to visualize the pattern.

In the real world however, you'll typically see the Observer Pattern defined like this:

> **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Let's relate this definition to how we've been talking about the pattern:



The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

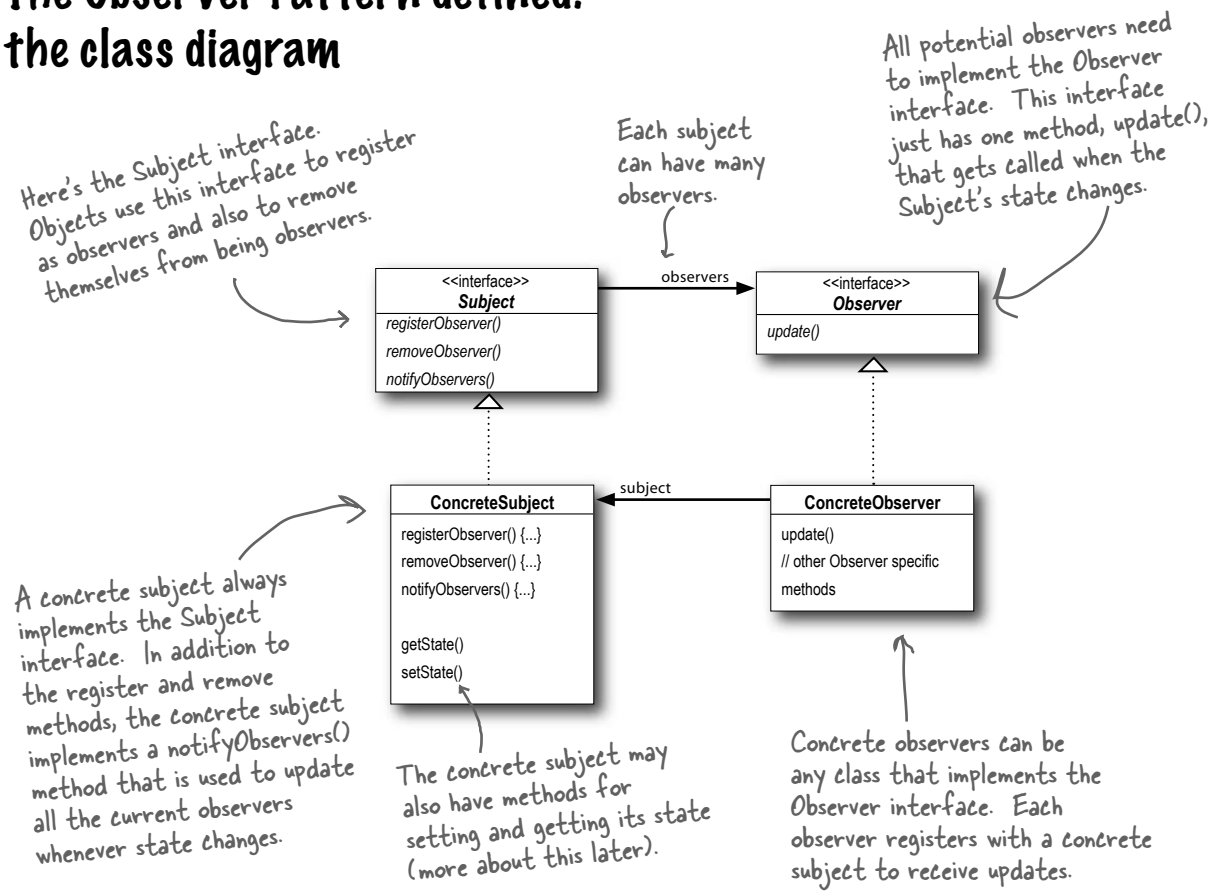
As you'll discover, there are a few different ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer interfaces.

Let's take a look...

The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

The Observer Pattern defined: the class diagram



there are no Dumb Questions

Q: What does this have to do with one-to-many relationships?

A: With the Observer pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they don't own it. There are many observers and they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE Subject to the MANY Observers.

Q: How does dependence come into this?

A: Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

Proxy Design Pattern

Intent

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

Use an extra level of indirection to support distributed, controlled, or intelligent access.

Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

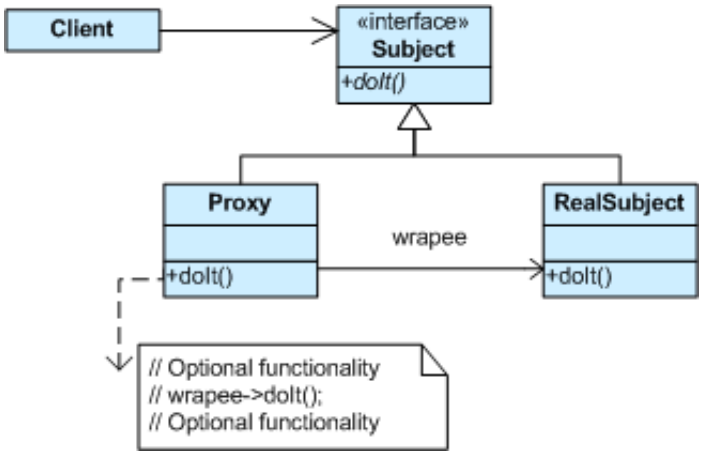
1. A virtual proxy is a placeholder for “expensive to create” objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the “stub” code in RPC and CORBA provides.
3. A protective proxy controls access to a sensitive master object. The “surrogate” object checks that the caller has the access permissions required prior to forwarding the request.
4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:
   Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),

   Loading a persistent object into memory when it’s first referenced,

   Checking that the real object is locked before it is accessed to ensure that no other object can change it.

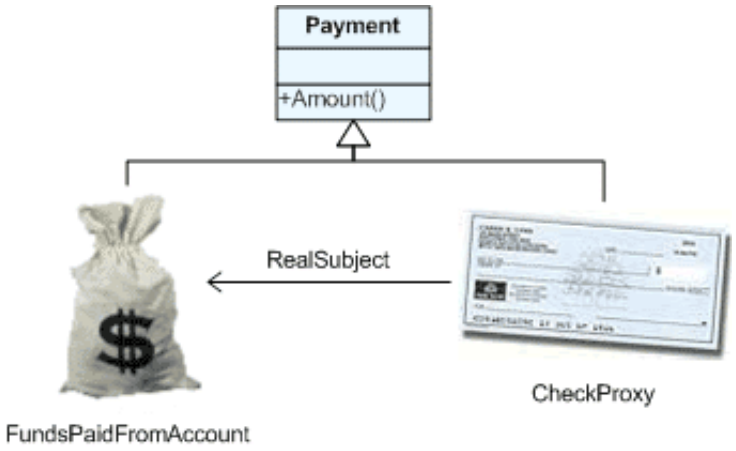
Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



State Design Pattern

Intent

> Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
>
> An object-oriented state machine
>
> wrapper + polymorphic wrappee + collaboration

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterixed by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

> Define a "context" class to present a single interface to the outside world.
>
> Define a State abstract base class.
>
> Represent the different "states" of the state machine as derived classes of the State base class.
>
> Define state-specific behavior in the appropriate State derived classes.
>
> Maintain a pointer to the current "state" in the "context" class.
>
> To change the state of the state machine, change the current "state" pointer.

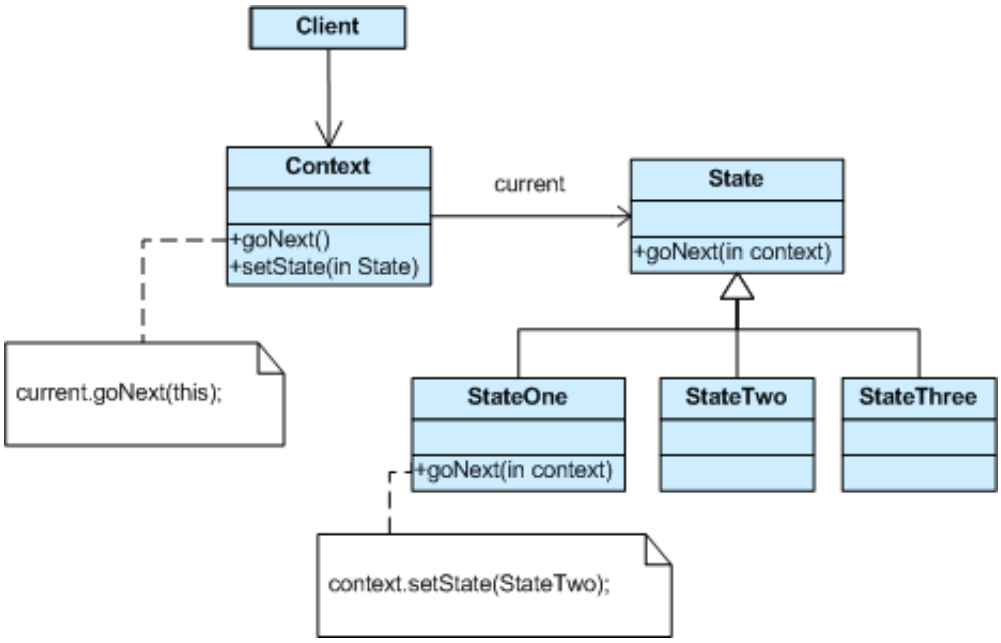
The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based

approach uses code (instead of data structures) to specify state transitions, but it does a good job of accomodating state transition actions.

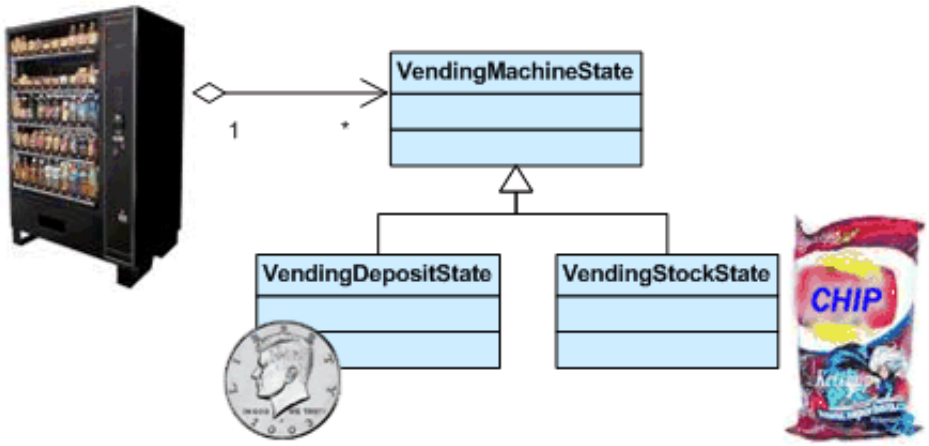
Structure

The state machine's interface is encapsulated in the "wrapper" class. The wrappee hierarchy's interface mirrors the wrapper's interface with the exception of one additional parameter. The extra parameter allows wrappee derived classes to call back to the wrapper class as necessary. Complexity that would otherwise drag down the wrapper class is neatly compartmented and encapsulated in a polymorphic hierarchy to which the wrapper object delegates.



Example

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.


VendingMachineState
1
*
VendingDepositState
VendingStockState
CHIP

Strategy Design Pattern

Intent

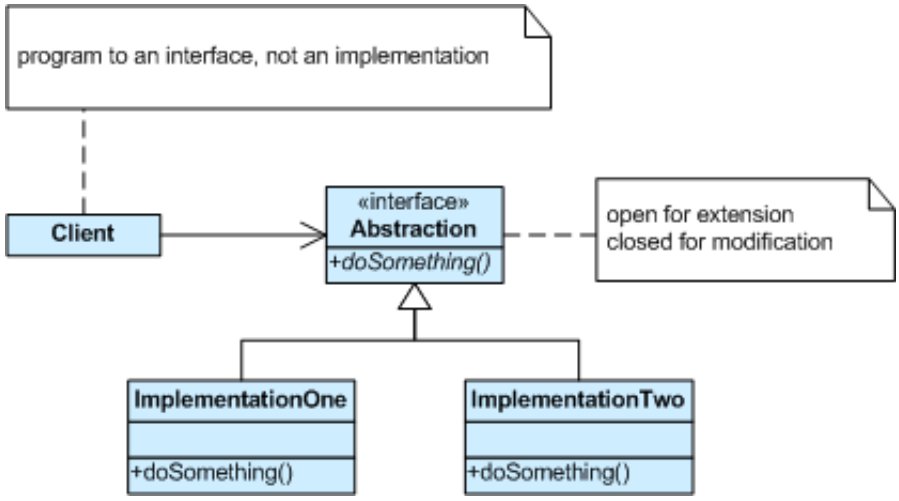
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

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.



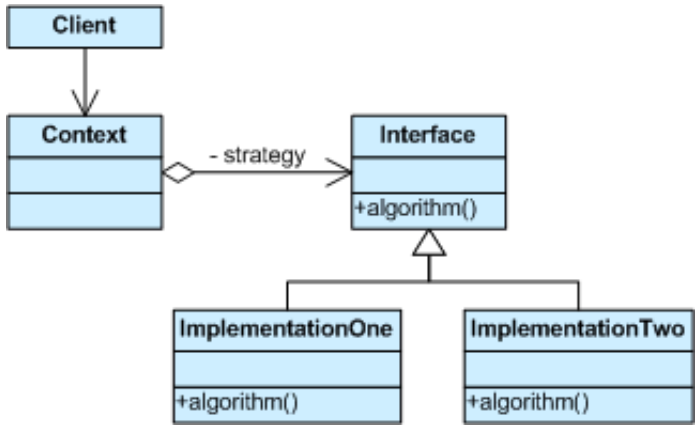
A generic value of the software community for years has been, “maximize cohesion and minimize coupling”. The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of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

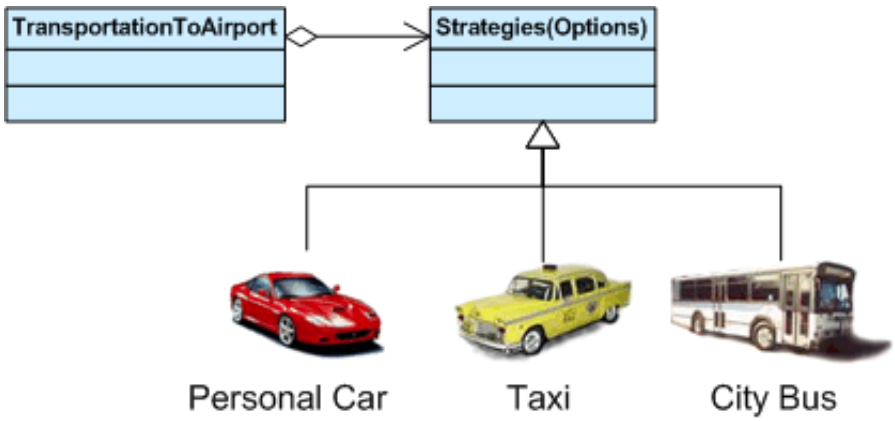
Structure

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.


TransportationToAirport
Strategies(Options)
Personal Car
Taxi
City Bus

Template Method Design Pattern

Intent

> Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
>
> Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

Problem

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

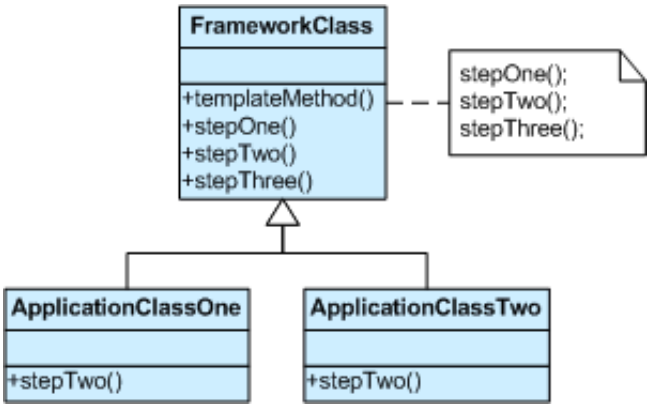
Discussion

The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

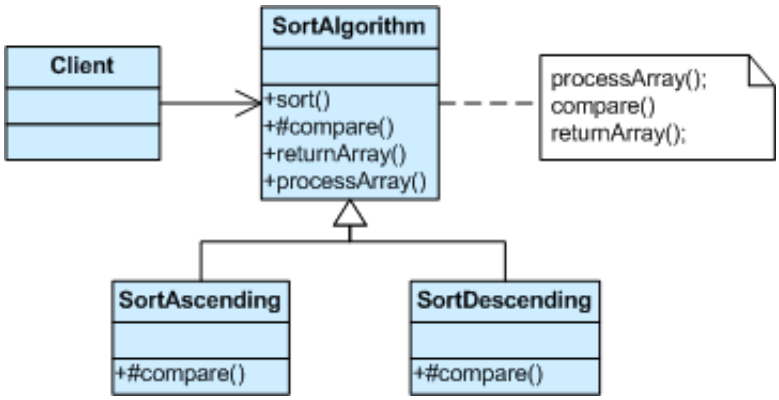
The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun". This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Structure

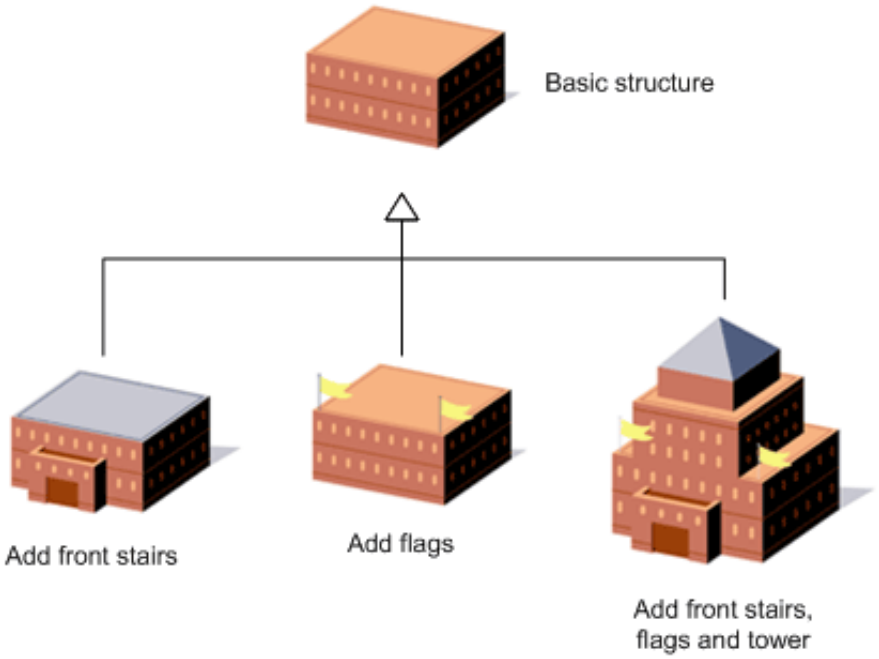


The implementation of `template_method()` is: call `step_one()`, call `step_two()`, and call `step_three()`. `step_two()` is a "hook" method – a placeholder. It is declared in the base class, and then defined in derived classes. Frameworks (large scale reuse infrastructures) use Template Method a lot. All reusable code is defined in the framework's base classes, and then clients of the framework are free to define customizations by creating derived classes as needed.



Example

The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the later stages of construction to produce a wider variety of models.


Basic structure
Add front stairs
Add flags
Add front stairs,
flags and tower