Notes on Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

- Bridge
- Chain of Responsibility
- Command
- Composite
- Decorator
- Façade
- Factory Patterns - General
- Factory Method
- Abstract Factory
  - *Notes on Factories*
- Functor
- Iterator
- Mediator
- Observer
- Proxy
- Singleton
- State
- Strategy
- Template Method
- Visitor
  - Extrinsic Visitor
  - Overloaded Visitor
  - *visitor notes*
- Discussion & Notes

Functor Pattern
functions as objects

Intent

They serve the role of a function, but can be created, passed as parameters, and manipulated like objects.

Since OO programs have no (naked) functions, only methods (of a class), to pass a function around we encapsulate it as a method of an object, using a standard method name, and pass the object.

Used in the Command pattern; with the enhancements of possible multiple functions, and some local state.

Known Examples

The Java comparator interface; *java.util.Comparator*; defines an interface for objects that act like functions pointers to compare objects.

Typically used for callbacks (see: *Observer*, *visitor*, and *command* patterns).

Example

```
public class Demo {
        public static bool compare ( int x, int y, Comparator c ) {
                return c.compare (x, y);
                }

        public static void main ( String args[ ]  ) {
                Comparator c1 = new maxComparator();
                bool result = compare (2, 3, c1);       // returns FALSE

                Comparator c1 = new minComparator();
                bool result = compare (2, 3, c2);       // returns TRUE
        }
```

```
public interface Comparator {
        bool compare ( int, int);
        }

public class maxComparator implements Comparator {
        bool compare (int x, int y) {
                return (x>y);
                }
        }
```

Also-Known-as

function object

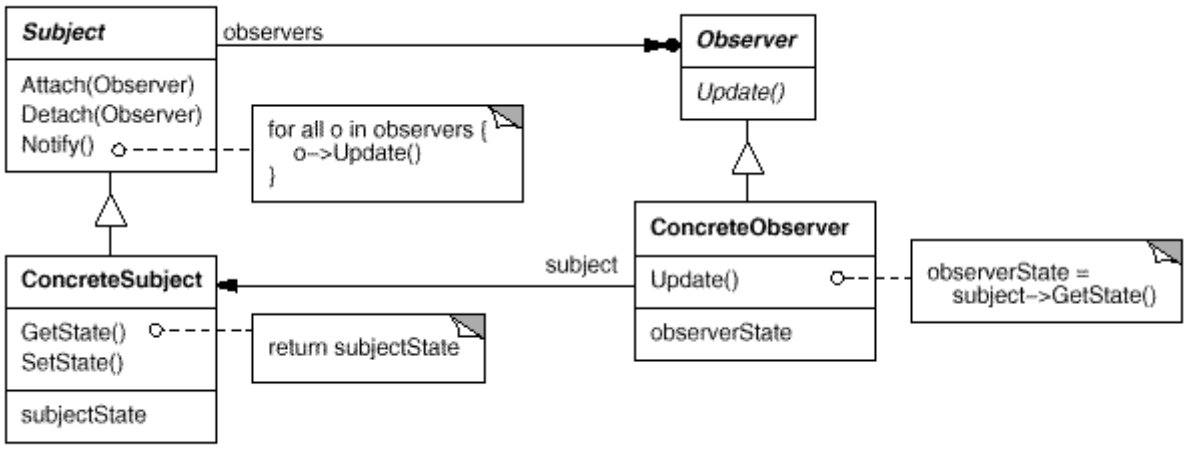
Notes:

- Compare to function pointers:
  - can use standard interface and inheritance to share implementations.

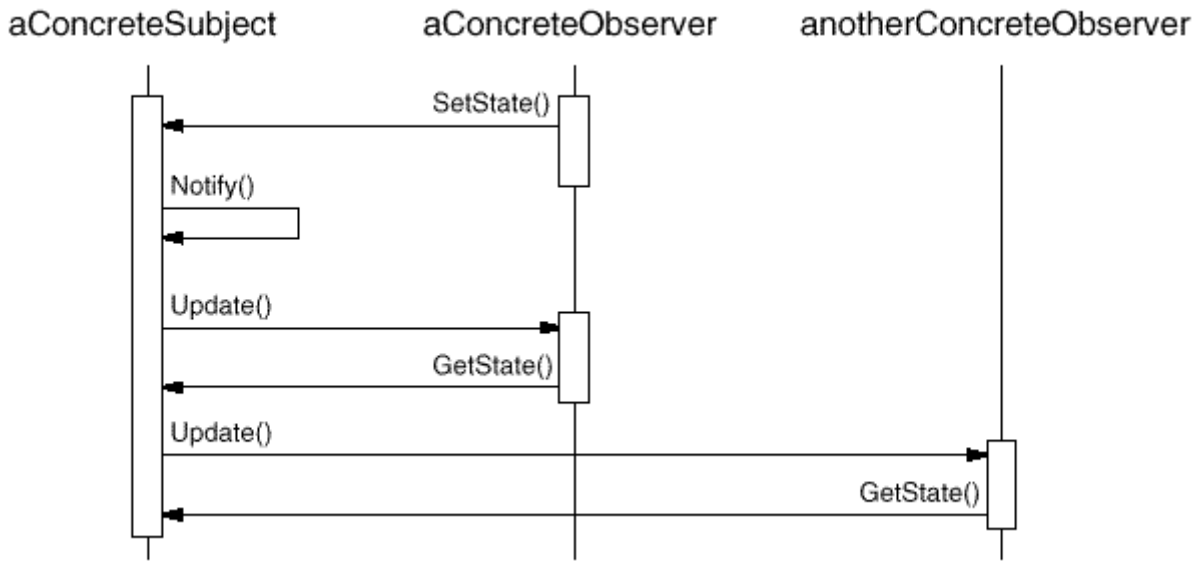
Observer Pattern

Intent
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Structure:



Interactions:



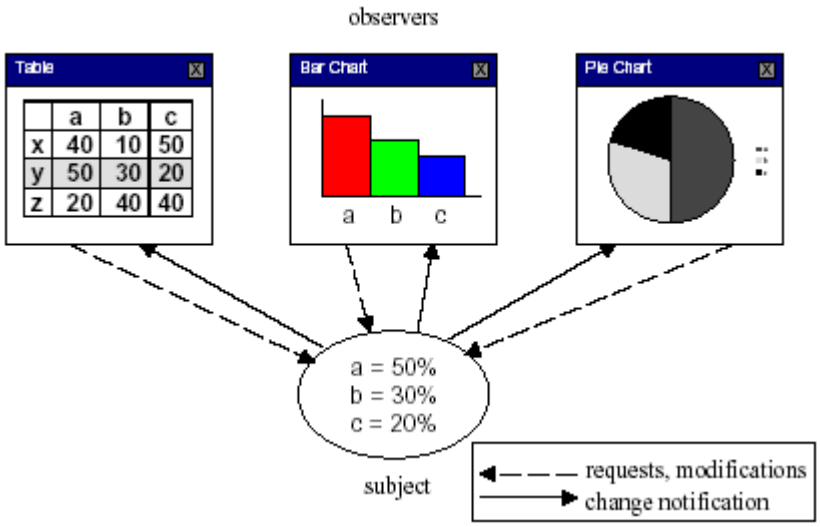
Also Known as:

publish-subscribe
Java Swing: *ActionListeners*
Model-View

Notes:

- The UML here shows only multiple observer sub-types, but a more general usage is where one could have varieties of subjects, derived from a base *Subject* type.
- The details of maintaining the subscriber (observer) lists is actually more involved, especially in the presence of threads, and usage in Swing is a prime example of this.

Example:



Notes:

- The entire Java Event model (since 1.1) is based on this pattern. Java has 11 different event listeners;
  Action Listeners = { *ActionListener, TextListener, AdjustmentListener*, …}
  Low-Level Listeners = { *ComponentListener, ContainerListener, MouseMotionListener, MouseLIstener, KeyListener*, …}

  There is then a whole system built from and on this, event adaptors, …

- Java actually provides an implementation of this in the *Observer/Observable* class library, found in *java.util.Observer* interface and *java.util.Observable* (base class).

- **Problem**, since Java has no MI, one cannot already be a derived class and also extend Observable!
  Solution: Use delegation, have your observable class contain an observable delegate.
  ⇨ *Fake-MI*

Issues:

- At one extreme, which we call the **push model**, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the **pull model**; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

- The **pull model** emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The **push model** might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

Discussion:

- One topic of design for the Observer pattern is the decoupling of the Observer and Subject. Certainly some decoupling is obtained, since the Subject only has a generic(?) list of Observers, and has no specific knowledge of their specific (sub-) type(s).

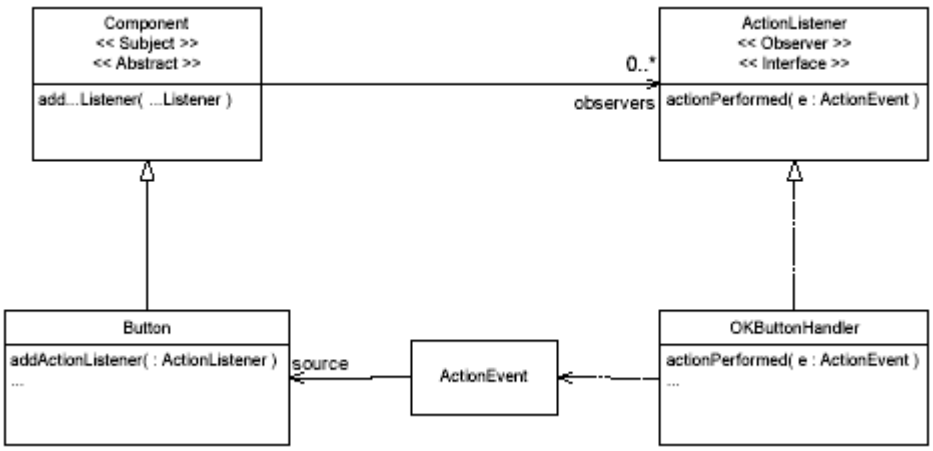
- However, because the two classes must communicate, they must have some shared knowledge for information interchange, unless a generic *notify()* event is sufficient (doubtful).

- Notice that the interface level declarations of the Observer Model the *update()* method has no parameters, thus it is not setup for a push model. This is because if it were, it would have to assume some particular type of information for the pushed argument(s), and thus no longer be a general interface, but specific to that application. Thus in this general model (which is what the Java

implementation is based on), one must have a pull model, where the derived *ConcreteObserver* & *ConcreteSubject* classes agree on a more specific and specialized data protocol for the *getState()* methods.

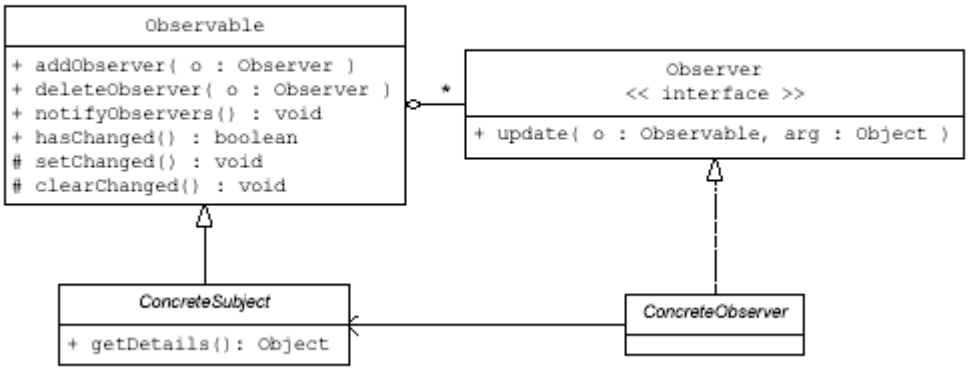
- One solution to this is to use a generic (type parameterized) version of the observer model…

Java Event Model



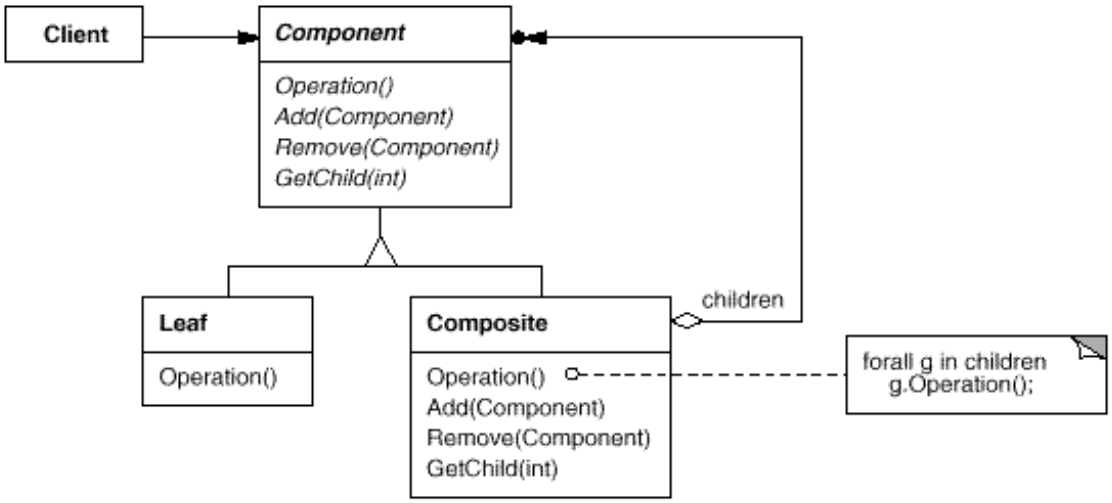
Java Observable Support

- The java.util package provides an Observable class and an *Observer* interface:

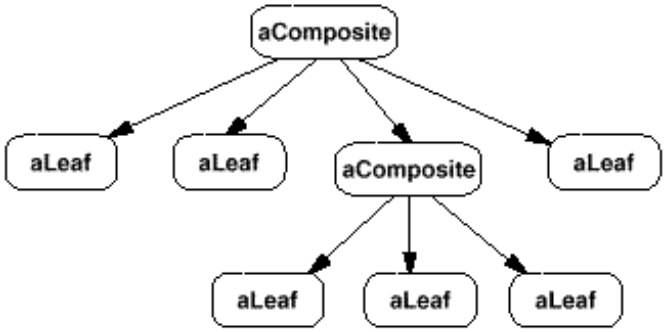


Composite Pattern

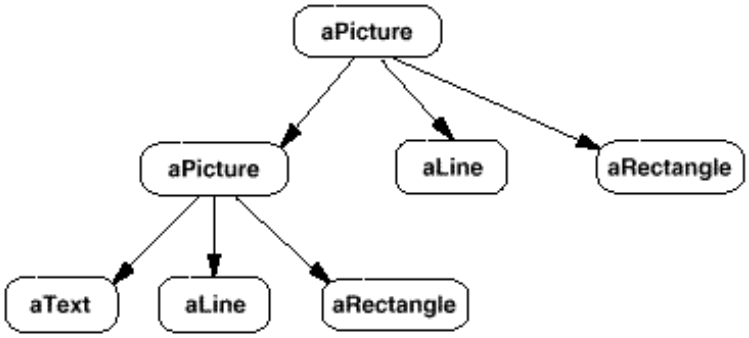
Structure:



A typical Composite object structure might look like this:



A typical composite object structure of recursively composed Graphic objects:



Summary:

Since the composite is based on the principle that the composite element broadcasts a message to all of it's components, which are all the same interface as itself, one could say:

$$\text{Composite} \approx \text{collection} + \text{polymorphic broadcast}$$

Notes:

- Note that the basic UML structure does not shoe the ability to have multiple sub-classes of leaf and/or composite nodes, or even multiple direct leaf types.
- There is a design choice between putting methods that differ between leaf/composite nodes in the base, or derived classes.

  The two choices are often called *transparent*; where one puts them in the base class, and *safety,* where they are in the derived classes. Transparent is more in the spirit of the pattern, where there is a single uniform interface to the all items in the composite.

  The *transparent* option means that a client cannot tell that the object is in fact a composite, or not. This approach is taken to the extreme where in fact the composite nature of an object is not revealed to the client in any methods. For example, no iterator would be provided; rather any result requiring internal iteration would do so transparently.

  The *safety* approach says that methods specific to leaf/composite nodes are provided only on those nodes, and similarly for composite nodes. This means that one would never have to deal at run-time with an *add()* message coming to a leaf node, for example. But, this means that at compile time one cannot send such a message to a wrong mode type, thus the two node types do not look the same – transparency is lost.

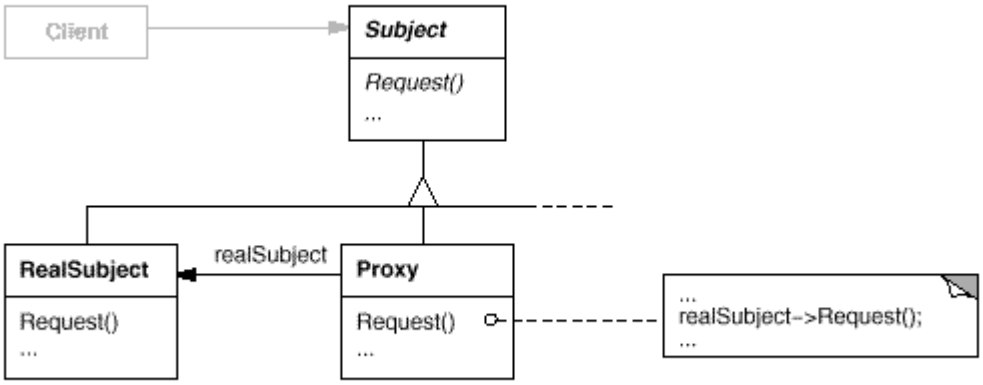
  [It seems that the GOF debated this issue in their original work, and long since resolved that the *add()* method should only be in the composite subclass.]
- Iteration is an interesting aspect of this decision, but in fact it is easy to proivde an iterator for either type of derived nodes. For a leaf node, one just provides a null iterator. For a composite, internally it would iterate over it's elements, recursively.

  Another approach to this is not to provide iteration, but to accept *visitors*. Then each visitor *accept()* method knows how to iterate properly within the composite strucutre.

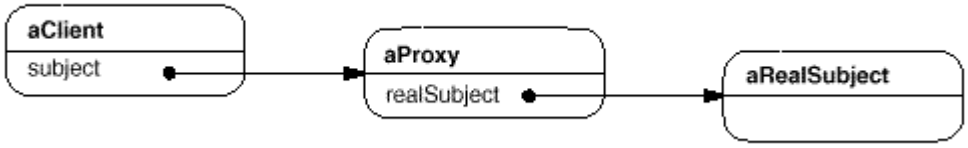
Proxy Pattern

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

Structure:



Example:



Notes:

- Dynamic proxies (Java 1.3) are a new and powerful addition!

  They address the problem of a standard wrapper addition desired for some class(es), but withoutthe need to manually generate new wrappered delegation versions of every method for each class.

  The feature uses the Java Reflection classes, and is all a bit mysterious and involved, but as always we can encapsulate all of this and make the capability very simle and easy to use!

  http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

- The GOF UML is wrong, because the proxy should have a link back to the interface, not to a specific concrete proxy object. This will allow one to have *layered* (or *recursive*) proxies.

- Interesting Example;

  Using Dynamic Proxies to Generate Event Listeners Dynamically , Mark Davidson
  http://java.sun.com/products/jfc/tsc/articles/generic-listener2/

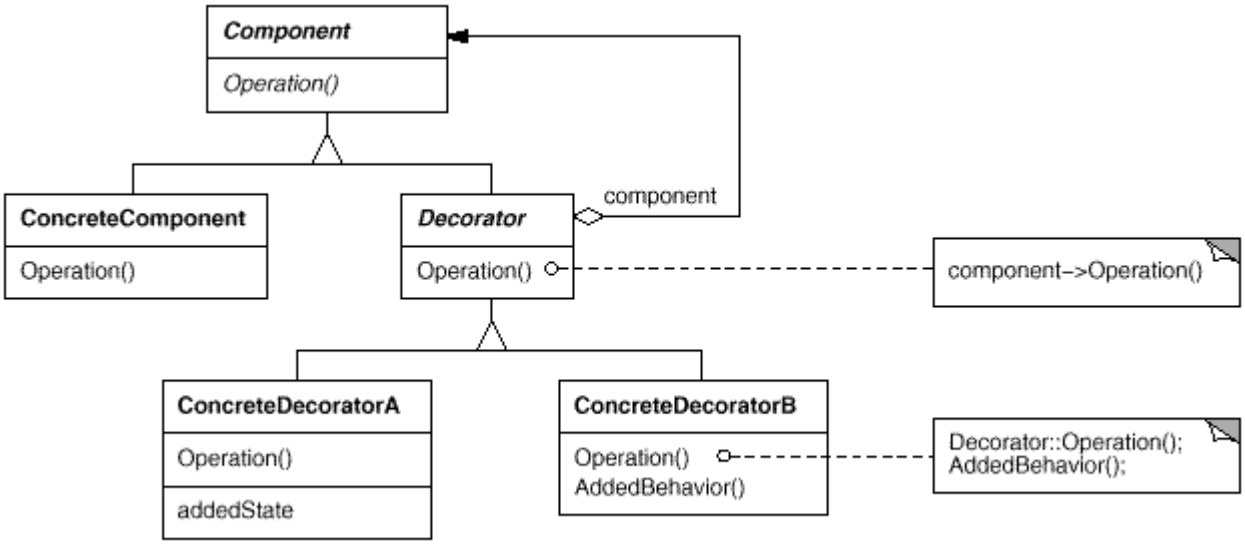
- Proxy patttern and Decorator are very similar; they differ only in the intent, to controll access to an object (perhaps with some value added), or to add features to an object – a fuzzy distinction at best. GOF says:

  > Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator.

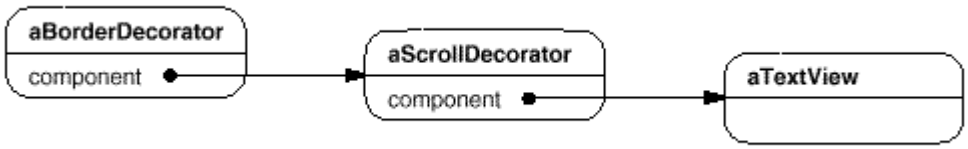
Decorator Pattern

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

Structure:



Example:



Notes:

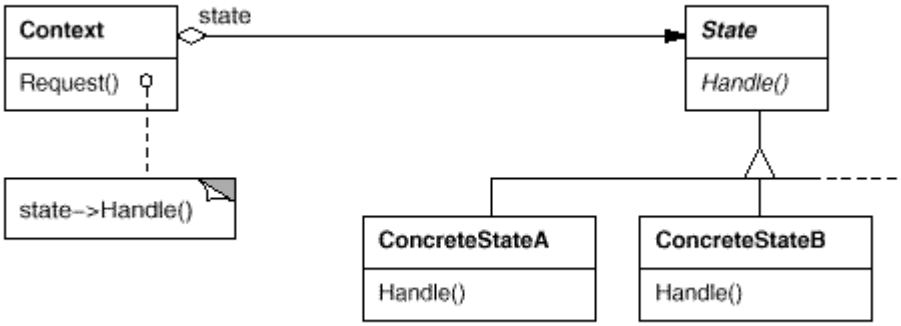
- Dynamic proxies can also be used for decorators.

  http://www.javaworld.com/javaworld/jw-01-2006/jw-0130-proxy.html

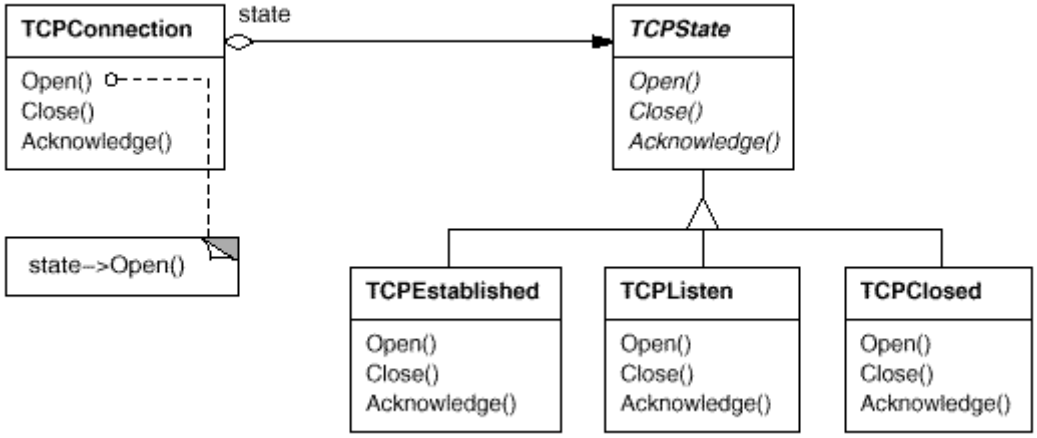
- The difference between a Decorator and Proxy is purely semantic….

State Pattern

Structure:



An example for a network protocol:



Notes:

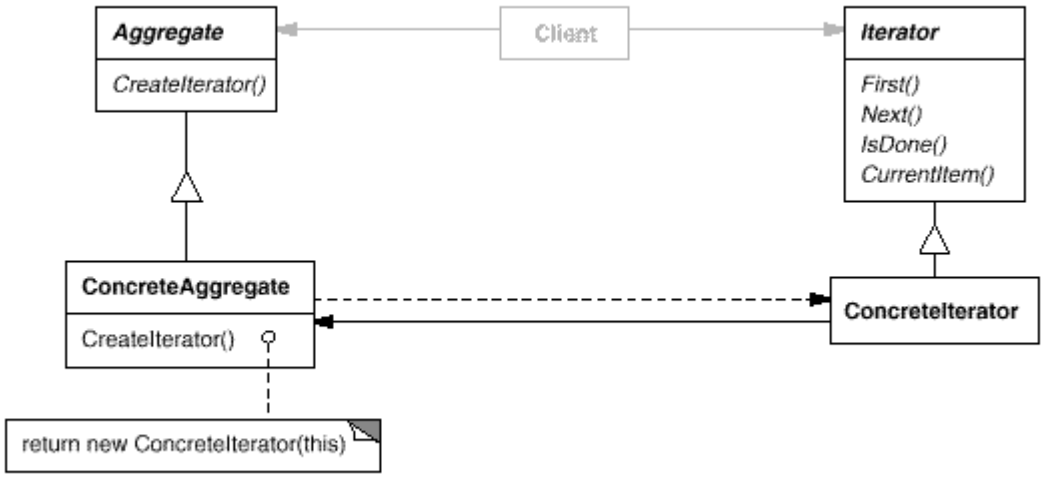
The implementation of State uses the Strategy Pattern.

See GOF for furtner notes.

Iterator Pattern

Intent
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Structure:



Notes:

- Having the iterator separate from the collection allows multiple simultaneous active iterations, each with its own (internal) state.
- Because the client interface is general to any iteration method and hides the actual iteration classes, one can have several different concrete iteration algorithms.
- A robust iterator assures that changes to the collection will not interfere with an ongoing iteration. Generally this means a separate copy is made, which is expensive.
- The fragility of client having to know what type of iterator to use for each type of collection is solved by the factory method pattern.

  In the Java Collections API this is by the use of a factory method: *Iterator<T> iterator()*
- *Null iterators.* A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal; that is, its *isDone()* operation always evaluates to true. [See: GOF]

Iterator Styles:

	Complete Traversal	Selective Traversal
External	Iterator it = c.iterator(); while (it.hasNext()) ProcFun(it.next())	Iterator it = c.iterator(CondFun); [Then use as normally!]
Internal	c.doAll(ProcFun)	c.doAll(CondFun, ProcFun)

Notes:

1) Of course, there are no (*bare*) functions in OO, so all functional arguments must be functors.

   The *CondFun* functor which always returns a boolean is called a *predicate*.

2) New Java iteration syntax:

```
Vector<T> v = new Vector<T> ();
for ( T item : v )
    item.doit();
```

3) Java *Iterable* interface

   - Required for use with new iteration notation!

```
public interface Iterable<E> {
  /**
   * Returns an iterator over the elements in this collection.
   */
  Iterator<E> iterator();
}
```

   -

4) Selective Iterators,

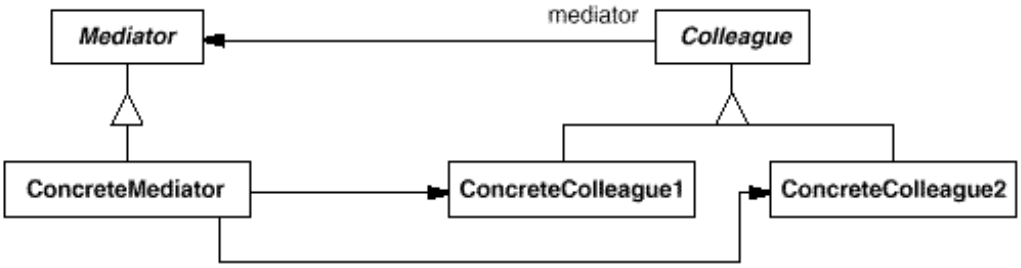
   For a selective iterator, one just passes the selection predicate to the constructor of the iterator.

   For an internal selective iterator, the predicate is passed in the invocation call, since that is where the actual Iterator is instantiated.

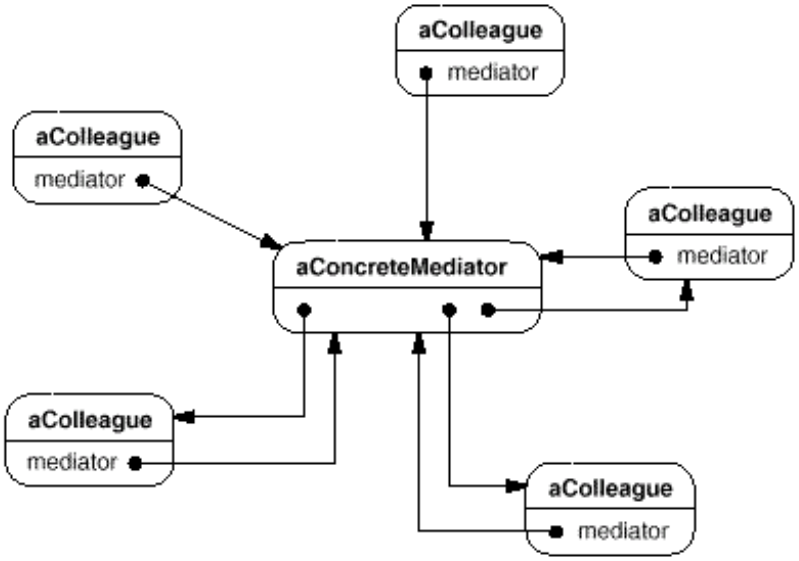
5) Selective Iterators ⇨ *Iterable*,

   It would be nice to use the new *for* syntax with selective and internal iterators. This means that one must not manually create a modified iterator, but must make a new version of the collection that is *Iterable* and will return the desired modified iterator.

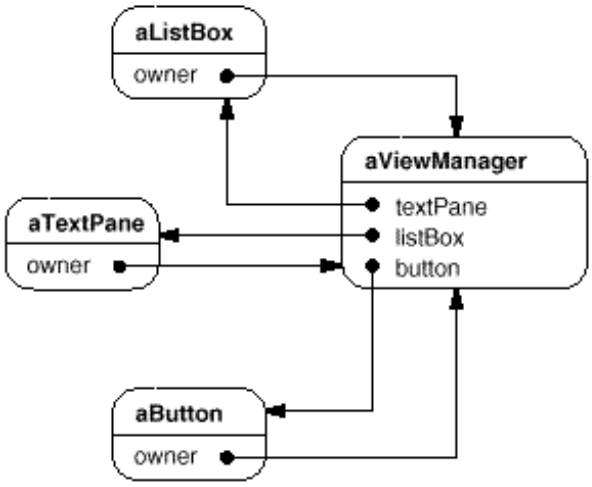
Mediator Pattern



A typical object structure might look like this:



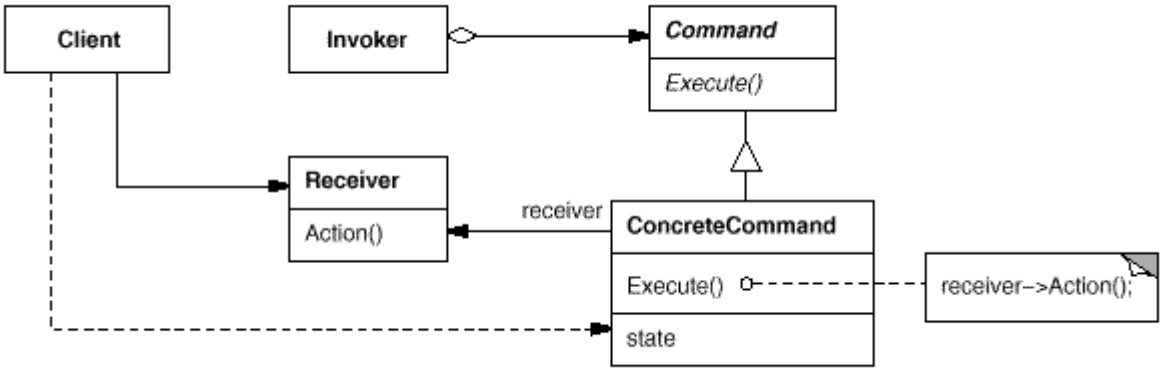
Sample Usage:



Notes:

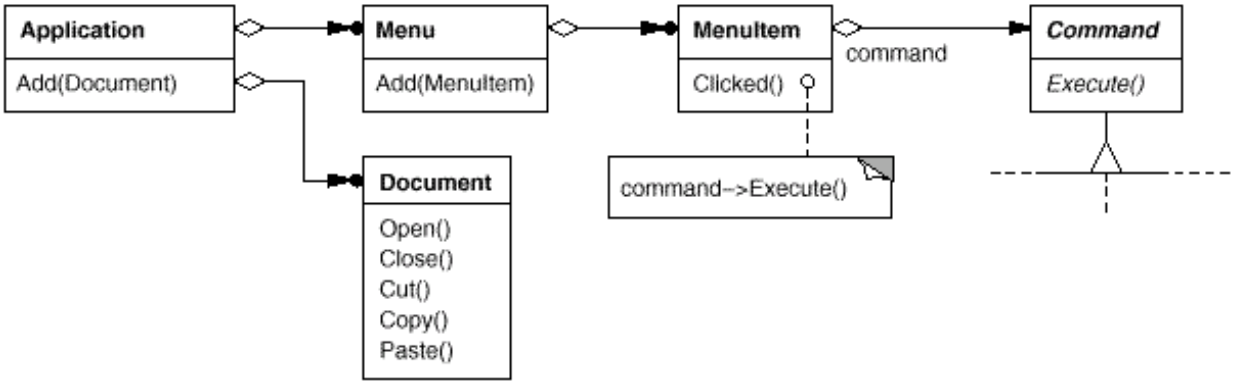
1) One must be cautious that the mediator does not become overly complex, since it centralizes all interactions.
2) The use of a mediator interface allows different *concreteMediator's*, and thus re-use of the colleagues.
3) Also see notes on Façade pattern.

Command Pattern

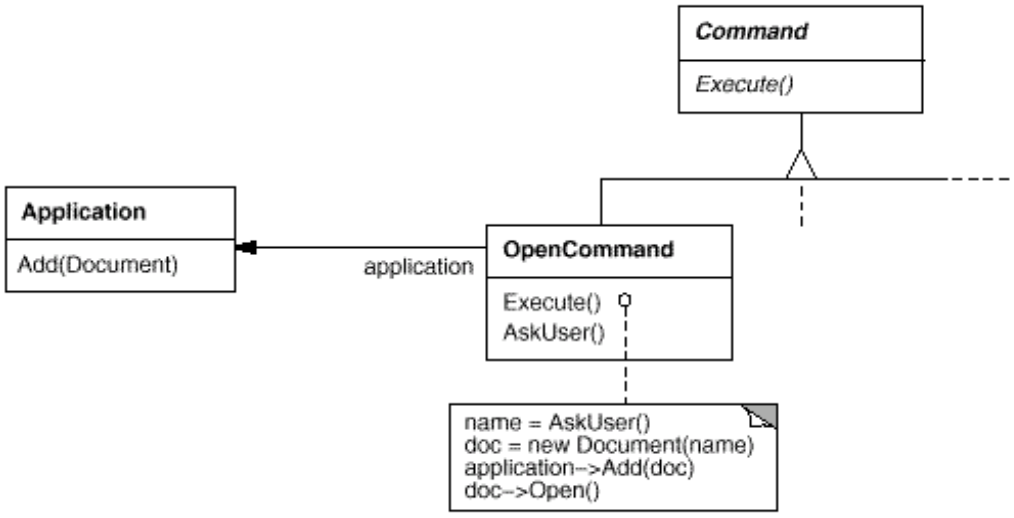


Sample usage for a menu of commands:

The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request. The receiver has the knowledge required to carry out the request.



The application configures each MenuItem with an instance of a concrete Command subclass. When the user selects a MenuItem, the MenuItem calls Execute on its command, and **Execute** carries out the operation. MenuItems don't know which subclass of Command they use. Command subclassesstore the receiver of the request and invoke one or more operations on the receiver.



(Also see Class notes)

Notes:

- The main goal of this pattern is the decoupling of the client, reciever, and invoker, all via the commands.
- The secondary (but important) result is that the encapsulation of commands as objects allows them to be managed in various ways; history, recall, logging, grouping (macros, transactions), undo/redo, etc.
- Note that some commands will have state, others be generic. For example, a *clear* command is generic, and one instance can thus be shared by all usages. On the other hand something like *push(x)* command has state (the value), and thus must have a unique instance for management.

  One can then either require that all commands are autonomous when sent to the manager, or can have the manager *clone* them before storing them. The former is generally more efficient, the later safer.

Factory Patterns - General

Intent

The general goal of all factory patterns is to provide an abstraction over the creation of objects. This is parallel to the standard capability of OO where we provide an abstraction over the usage of objects, through { inheritance + Polymorphism }. This sub-type abstraction then lets us decouple clients of an object (via interface messages) from the actual details of the object's functionality (implementation, via methods). The second goal of factory patterns is to allow more dynamic and flexible methods to create objects. Anytime one has a call to *new()* in their code, this is a static fixed specification of the exact type of the object to be created. We would like to make this more flexible, with the ability to decide at run-time what is to be created.

So how do we get the same benefits for object creation? This basic idea is sumarized in the first versions of this pattern, called *virtual constructors*, i.e. the goal to somehow have polymorphic constructors. Of course this is a paradox, since polymorphism is based on run-time self-dispatch of objects, and a constructor is pre-object.

In reality there is no magic way around this boot-strapping issue, but at least we can use the features of OO to localize and modularize object construction. This is the basis of all of the *factory patterns*.

This family of patterns includes:

- **Factory function**:
  Of course, in OO we don't have functions, so this would become a static factory method.

  The benefit of a factory function is that it *localizes* and *encapsulates* the information about what type of object to create.

  It might just produce some fixed desired target object type, or may use some context or take parameters to decide internally what sub-type to return; thus called a **parametric factory**. A parametric facotry decides at run-time what (sub-)type of object to create; and is thus more dynamic ≈ flexible.

- **Factory Method**:
  Here the creation is delegated to a sub-type, and then once given an appropriate sub-type of creator object, it can transparently create the desired target object types. These then generally create a static binding between the sub-types of the factory class, and the corresponding sub-typesof the target class.

  It is an interesting approach, because it puts the static bindings into the sub-class create() methods, and then uses polymorphic to dynamically invoke the approporiate method.

  Of course, one could then also have **parametric factory methods**. Thix is a tedt.

- **Abstract Factory**:
  One may need to have one level higher abstraction, the ability to have a factory which can create a group (family) of related objects, and then choose which group one wants them from. This *abstract factory* object would then create a specific *factory* object, which in turn would be able to create specific concrete obejcts form the desired family.

  And, **parametric abstract factories**.

- **Reflective Factory**:
  One can use the reflection properties of Java to dynamically create any object from the <String> name of the class.

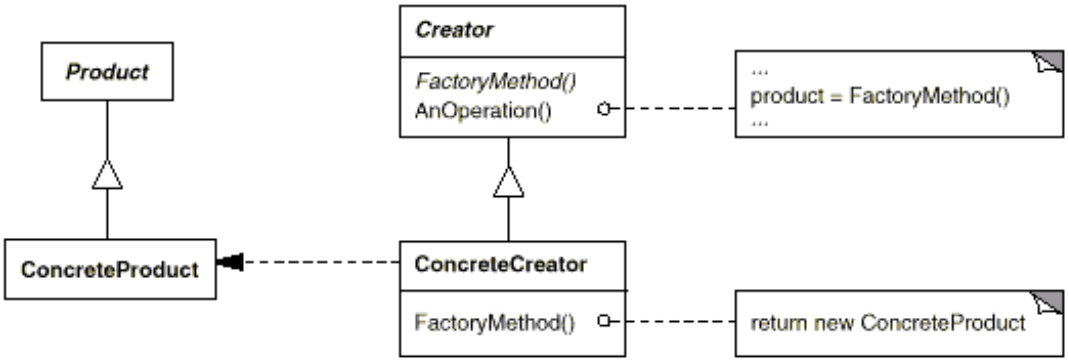
There are other related creational patters; See GOF.

Notice that all factories rely on the standard OO feature of sub-type abstraction, in that they retuen any of a set of derived sub-type objects under the base class type (=LSP).

Factory Method 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.

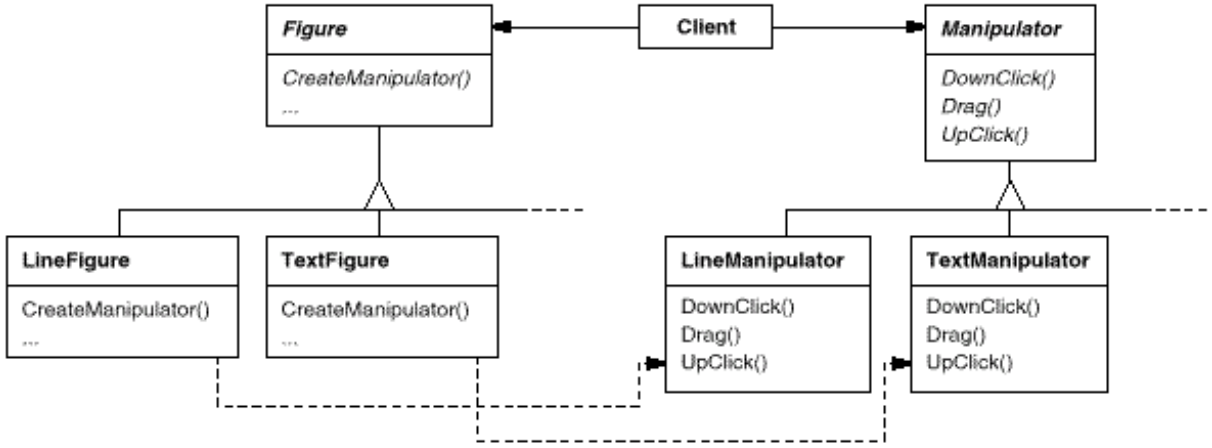


Notes:

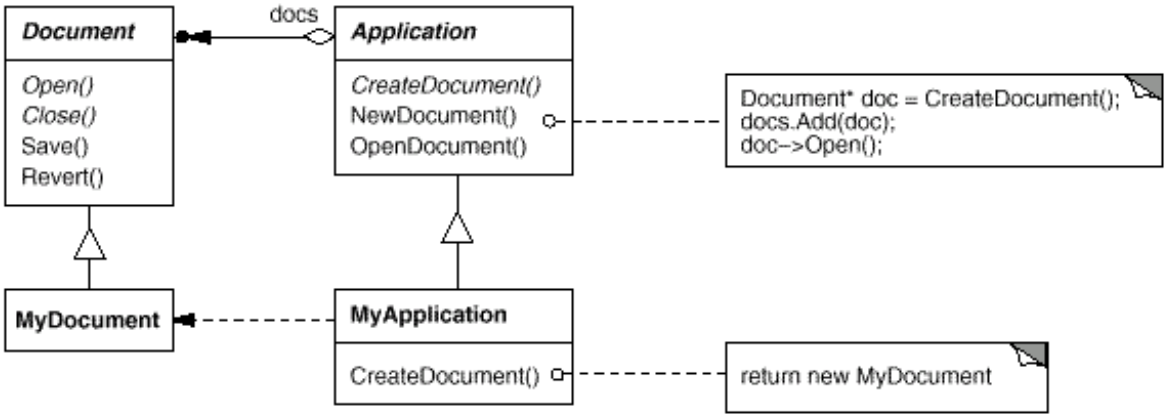
- Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.
- It allows sub-classes to override how parent classes instantiate objects, which would be suitable for the new extended class.
- *Parameterized factory methods*. Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface. (See GoF)

  Overriding a parameterized factory method lets you easily and selectively extend or change the products that a Creator produces.
- Often used in Frameworks, to allow a way for the framework to manufacture user plug-in objects. (See GoF example.)

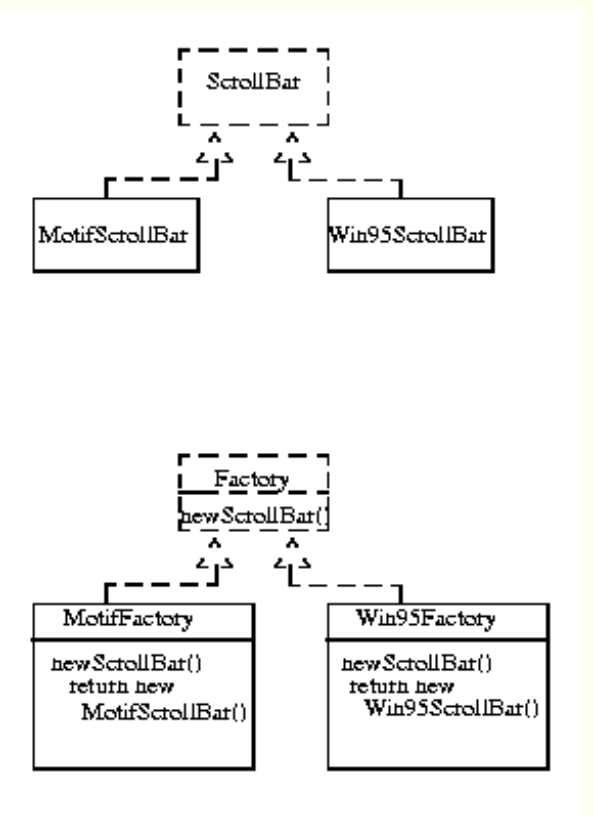
Example:



Example:



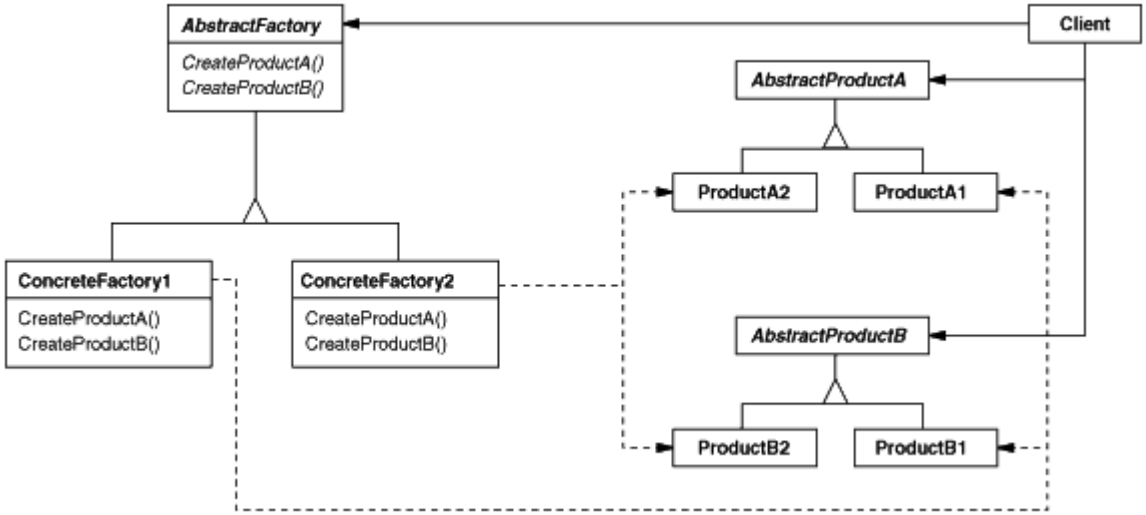
Multiple GUI factory:



Abstract Factory Pattern

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



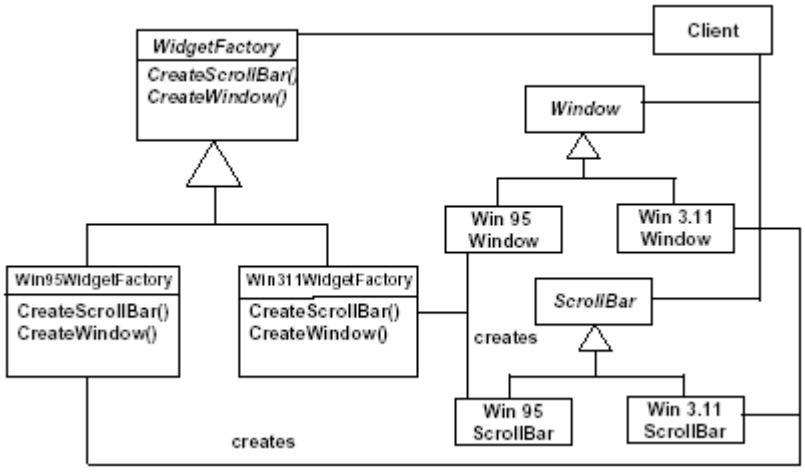
Notes:

- Normally a single instance of a *ConcreteFactory* class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different *concrete factory.*
- *AbstractFactory* defers creation of product objects to its *ConcreteFactory* subclass. This *ConcreteFactory* class must have a *CreateProduct (factory)* method for each desired *concrete product* in the family.
- AbstractFactory only declares an interface for creating products. It's up to *ConcreteProduct* subclasses to actually create them. The most common way to do this is to define a *factory method* for each product. A concrete factory will specify its products by overriding the factory method for each product family.
- I should note that in GOF the AF pattern is just to provide an interface for a family of factories, it does not specifically address the creation of these factories. I expand this to define an AF as the method to create such a hierarchy of *ConcreteFactories* as well.

Discussion:

Abstract Factory ≈ Factory Factory → Meta-factory (!)

Example:

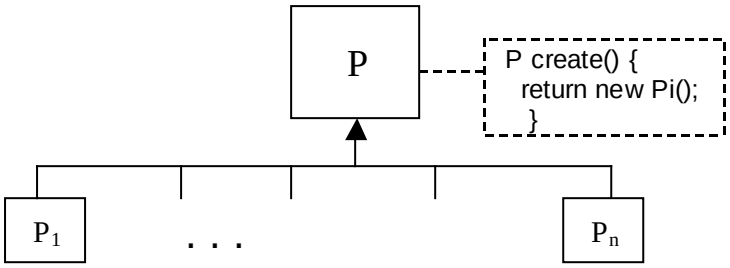

Client
WidgetFactory
CreateScrollBar()
CreateWindow()
Window
Win 95
Window
Win 3.11
Window
Win95WidgetFactory
CreateScrollBar()
CreateWindow()
Win311WidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
creates
Win 95
ScrollBar
Win 3.11
ScrollBar
creates

Notes on Factory Pattern:

- There are a variety of ideas and implementations for factories, which all can be classified by the placement, strucutre, and logic of the decision on what sub-types of a class to create.

Factory:

One class contains logic to manufacture the appropriate sub-types of objects. So the base class contains a *create()* method which knows which derived type to return to a client.



This could be a static method on the factory class.

Typically this *create()* method would have to have a parameter to determine which type of object to instantiate. Otherwise, how would it know. Some contextual or state basis for such a decision is possible, but uncommon. Because of this argument this simple factory becomes another pattern…

Parametic factory:

In this case the *create()* method has an argument to provide data to determine which sub-type to manufacture. This is then basically a:

$$\text{data} \rightarrow \text{(sub-)type}$$

interpreter. The data is dynamic, and the sub-types are hard-coded into the selection logic. One still has this maintenance problem, but it is all localized in one place.

As usual, one would like to replace this if/else static logic with polymorphism, but on what? The construction process is pre-object, and poly depends on object based method dispatch.

Factory method:

Relates two hierarchies of *creators* and associated *products*. Each *concrete creator* knows about the specific *concrete product* it creates. Typically the creator is someone who wants to use the product, e.g. it is a related application class, who (therefore) knows exactly what type of object it wants.

This is quite clever! The answer to "where is the logic for what to create", is that we already decided what component type to create, and since each component has a static known relationship to its desired (or related) object type, just let it provide this in its *create()* method; this is the *factory method* for which the pattern is named! So, the logic for what to create is gone, and assumed to be a decision already made when the associated component with the *create()* method was created.

One good example of this is the relationship between a hierarchy of collection classes, and their associated iterators. Once one has decided which specific collection (sub-)type to use, and one wants an iterator for it, it would not be a good thing to have to statically encode the type of iterator in the client code. Rather, one asks the collection object itself to create its associated iterator class;

```
Iterator it = c.getIterator()
```

But what about the actual collection object itself; GoF calls this component a *concreteCreator*, but who creates it!? Where is the logic that selects the sub-type of collection to use, and thus also the associated (encoded factory for) the iterator class. (Have we just hidden or moved the problem around?) Usually it is associated with a product, or associated processor, who knows what type of

product they want. In the end, someone, somewhere, must make some decisions about sub-types needed for an application. So use of a factory method does not preclude or subsume the problem of the need for an additional factory of some type for the original object.

Using the factory method:
Assuming one has solved this, then the client (application) uses a poly *create()* method on some component (C_i), which then polymorphically executes the appropriate C_i::*create()* method of that object to instantiate the proper concrete sub-class type object. Note that C_i::*create()* uses a static binding to it's related concrete product (*return new* $P_i()$), it is a fixed, known relationship.

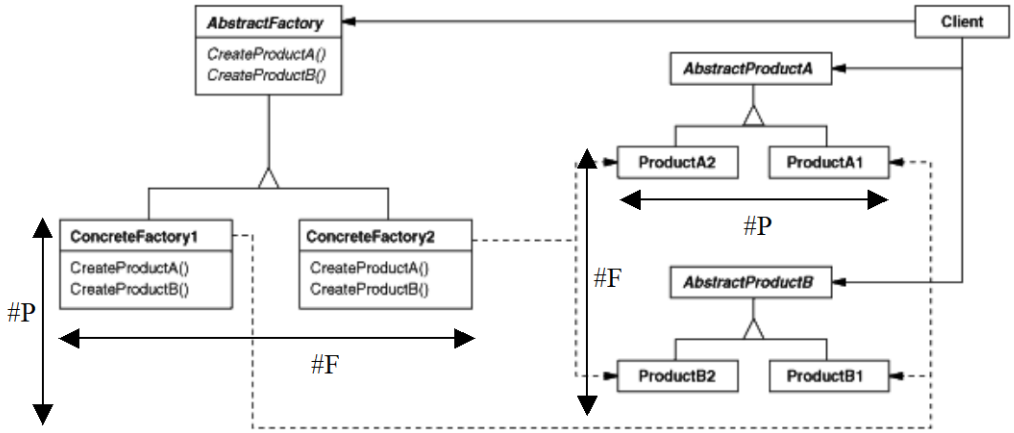
So, locating the poly element in this pattern easy; the poly method is named in the title – the *create()* (≈factory) method! This is the reason that this pattern is sometimes referred to as a *virtual constructor*, a cute oxymoron!

Example: The *iterator pattern* includes an *iterator factory* to create iterators associated with each iteration (collection) class. This is just a *factory method* pattern, applied to iterators.

Abstract factory:

This pattern has a somewhat different goal (as described above), which is to create factory objects. It still thus has all the regular factory issues as above, but the concrete factory objects it produces are used in a somewhat different manner.

Abstract Factories relate a hierarchy of factories, and a *series* of hierarchies of related abstract products (a *family* of types!), each with sub-type trees of same structure - since each product has to be available in each variety of production (each factory type). So this is a 2D hierarchy; one of products, and another hierarchy(s) of factories. If we call #P the number of products, and #F the number of product families;



So: { $\forall F_i \rightarrow$ known (static) how to *create*$P_j()$ ($\forall$j =A,B,C, …) }$_i$

Note that @P_i binding, the client knows that it has a factory ($F_i \in F$), and so just calls *F.createA()*, *.createB()*, etc. This is sort of a reverse *factory method*, in that there the client knew what it needed, and here the creator knows what to create.

But now, who creates the creator(!) and how? Note that each creator will know how to create a full family of products, so we don't need to know anything about sub-product decisions in the creator or factory. This information is in the C_i 's. So we decide which C_i ($\forall$ Products) by deciding which family to adapt/use (i.e. which factory to create). This is a bigger, once for all products, global decision. Once made there are not individual "which C_i?" decisions to make.

The creation of *concrete factories* by the *abstract factory* could be done in a variety of ways, and could involve poly (as above), since AbsFactory $\in$ Factory!

But the important poly here is in the usage of the actual *concrete factory* objects. Once the client has an actual (concrete) factory, it makes *createXxx()* calls to it, which are poly bound to the appropriate

(actual concrete) factory (sub-)class used. That class has static binding to the actual concrete product classes to be instantiated. So, we need poly on the *creates()*, because we don't know the actual sub-type of factory we have, which family line of types we are creating.

So, the factories created are not the typical factory, or factory method objects. Rather they are a pre-packaged set of *create()* methods for all products in a particular type or product line.

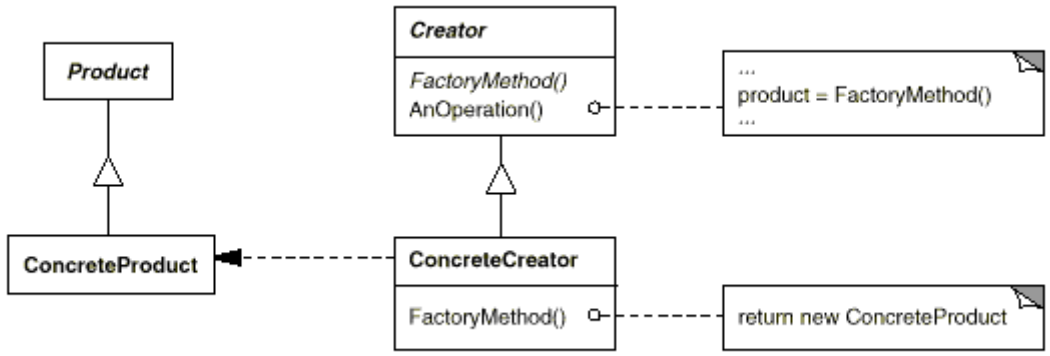
Factory usage:

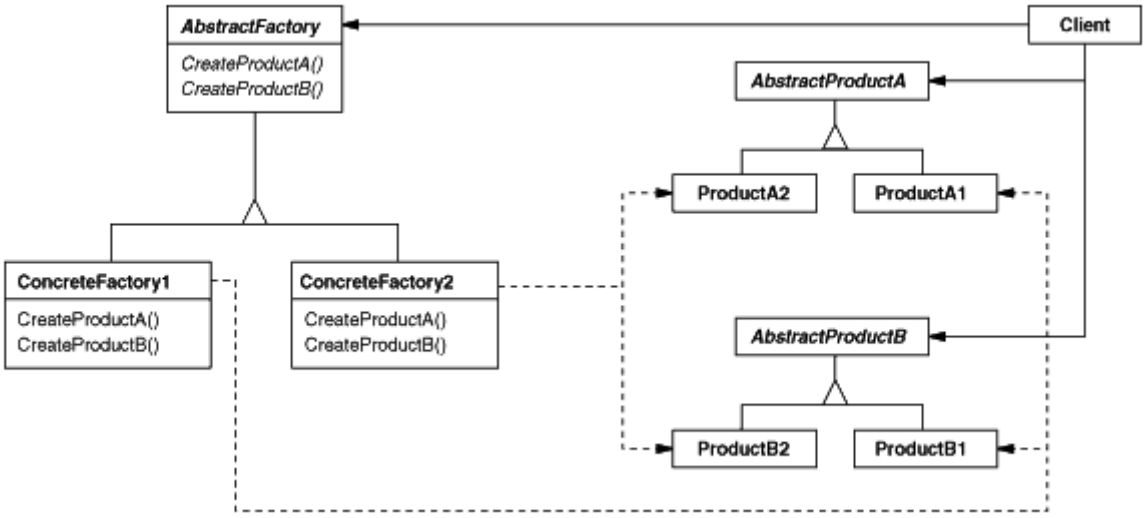
It may be that a class wanting to build it's own objects can be parameterized (in its constructor) with a factory object, and then use it internally for it's creations. This would allow a general class to be specialized by its attachment to a specific factory.

For example, one could think of a Store class, which can be instantiated to operate on various types of products, e.g.

```
BookStore  s1  = new Store(new Factory("Books"));
OfficeStore s2  = new Store( new Factory("Products"));
or…
BookStore  s1  = new Store( absFactory.create("Books") );
OfficeStore s2  = new Store( absFactory.create("Products") );
```

Notes:
Note that the UML for factory method & abstract factory are the same, with some simple transformations and renaming.





Both have an Abstract Creator (AC) interface, with several Concrete Factory (CF_j) children {j=1,m}.

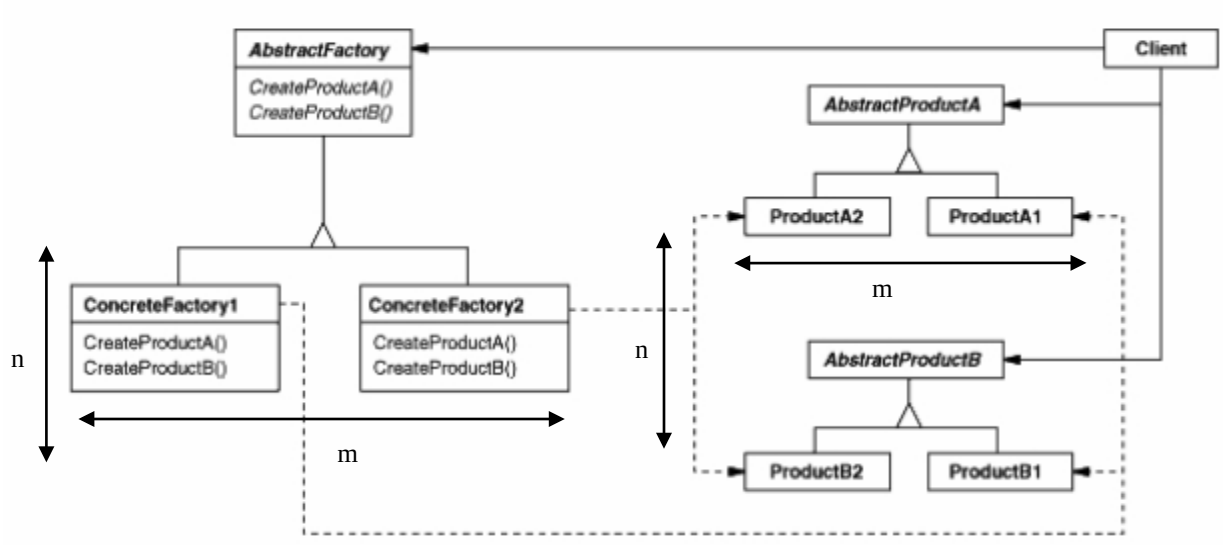
They also have a product hierarchy(s) (AP_i), and Concrete Products ($CP_{i\ j}$), {i=1,n}.

m is the number of different CFs, the number of different ways to create things, and n is the number of CPs, the number of different things to create.

Each CF_j has methods to create $CP_{i\ j}$ $\forall$i. The difference is (just) that in Factory Method, there is a single product family, and thus ||i||=1, whereas in an Abstract Factory, ||i||=n, and there are thus n families of AP_i and thus also n Factory Methods in each CP_j class.

So the difference is that in FM the creation hierarchy is oriented towards some other general goals (e.g. collections), but has one method for creation, the Factory method. Whereas in an AF design, the creation hierarchy is dedicated to creation, and so has multiple FM's @CF_j sub-class. SO, as GOF notes, "AF is implemented with FMs".

Consider an AF w/ n=1, and you have a FM design! ☺

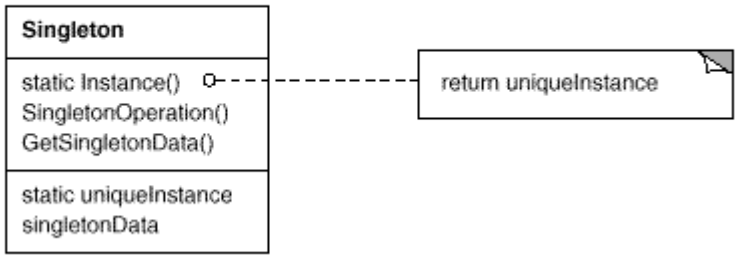


m = #F (Number of Families of products)
n = #P (number of products in each family)

Singleton Pattern

Intent

Ensure a class only has one instance, and provide a global point of access to it.



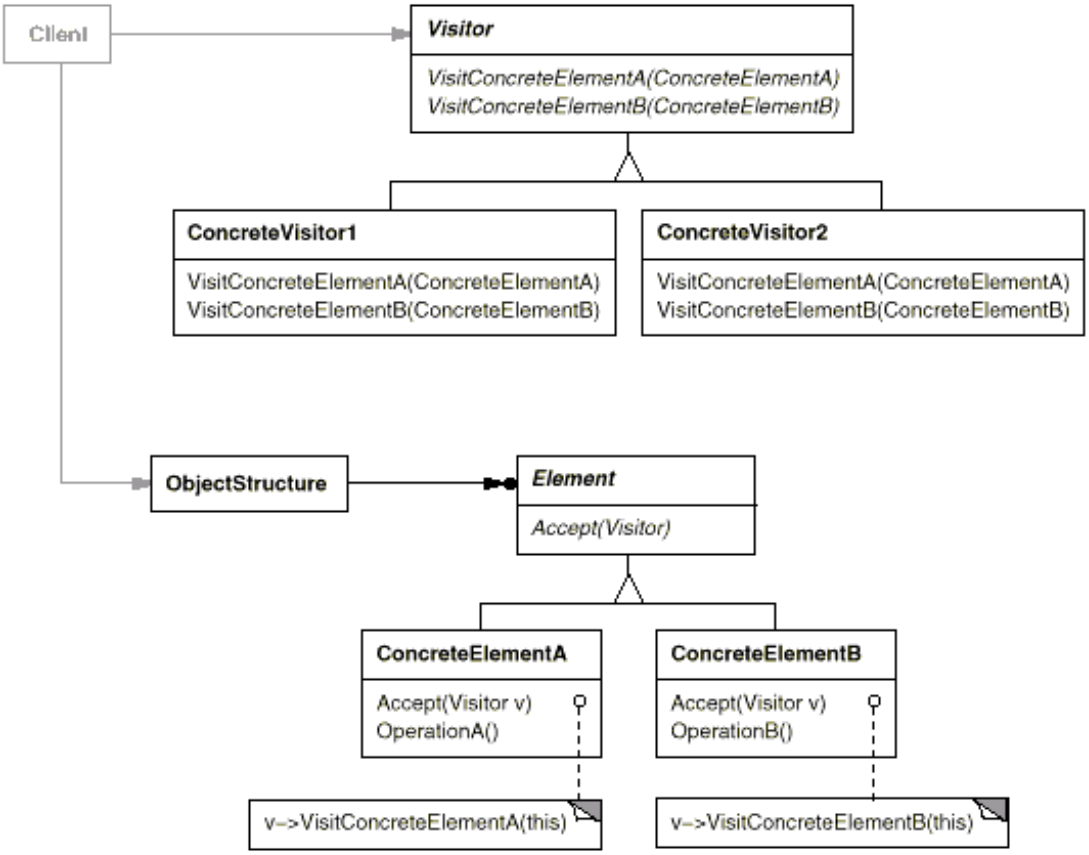
Notes:

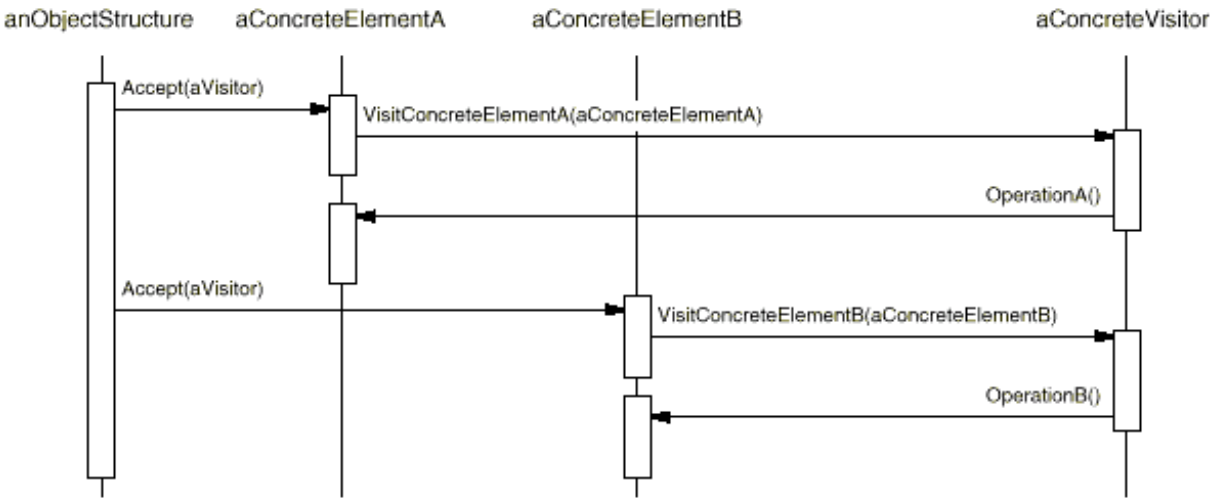
- Singleton ⇨ multi-instance; and then object pooling.
- This is really a simple (special) form of a factory.
- Singleton is one of the simplest, and also deceptively simple, patterns.
- A simple and standard implementation is just a single static datum. But there are problems with this;
  - one may not have enough information at compile (load) time to create the desired shared object(s).
- Trouble and complexity enter when one tries to add optimizations, and lazy instantiation.
- Typically the class will have a protected constructor, thus preventing user (direct) instantiations of objects.

Visitor Pattern

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.





Implementation
Each object structure will have an associated Visitor class. This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure. Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement, allowing the Visitor to access the interface of the ConcreteElement directly. ConcreteVisitor classes override each Visit operation to implement visitor-specific behavior for the corresponding ConcreteElement class.

Each visited class must then have a method of this general form:

```
public void accept(Visitor v) {
    v.visit(this);
    }
```

Implementation Issues:

1. Double dispatch.
Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called *double-dispatch*. Languages like C++ and Smalltalk support *single-dispatch*.

In single-dispatch languages, two criteria determine which operation will fulfill a request: the name of the request and the type of receiver.

"Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of two receivers. Accept is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on each class of element

This is the key to the Visitor pattern: The operation that gets executed depends on both the type of Visitor and the type of Element it visits. Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use *accept()* to do the binding at run-time. Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

2. Who is responsible for traversing the object structure?

A visitor must visit each element of the object structure. The question is, how does it get there? We can put responsibility for traversal in any of three places: in the object structure, in the visitor, or in a separate iterator object.

3. Accumulating state. Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the visitor by the operations that perform the traversal, or they might appear as global variables (= poor design → ☹).

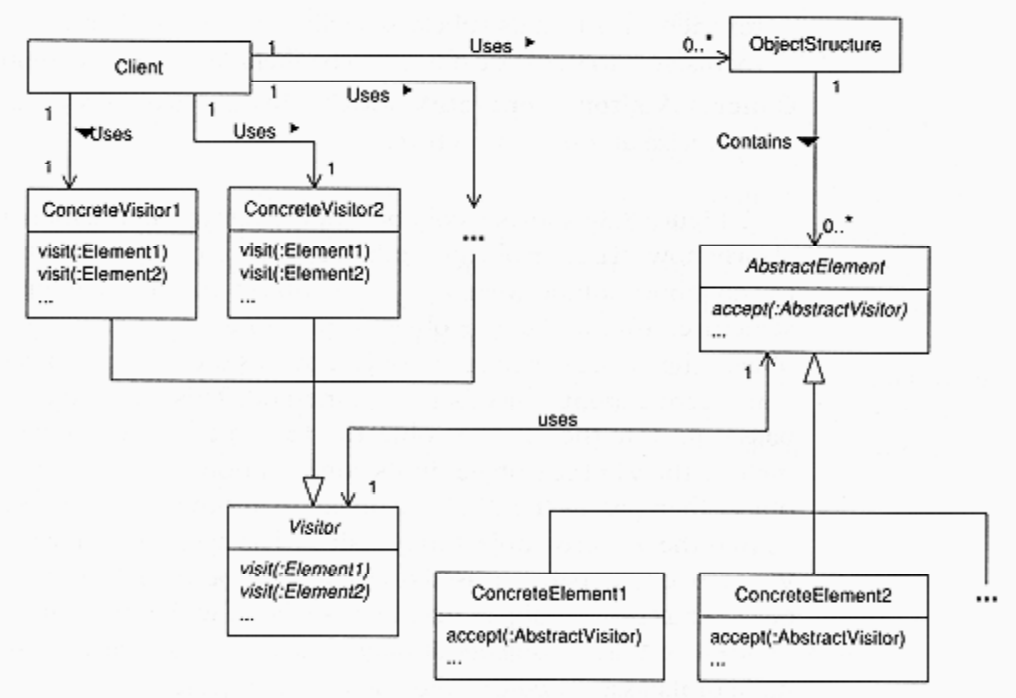
4. Visiting across class hierarchies. A visitor can visit objects that don't have a common parent class. You can add any type of object to a Visitor interface. For example,

```
class Visitor {
public:
   // ...
   void VisitMyType(MyType*);
   void VisitYourType(YourType*);
};
```

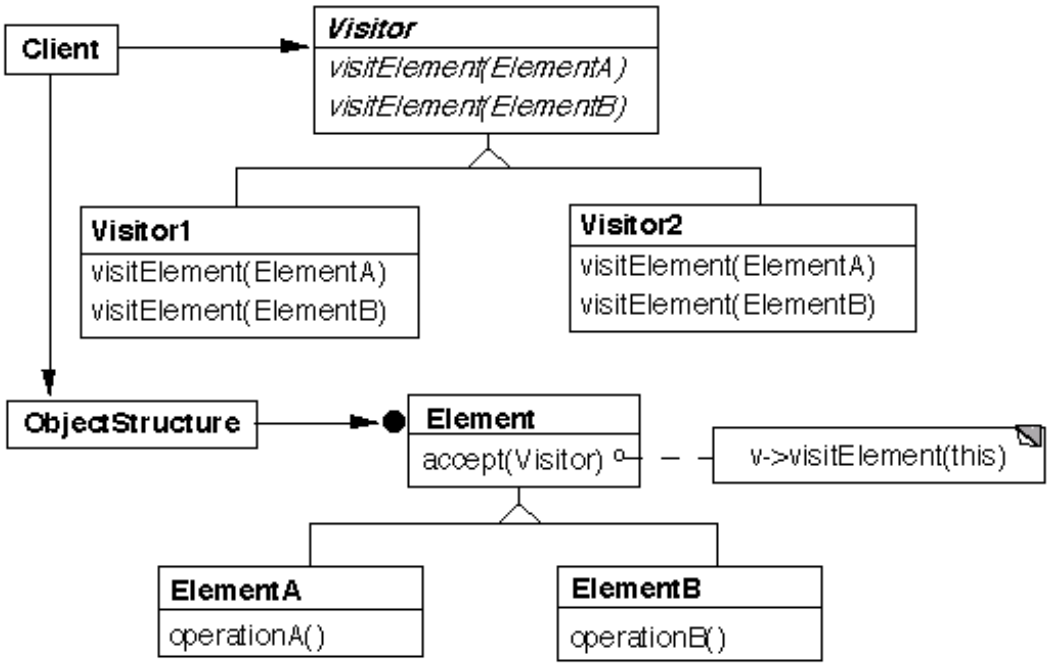
MyType and YourType do not have to be related through inheritance at all.

5. Method naming: We could use function overloading to give these operations the same simple name, like Visit, since the operations are already differentiated by the parameter they're passed. There are pros and cons to such overloading. On the one hand, it reinforces the fact that each operation involves the same analysis, albeit on a different argument. On the other hand, that might make what's going on at the call site less obvious to someone reading the code. It really boils down to whether you believe function overloading is good or not.

Example:

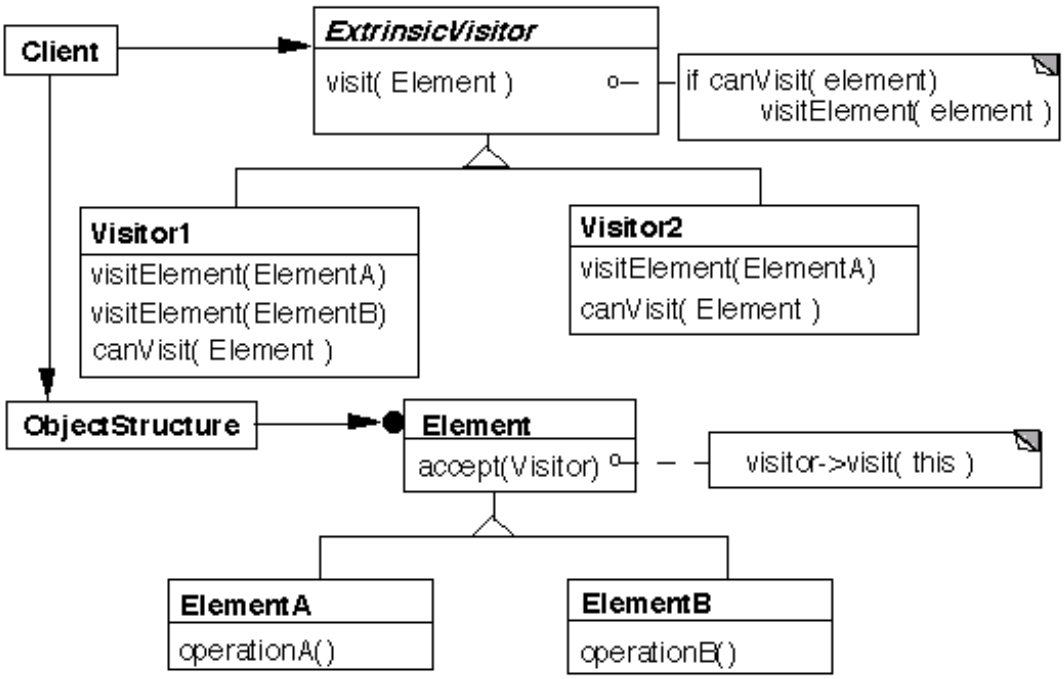

Client
Uses
ObjectStructure
Uses
Uses
Contains
ConcreteVisitor1
visit(:Element1)
visit(:Element2)
...
ConcreteVisitor2
visit(:Element1)
visit(:Element2)
...
AbstractElement
accept(:AbstractVisitor)
...
uses
Visitor
visit(:Element1)
visit(:Element2)
...
ConcreteElement1
accept(:AbstractVisitor)
...
ConcreteElement2
accept(:AbstractVisitor)
...

Visitor Variations: Visitor with Overloading



Overloading allows the accept method to be implemented once in the Element class.

Visitor Variations: Extrinsic Visitor



- The Visitor takes the responsibility of making sure a visitor can visit a particular concrete element
- Each Visitor knows which types of concrete classes it can visit
- The canVisit method of each visitor returns true is that visitor can visit that concrete type
- New visitors can be created without changing the Element classes
- New Element classes can be added without having to change old visitors

The above is slightly modified from the original ExtrinsicVisitor pattern.

In the original:

- Element does not have an accept method
- The visit method of ExtrinsicVisitor does not call canVisit
- ObjectStructure calls canVisit to determine if an element can be visited, then calls visit
- The visitElement methods are not over loaded

Ref: http://www.eli.sdsu.edu/courses/spring98/cs635/notes/visitor/visitor.html

Visitor Discussion & Notes:

The *Visitor Pattern* relies on both *overloading* and *overriding* of methods.

a) What are the conventional names and signatures of the overloaded and overridden methods in the Visitor Pattern

b) Describe specifically where each of these (overloading, overriding) is used, and what value is provided by each.

Solution:

1) *V::visit(D_i d)* = Overloaded (@ V_j) & Overridden ($\forall D_i \in @V_j$)

2) *D_i ::accept(Visitor v)* = Overridden

Discussion: Recall that the visitor pattern has two parallel hierarchies, one of data objects, $\{D_i \subset D\}$, where D is the (abstract) base class type of the hierarchy of concrete data object types, and a related set of visitors $(V_j \subset V\}$, where V is the abstract base of the tree of derived concrete visitor classes.

The *Visitor::visit(D_i d)* methods are overloaded, one for each sub-type D_i , in each concrete visitor class, and overridden from *V::visit()*, one (set) for each concrete visitor type V_j. Note the 2D element here, these *visit()* methods are both overloaded @D_i and overridden @V_j – this is the *double-polymorphism* of this pattern.

The *D_i ::accept(Visitor v)* methods override the base class *D::accept()* method, and when called by the traversal iteration of the data objects, the appropriate *accept()* method is invoked by polymorphism on the actual (sub-)type of the data object used. Note that the Iterator (traversal client) uses *D.accept()* calls, because it does not know the actual dynamic types of the objects referenced in the collection elements, and it is by polymorphism via the actual dynamic types of the referenced objects that the correct method is called.

In this dynamically bound method, each item then invokes the *visit()* method on themselves: *v.visit(this)*, where $\{this \in D_i\}$, and the appropriate[overloaded] *V_j.visit(D_i)* message is chosen, and further the actual method used (dynamically bound to this message) is based on the actual type of the concrete visitor v $\{v \in V_j \subset V\}$ of the visitor hierarchy (V); thus the second level of the *double-dispatch*.

```
Di::accept( visitor v) {
        v.visit( this );    // (this ∈ Di ), (v ∈ V)
        }
```

To summarize:

Client: *Iterate* { $\forall$ *D_i* $\in$ Collection(D) }

D::accept(visitor v)$\big|_{msg}$

(poly) $\rightarrow$ *D_i*::accept(visitor v) $\big|_{method}$

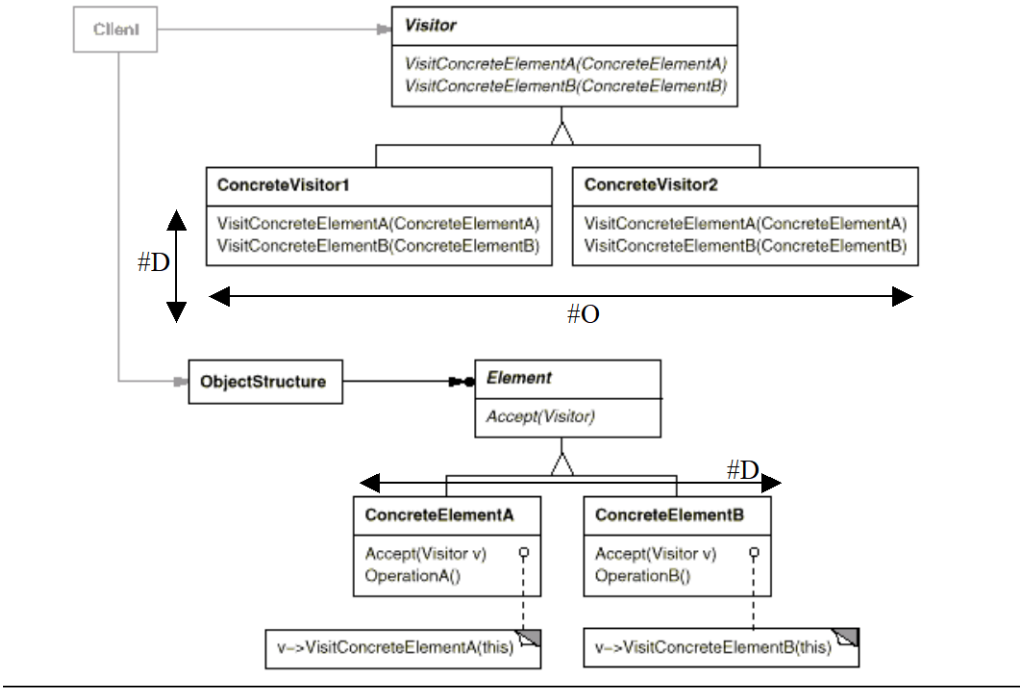
(calls) $\rightarrow$ v.visit(D_i) $\big|_{msg}$

(poly) $\rightarrow$ V_j::visit(item D_i) $\big|_{method}$

$\approx \rightarrow V_{ij}$ (Note that there are thus (i*j) different actual *visit()* methods!)

Discussion:

Because of the relationship of the two type hierarchies realted by the visitor pattern, there is a constraint on the structure of the hierarchies. Call the number of operations #O, and the number of data sub-types #D;



Discussion:

- Question: Could one have more levels of polymorphism, for example triple-polymorphism?

  Yes; pass arguments, and at each successive level, convert parameters to object.method calls. The key point here is that *f(x.y)* is not poly, but *x.f(y)* is.

- Note that the implementation of the accept methods are all the same, sort-of. That is the code looks identical, and is syntactically identical, but the body of each is actually a different call. This is because the call *v.visit(this)*, the argument this is of a different type in each receiving object of type D_i (this $\in$ D_i). So the call to visit is statically bound to the appropriate specific concrete method on the visit type class. Then, this is poly bound to the actual type of the invoking v_i used in this accept.

  Thus these *accept()* methods would be a great candidate for a generic method, parameterized over the type(s) of D_i.

- Note that the calls to visit(D_i) are all overloaded in each Vj. This is emphasized in the GoF formulation of the pattern, where each name is in fact different, and named by the type of its D_i argument. Of course, this is a redundant naming, since by overloading we can share a common name (visit) and select based on the argument type. This method is called an *overloaded visitor* pattern, but is actually the more common approach.

  Any time one has two parts of a definition that are dependent, and one can be mechanically determined by the other, it is simplest to let the compiler do this. It also eliminates errors, where a programmer might accidentally write:

V_j::visitTypeD_i (D_k)

- See the DoF implementation notes for a discussion of ways to have the visitor traverse the collection of data elements. Three methods are discussed, the data elements control the traversal, the visitor, or the collection (via internal, or external iterators).

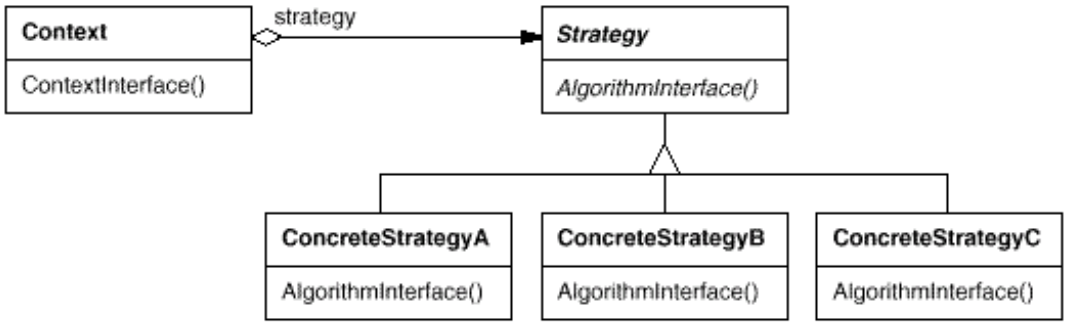
  One particularly interesting observation is that:

  > "The main difference is that an internal iterator will not cause double-dispatching—it will call an operation on the *visitor* with an *element* as an argument as opposed to calling an operation on the *element* with the *visitor* as an argument."

Strategy Pattern

Intent:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Useful when you want to do the same thing, but perhaps in several different ways. Strategy lets the algorithm vary independently from clients that use it.



Use the Strategy pattern when:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

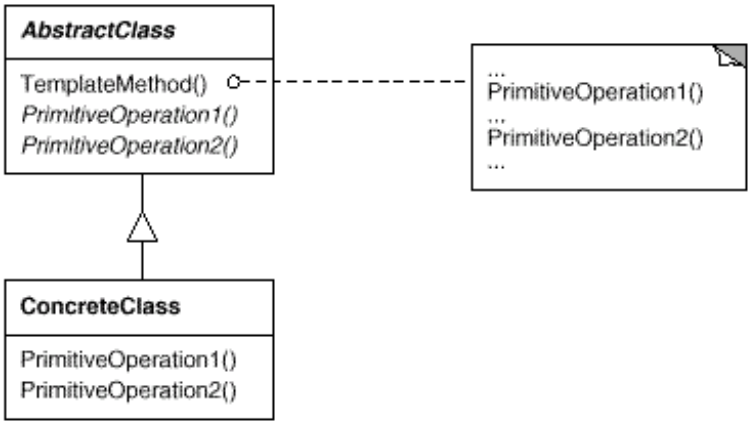
Notes:

- Very similar in structure to *command* pattern, difference is in intent.
- The *strategy* class can provide a context, which controls the selectuion and use of the algorithm.
- How is *strategy* different/similar to the *state* pattern?
  - Strategy switches an implementation algorithm for a fixed result, and the user result is the same. User or context object provides strategy switching logic (could be done by client).
  - State switches an entire object sub-type appearance, and all associated behavior/methods. The object behaves differently. State also provides the state switching logic internally.

Template Method Pattern

Intent:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



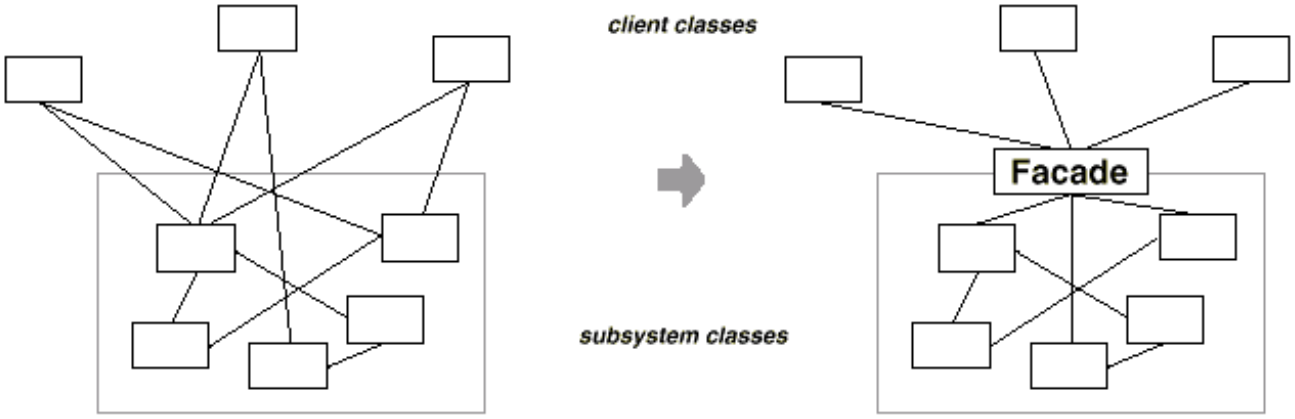
Method:

- Write abstract class that defines basic logic.
- Provide concrete methods which call abstract methods which (will) provide the missing logic
- Derive concrete classes that provide the missing logic in subclass methods that override the (base class) abstract methods.

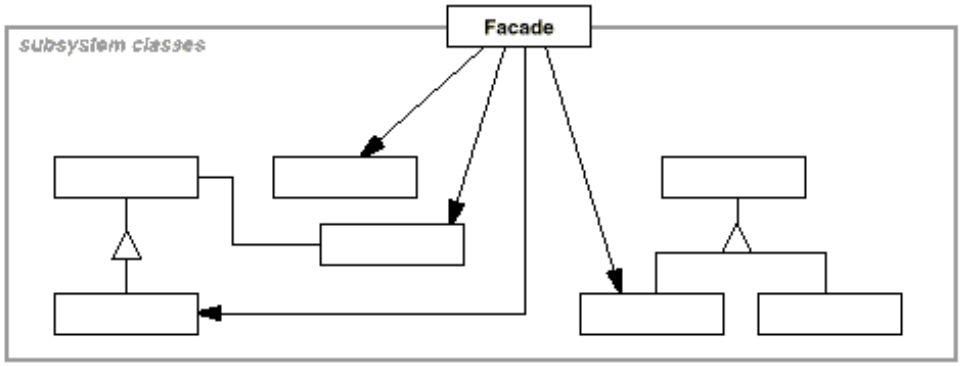
Façade Pattern

Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.



Structure:



- GOF: "Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it."

Bridge Pattern

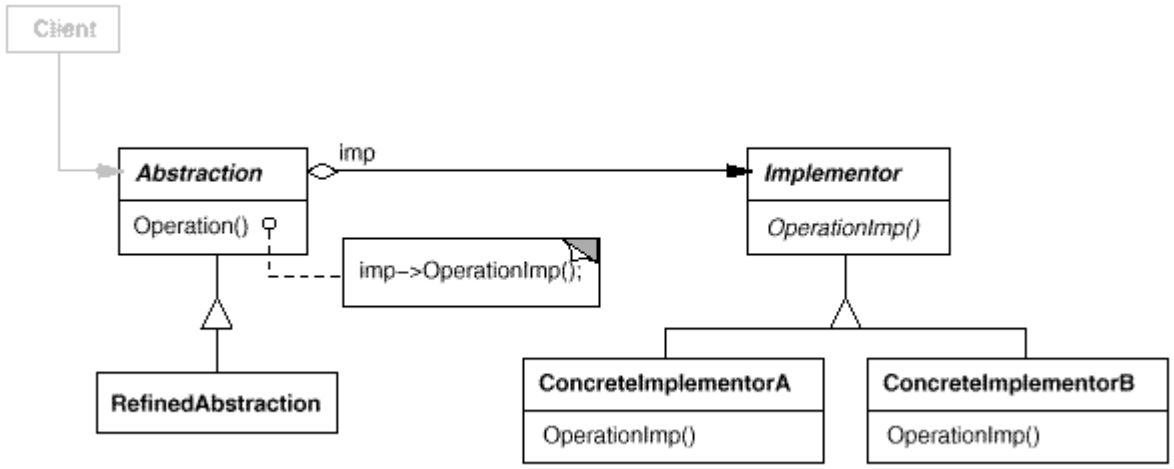
Intent:

Decouple an abstraction from its implementation so that the two can vary independently.

Motivation:

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies. There is one class hierarchy for interfaces and a separate hierarchy for specific implementations.



Usage:

- Abstraction forwards client requests to its Implementor object.

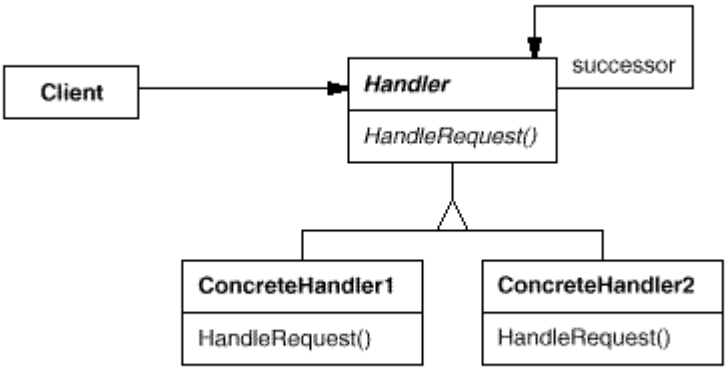
Notes:

- Avoids a permanent binding between an abstraction and its implementation. This allows the implementation to be selected or switched at run-time.
- both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- *Improved extensibility*. You can extend the Abstraction and Implementor hierarchies independently.
- *Shared implementaions*: For example, COW, or COA shared string representations.
- Can hide delegation of operations to implementation by providing utulity in base class which actually does the delegation, then in derived abstraction classes, they just use this inherited function, which hides the actual delegation.
- Note a good example of this is the Java Swing, Look-and-Feel implementation options.
- Note that this pattern also has two hierarchies, but unlike *visitor pattern* is not double dispatch. The client call uses one level of poly to get the appropriate *abstraction*. This then (internally) uses the poly *implementation* methods to accomplish its actions, but the user does not directly (double-poly) actually call any *impl* methods.

Chain of Responsibility

Intent
Decouple the sender of a request and its receiver by creating a pool of potential receivers, and giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Structure



Notes:

- One can dynamically manage the pool of command recipients.
- Because the chain is processed sequentially, the processors should be ordered from most specific to most general. How this is managed without exposing the internal structure of the chain is an issue.
- Allows a collection of specialized (decoupled) processors to act in a unified manner.
- Note: The processors are *mutually-exclusive*, only one processes a request. This is different than a pipeline or sequential processors where each one would add some value.

General Discussion and Notes

Question:

We saw several patterns with multiple derivation hierarchies, but each was somewhat different.

a) Describe the patterns with this quality, and what each hierarchy was used for in that pattern.
b) Describe how each was similar, and different in its use of polymorphism. You need to describe the basic problem solved by polymorphism in each pattern.

Solution:

a) Visitor: Data and Operations hierarchies, with poly over both. Client uses poly to choose a sub-type of data for elements in a typed collection, and to choose an operation to apply, from a derived type fo the operation family. Visitor pattern then used double-poly to actually apply the correct operation to each concrete data element in the collection.

b) Bridge: Hierarchy of *implementations*, and of *abstractions* (components). Poly is used for each, (but not double-poly like *visitor*). Implementation is selected once by client, and then same implementation is used for all components. Switching implementations is rather like switching strategy, one switch for all.

   Poly is used in base class (typically) to access operations on current *concrete implementation* selected via common *abstract implementation* (*implementor*) interface. Client uses poly over various *refined abstractions* of a common *abstraction* interface, each of which uses base class (typically) to access current *concrete implementation*.

c) Factory method: hierarchies of *creators* and associated *products*. Each *concrete creator* knows about the specific *concrete product* it creates. Typically the creator is someone who wants to use the product, e.g. it is a related application class, who (therefore) knows exactly what type of object it wants.

   So the client (application) uses a poly *create()* method on some component, which then polymorphically executes the appropriate A_i::*create()* method of that object to instantiate the proper concrete sub-class type object. Note that A_i::*create()* uses a static binding to it's related concrete product, (*return new* P_j*()*)it is a fixed, known relationship.

   So, this answer is easy; the poly method is named in the title – the *create()* (≈factory) method!

d) Iterator: The *iterator pattern* includes an *iterator factory* to create iterators associated with each iteration (collection) class. this is just a *factory method* pattern, applied to iterators..

e) Abstract factory: Hierarchy of factories, and a *series* of hierarchies of related abstract products (a *family* of types!), each with sub-type trees of same structure (since each product has to be available in each variety of production (each factory type).

   The creation of *concrete factories* by the *abstract factory* could be done in a variety of ways, and could involve poly (as above), since AbsFactory ∈ Factory!

   But the important poly here is in the usage of the actual *concrete factory* objects. Once the client has an actual (concrete) factory, it makes *createXxx()* calls to it, which are poly bound to the appropriate (actual concrete) factory (sub-)class used. That class has static binding to the actual concrete product classes to be instantiated. So, we need poly on the *create()*'s, because we don't know the actual sub-type of factory we have, which family line of types we are creating.

   So, this answer is (also) easy; the poly method is named in the title – the abstract *create()* (≈factory) methods!

Question:

Factory Method and *Visitor* design patterns both have dual hierarchies that the pattern connects; so in this way they are similar.

a) Describe how and where polymorphism and overloading (if any) is used in the *abstract factory* pattern. (Give the names and signatures of the methods).

b) Describe how and where polymorphism and overloading (if any) is used in the *visitor pattern*. (Give the names and signatures of the methods).

c) How are they similar and how are they different?

Solution:

a) *Factory method* is a <u>*creational* pattern</u>, *visitor* is <u>behavioral</u>; very different.

b) <u>Visitor</u>: Data and Operations hierarchies, with poly over both. Client uses poly to choose a sub-type of data for elements in a typed collection, and to choose an operation to apply, from a derived type fo the operation family. Visitor pattern then used double-poly to actually apply the correct operation to each concrete data element in the collection.

V::visit(D_i *d)* = Overloaded (@V_j) Overridden (@D_i)

D_i *::accept(Visitor v)* = Overridden (@D_i)

<u>Discussion</u>: Recall that the visitor pattern has two parallel hierarchies, one of data objects, $\{D_i \subset D\}$, where D is the (abstract) base class type of the hierarchy of concrete data object types, and a related set of visitors ($V_j \subset V\}$, where V is the abstract base of the tree of derived concrete visitor classes.

The *Visitor::visit(*D_i *d)* methods are overloaded, one for each sub-type D_i , in each concrete visitor class, <u>and overridden</u> from *V::visit()*, one (set) for each concrete visitor type V_j. Note the 2D element here, these *visit()* methods are both overloaded @D_i and overridden @V_j – this is the *double-polymorphism* of this pattern.

The D_i *::accept(Visitor v)* methods override the base class *accept()* method, and when called by the traversal iteration of the data objects, the appropriate *accept()* method is invoked by polymorphism on the actual (sub-)type of the data object used. Note that the Iterator (traversal client) uses *D.accept()* calls, because it does not know the actual dynamic types of the objects referenced in the collection elements, and it is polymorphism via the actual dynamic types of the referenced objects that the correct method is called.

In this dynamically bound method, each item then invokes the *visit()* method on themselves: *v.visit(this)*, where $\{this \in D_i\}$, and the appropriate[overloaded] V_j*.visit(*D_i*i)* message is chosen, and further the actual method used (dynamically bound to this message) is based on the actual type of the concrete visitor v $\{v \in V_j \subset V\}$ of the visitor hierarchy (V); thus the second level of the *double-dispatch*.

D_i::accept(visitor v) {
    v.visit(*this*); // (*this* $\in D_i$), (v $\in$ V)
    }

To summarize:

D::accept(visitor v)$\big|_{msg}$

(poly) $\rightarrow$ D_i::accept(visitor v) $\big|_{method}$

(calls) $\rightarrow$ v.visit(D_i) $\big|_{msg}$

(poly) $\rightarrow$ V_j::visit(item D_i) $\big|_{method}$

c) <u>Factory method</u>: hierarchies of *creators* and <u>associated</u> *products*. Each *concrete creator* knows about the specific *concrete product* it creates. Typically the creator is someone who wants to use the product, e.g. it is a related application class, who (therefore) knows exactly what type of object it wants.

So the client (application) uses a poly *create()* method on some component, which then polymorphically executes the appropriate A_i*::create()* method of that object to instantiate the proper

concrete sub-class type object. Note that A_i::*create()* uses a static binding to it's related concrete product, (*return new* P_j*()*)it is a fixed, known relationship.

So, this answer is easy; the poly method is named in the title – the *create()* (≈factory) method!

d) Abstract factory: Hierarchy of factories, and a *series* of hierarchies of related abstract products (a *family* of types!), each with sub-type trees of same structure (since each product has to be available in each variety of production (each factory type).

The creation of *concrete factories* by the *abstract factory* could be done in a variety of ways, and could involve poly (as above), since AbsFactory ∈ Factory!.

But the important poly here is in the usage of the actual *concrete factory* objects. Once the client has an actual (concrete) factory, it makes *createXxx()* calls to it, which are poly bound to the appropriate (actual concrete) factory (sub-)class used. That class has static binding to the actual concrete product classes to be instantiated. So, we need poly on the *create()*'s, because we don't know the actual sub-type of factory we have, which family line of types we are creating.

So, this answer is (also) easy; the poly method is named in the title – the abstract *create*P_i*()* (≈factory) methods!

Summary: Very different intent and usage.
Abstract Factory: Poly used to access actual *concrete factory*, via abstract methods in the *AbstractFactory* interface, to do actual *createProduct()*$|_i$. Since the returned product is upcast ($D_i \rightarrow D$), all method calls to it are also poly. A simple way to say it; poly over factories, and products!

No overloading. (Could overload *create(*P_i*)* methods).

Visitor: Poly used to invoke *accept*(v_j) method of appropriate D_1 object, and then it uses poly to invoke appropriate V_j.*visit*(D_i) method for visitor object.

Overloading can be used to have single name *visit*(D_i) instead of *visit*D_i(D_i).

The poly in *abstract factory* takes place at two (different) times, once at object creation time (the factory *create()* poly), and another (later…) on all uses of the object methods.

The poly on visitors takes place all at once, the two levels transparent to the caller.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Abstract Factory	Adaptor	Interpreter
				Template Method
	Object	Abstract Factory	Adaptor	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Façade	Momento
			Proxy	Flyweight
				Observer
				State
				Strategy
				Visitor