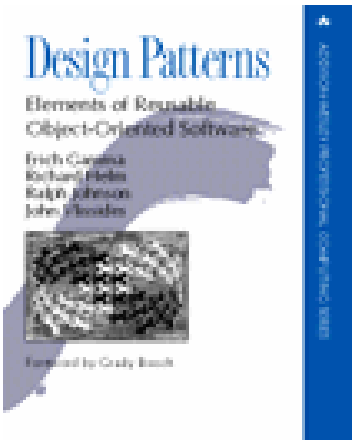
Design Patterns

3 day workshop

Presenter: Andy Bulka
http://www.atug.com/andypatterns
abulka@netspace.net.au

On behalf of
InterSkill Pty Ltd



Acknowledgements

Some materials in this handout have been taken from the following sources:

Huston Design Patterns
http://home.earthlink.net/~huston2/dp/patterns.html

Data & object factory.
http://www.dofactory.com/Patterns/Patterns.aspx#list

Some diagrams from
www.javacoder.net

Contents

Abstract Factory

Allows you to create an instance of a factory which itself can create a family of classes.

Definition

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

Intent

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

Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

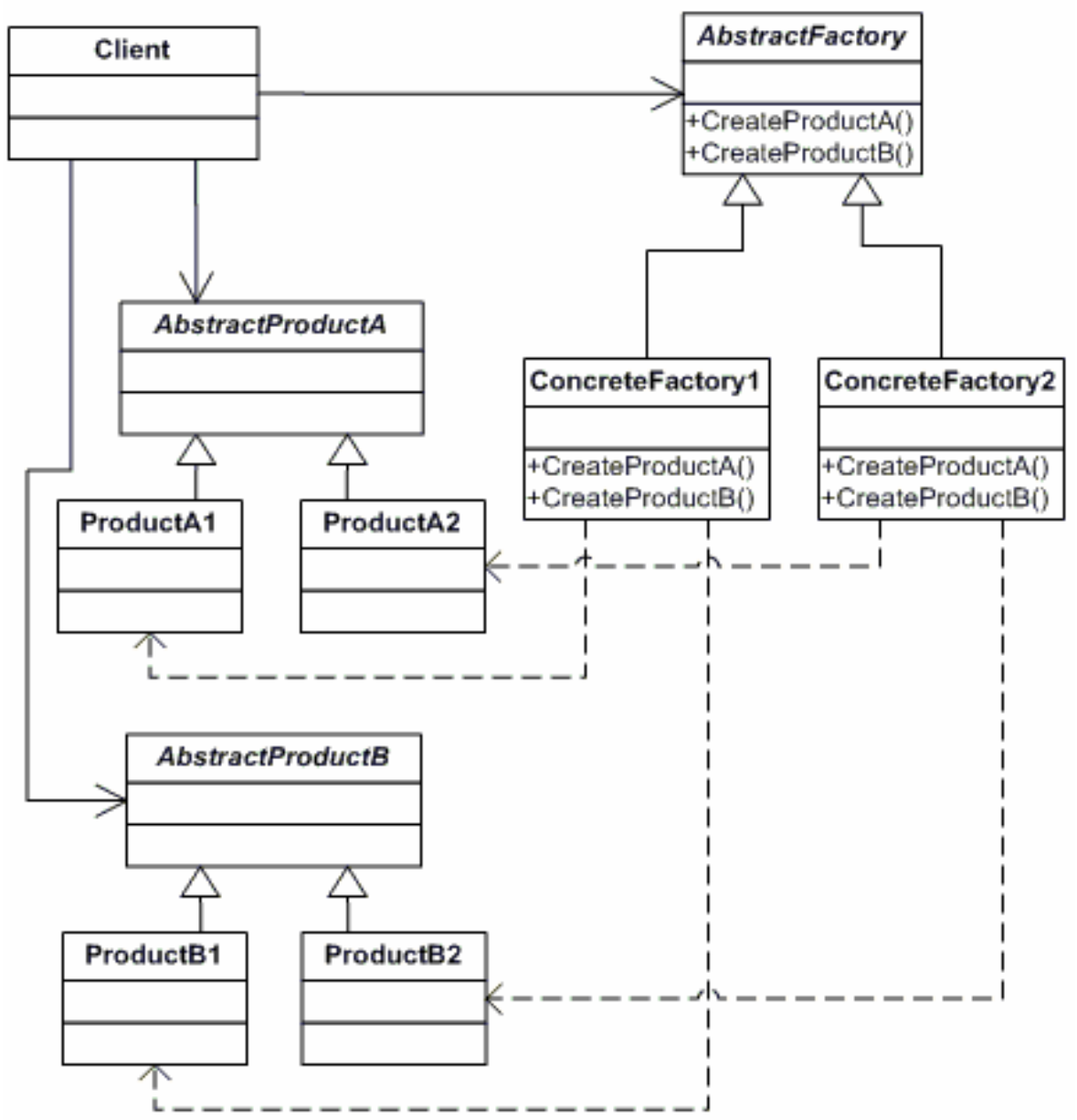
Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

Andy Insight: You can add a new a new Concrete Factory (and related product classes) and *voila*, the client code doesn't know about this, and you have your client code seamlessly driving a totally different implementation (product family). You can pull off this trick only if you follow the rule that the client code only talks to the AbstractProduct interface. Low coupling.

Abstract Factory



Andy's tips on Abstract Factory

- Abstract factory similar to factory method, in that there is something being overridden.
- Abstract factory is the same as factory method, except there is more than one Creation method. E.g. CreateWorker, CreateAdministrator, CreatePoliceman - such that the class containing the factory methods might as well become a sole purpose class for dispensing these related classes.
- The abstract factory is a mere mechanism for delivering A versions or B versions. E.g. Client wants A version of products
- Client programs against interfaces thus can switch between A or B. Specifically, the client only talks to

  IAbstractProductFactory
  IProduct1
  IProduct2
  IProduct3 etc.

```
IAbstractProductFactory f =
                        new ProductFactoryVersionA()
// above choice is made at compile time,
// via factory method (run time)

// Client code can now...
IProduct1 p1 = f.CreateProduct1()
IProduct2 p2 = f.CreateProduct2()
IProduct3 p3 = f.CreateProduct3()
```

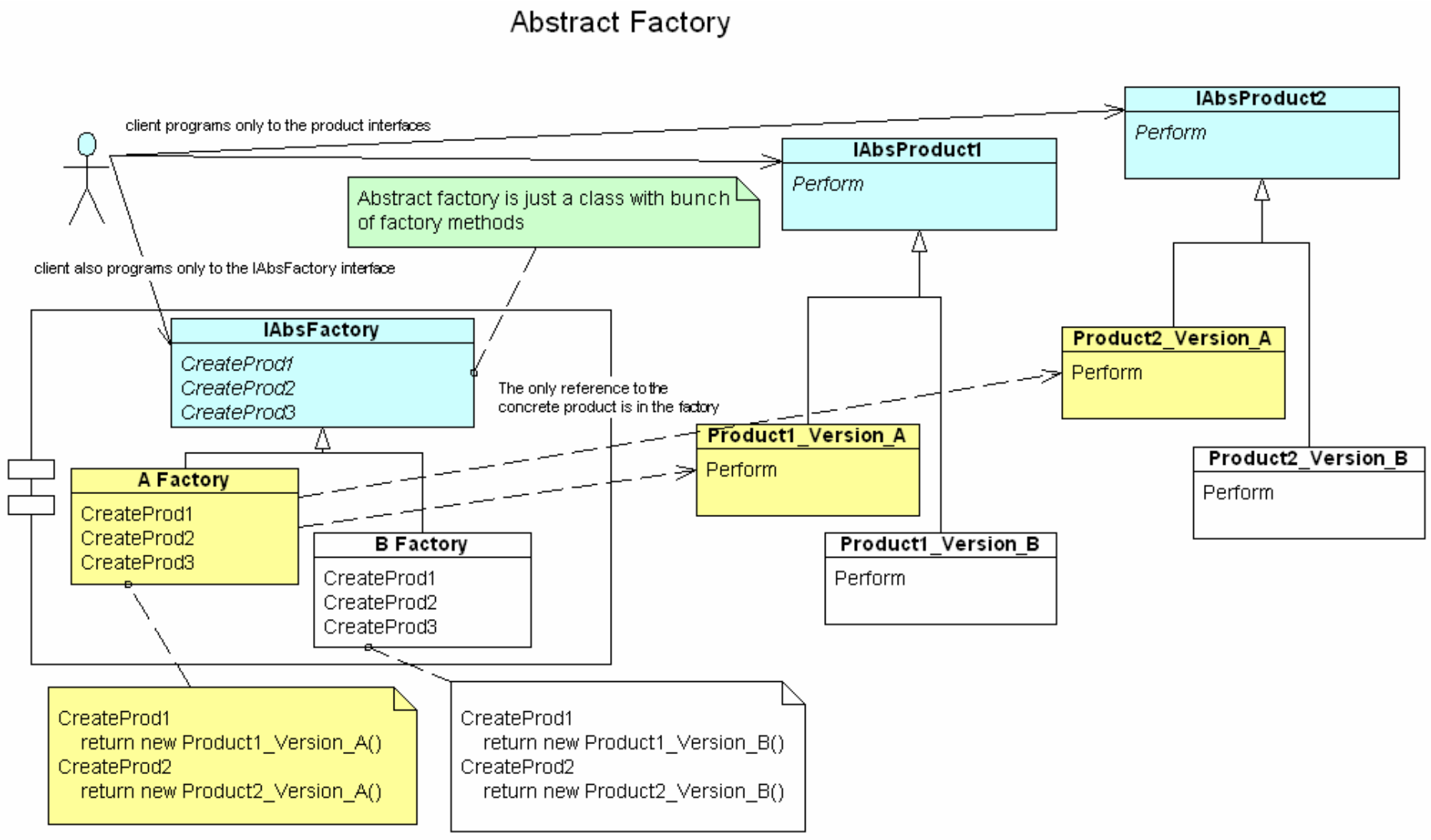
All products p1, p2, p3 are in the above example are A versions, and compatible with each other.

Puzzle

```
factory = GetAppropriateFactory("3D")
  or
factory = GetAppropriateFactory("2D")

factory.CreateBox()
factory.CreateCircle()
factory.DrawLine()    // is this right?
```

Abstract Factory



Abstract Factory

Creates an instance of several families of classes.

Andy's thoughts on Abstract Factory methods

What do the factory methods on an Abstract Factory do?
The factory methods on an Abstract Factory should, in my opinion, should

- only create 'products' - which are instances of classes
- should have the phrase 'create' somewhere in its name e.g. CreateProd1, or CreateProd2
- each product should be of a different class type
- should not do any real work or computation

I once saw a presentation on Abstract Factory where the factory methods on the concrete abstract factory class actually did complex computational work (each one did something different) and all returned a result class of the same type (a data set, as it happens). This is not abstract factory. The methods in the factory should not have done any real WORK just merely returned DIFFERENT 'products'. Products are a bunch of different classes which are meant to be instantiated as a family of classes, which typically work together. Thus the idea of an 'abstract factory class generating a FAMILY of DIFFERENT but relatedclasses' was entirely missing, and this it seems to me to be the essence of abstract factory. What was being actually presented was simply a compile time choosing of one of two different implementations of an interface, which I call the Interface Pattern and discussed here.
http://www.atug.com/andypatterns/the_road_to_bridge.htm#Interface pattern

Non-software Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the

other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

www.dofactory.com’s structural code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

www.dofactory.com’s real-world code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

Notes:

Builder

Definition

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Discussion

Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/ distinction is on creating complex aggregates.

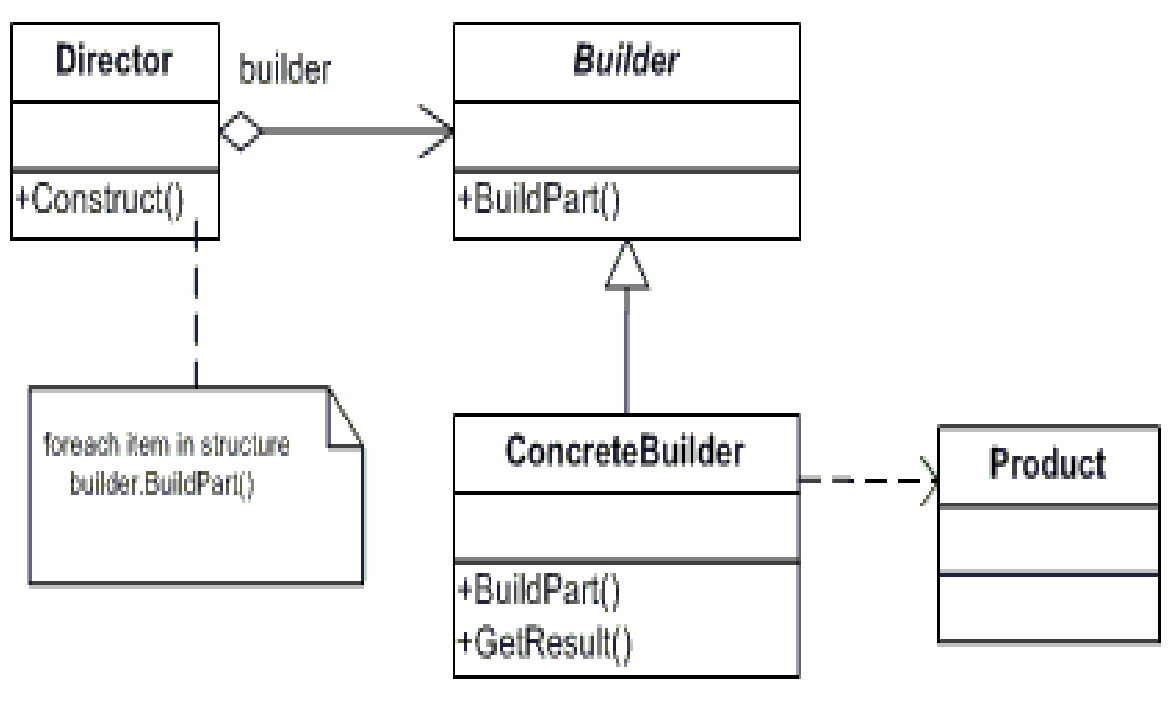
The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy car). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Builder


Director
builder
Builder
+Construct()
+BuildPart()
foreach item in structure
builder.BuildPart()
ConcreteBuilder
Product
+BuildPart()
+GetResult()

Andy's Tips on Builder

- Avoids creating invalid/incomplete complex objects esp. when reading from streams etc.
- Like abstract factory or factory method – it builds things.
- Intent is to pass the builder lots of info (esp. when reading from streams etc.) and then when all of the info in in, ask for the finished object, which is guaranteed to be valid. **Thus complex business logic for validation can live in the builder rather than in the final object itself.**
- Builder delays building the object and avoids the situation where you first create the object, fill in some of its attributes, and end up not having received enough info from the stream/source thus end up with an invalid object.
- Strict Builders and Forgiving Builder which fills in missing informatin for you.
- Separates the construction of a complex object from its representation.

Non Software Examples

The use of a bread machine can be considered to be a *Builder*. If a wife asks her husband to make some cinnamon walnut bread, she does not care what process he uses. Whatever process is used, the wife expects a loaf of bread, containing cinnamon and walnuts.

- The algorithm for creating a complex subsystem shall be independent of the parts that make up the subsystem and how the parts are assembled.
- The creation process for a complex subsystem shall support multiple representations of that subsystem.

www.dofactory.com's structural code demonstrates the Builder pattern in which complex objects are created in a step-by-step fashion. The construction process can create different object representations and provides a high level of control over the assembly of the objects.

www.dofactory.com's real-world code demonstates the Builder pattern in which different vehicles are assembled in a step-by-step fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

Builder

Non-software example

Rules of thumb

Sometimes creational patterns are complementory: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Builder is to creation as Strategy is to algorithm. [Icon, p8-13]

Builder often builds a Composite. [GOF, p106]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Participants

The classes and/or objects participating in this pattern are:

* Builder (VehicleBuilder)
  - specifies an abstract interface for creating parts of a Product object
* ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
* Director (Shop)
  - constructs an object using the Builder interface

Notes:

Factory Method

Creates an instance of several derived classes

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

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.

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. [GOF, p136]

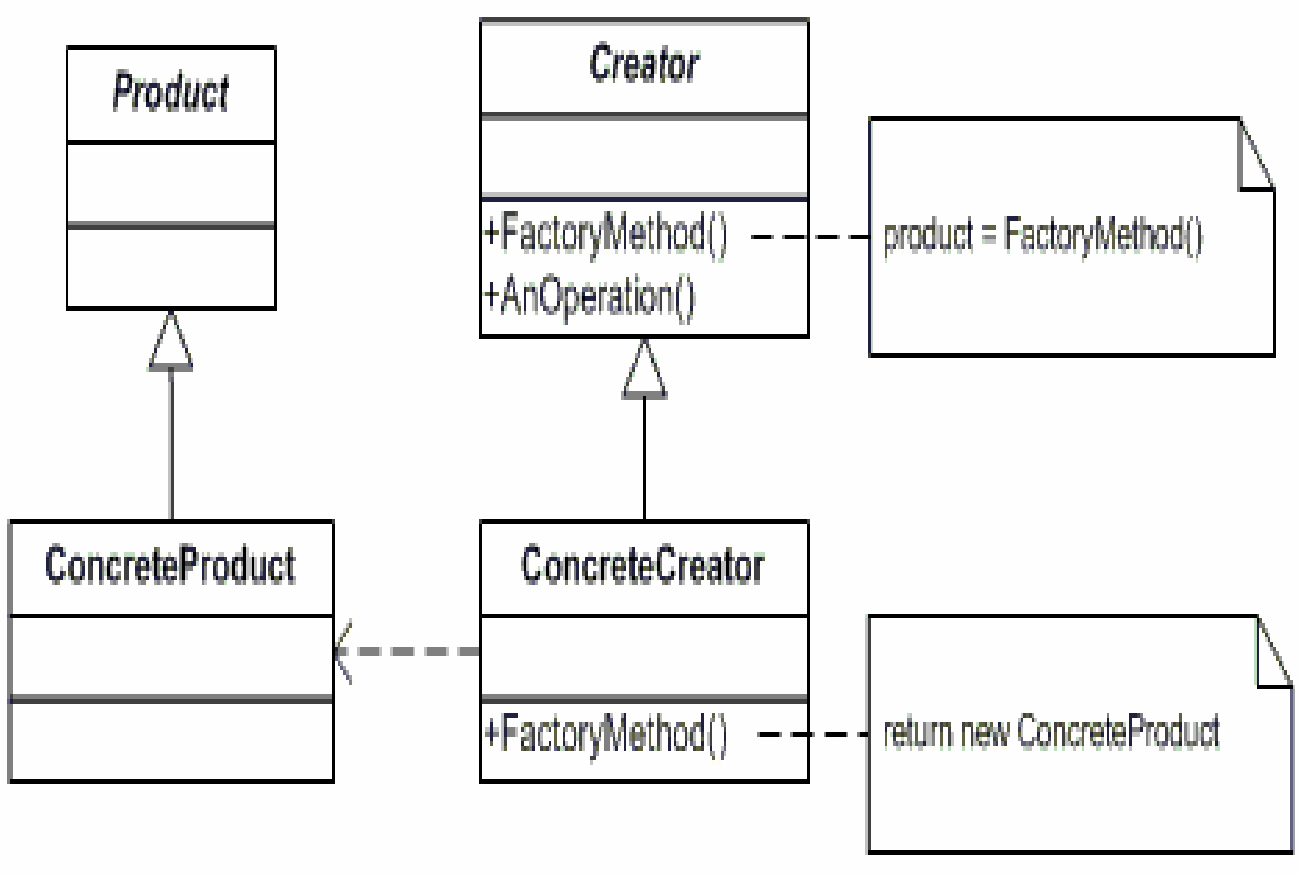
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation). [GOF, p136]

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.

- A subsystem shall not have the "type" of component it is responsible for creating hard-wired.
- The system shall be "open for extension, but closed for modification".

Factory Method



Factory Method

Non Software Example

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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

A bread machine is an example of a *Factory Method.* The machine provides an interface for creating bread, but the exact type of bread is deferred until a recipe is followed. The same machine can be used to make white bread, wheat bread, cheese bread or cinnamon bread.

Participants

The classes and/or objects participating in this pattern are:

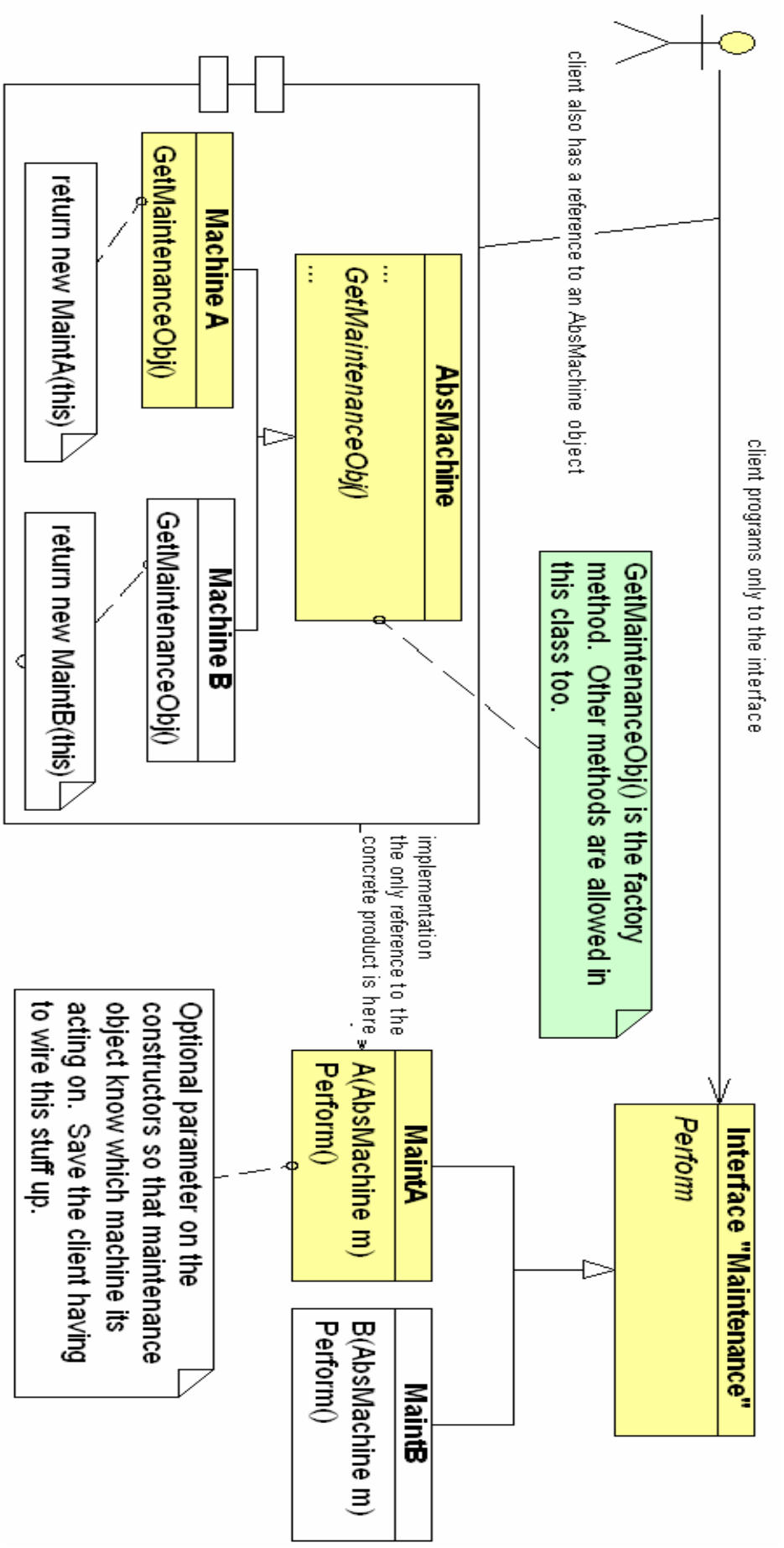
* Product (Page)
  - defines the interface of objects the factory method creates
* ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)
  - implements the Product interface
* Creator (Document)
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
* ConcreteCreator (Report, Resume)
  - overrides the factory method to return an instance of a ConcreteProduct.

Notes on the diagram opposite:

- Subclasses override the factory method *method* and return the appropriate product relating to their subtype, using code.

- Client refers to the abs prod. interface only.

Factory Method



Factory Method

Rules of thumb

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Methods are usually called within Template Methods. [GOF, p116]

Factory Method: creation through inheritance. Prototype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

www.dofactory.com’s structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

www.dofactory.com’s real-world code demonstrates the Factory method offering flexibility in creating different documents. The derived Document classes Report and Resume instantiate extended versions of the Document class. Here, the Factory Method is called in the constructor of the Document base class.

The road to Factory Method – Article

Taken from the article by Andy Bulka at:
`http://www.atug.com/andypatterns/the_road_to_bridge.htm`

This story leads us from the idea of what a ‘factory’ is trying to achieve, through some basic alternate implementations of factories, and then to the ’factory method’ style implementation presented by GOF (the original design patterns book).

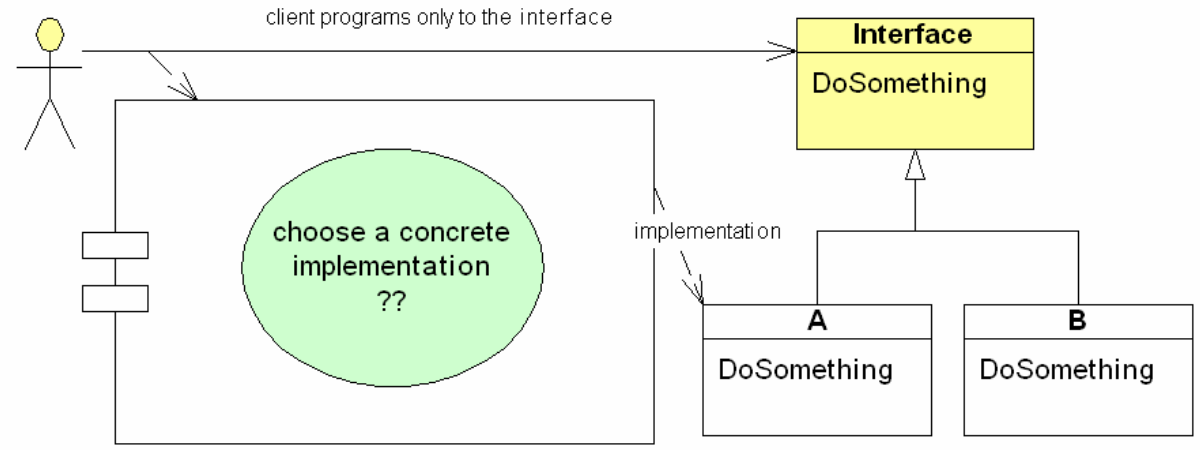
Note that design patterns are mean to be ’ideas’ and not to be tied to particular implementations. As such, any ’method’ that when called, returns an instance of a class is a ’factory method’.

The idea of a Factory

Create A or B at runtime by asking another class to create the concrete object for us. Pass in the flag to the factory or let the factory decide for itself which implementation we want.

Factory class is the only class to refer to concrete products. The client refers to the interface/abstract class only.

We are still talking directly to the concrete object (**either an A or a B**).



The road to Factory Method

There are a number of factory method variants:

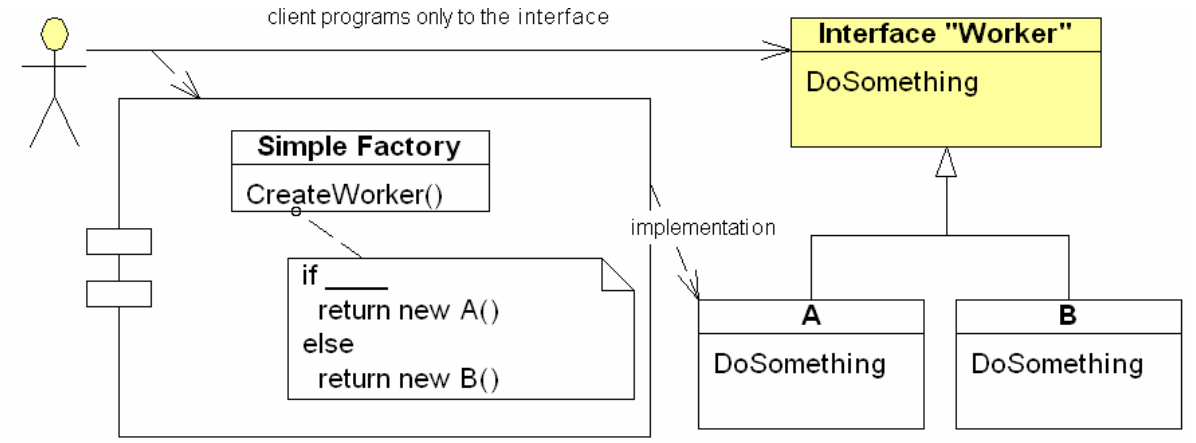
Simple Super Dumb Factory

Encapsulates the "dynamic run time choice" solution discussed in the beginning of this talk. Benefit is that the conditional logic containing the if statement is hidden and possibly centralized in a factory class.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via conditional code.

```
Factory f = new SimpleFactory()
Worker o = f.CreateWorker()
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```



The road to Factory Method

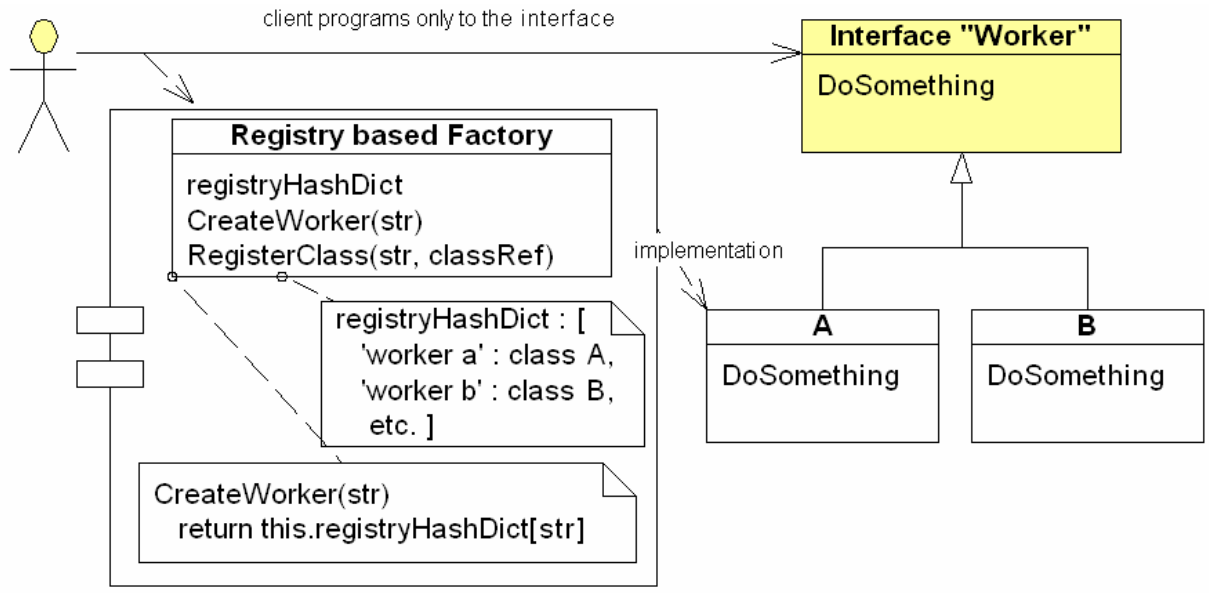
Registry Based Factory

Maintains a registry of mappings between strings (or any type of key e.g. objects, class references, numbers etc.) and class references. Benefit: more generalized, no if statements.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via registry key.

```
key = 'worker a'   // in setup code somewhere
Factory f = new RegistryFactory()
Worker o = f.CreateWorker(key)
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```



The road to Factory Method

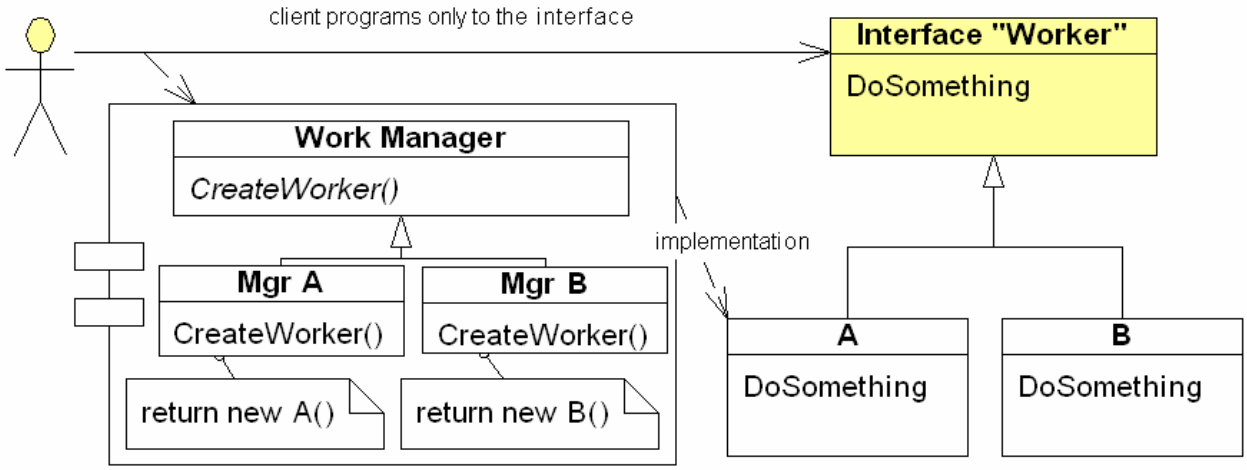
GOF Factory Method

Assumes the client *already has* an instance of some class which needs either a A or B version of a worker class.

Each alternative instance of the existing class overrides a create method differently, each instantiating a different concrete product - typically one matching their own functionality. Benefit: no class reference language facilities required.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via polymorphic override.



The road to Factory Method

Note that the choice as to which Work Manager (MgrA or MgrB) to instantiate in the first place is going to be an issue, but is not the point of this example. The point is that once you have a particular brand of work manager, then you will get a related brand of of worker via the suitably overriden CreateWorker factory method.

```
WorkManager f = new MgrA()      // done somewhere in setup
Worker o = f.CreateWorker(key)
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

There will be parallel hierarchies, e.g. the WorkManager and the Worker hierarchies closely match, with A and B versions of their subclasses. Start to think of a *family* of classes.

Andy's Final Tips

- We create a class by calling a **method**, and we don't know exactly which class we are going to get. Though typically we *do know* what interface or abstract class the instance supports (so that we know how to 'drive' it).
- GetEnumerator() is a factory method. It returns an interator object – but the type we get depends on the collection we are iterating over.

Notes:

Prototype

A fully initialized instance to be copied or cloned

Definition

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Problem

Application "hard wires" the class of object to create in each "new" expression.

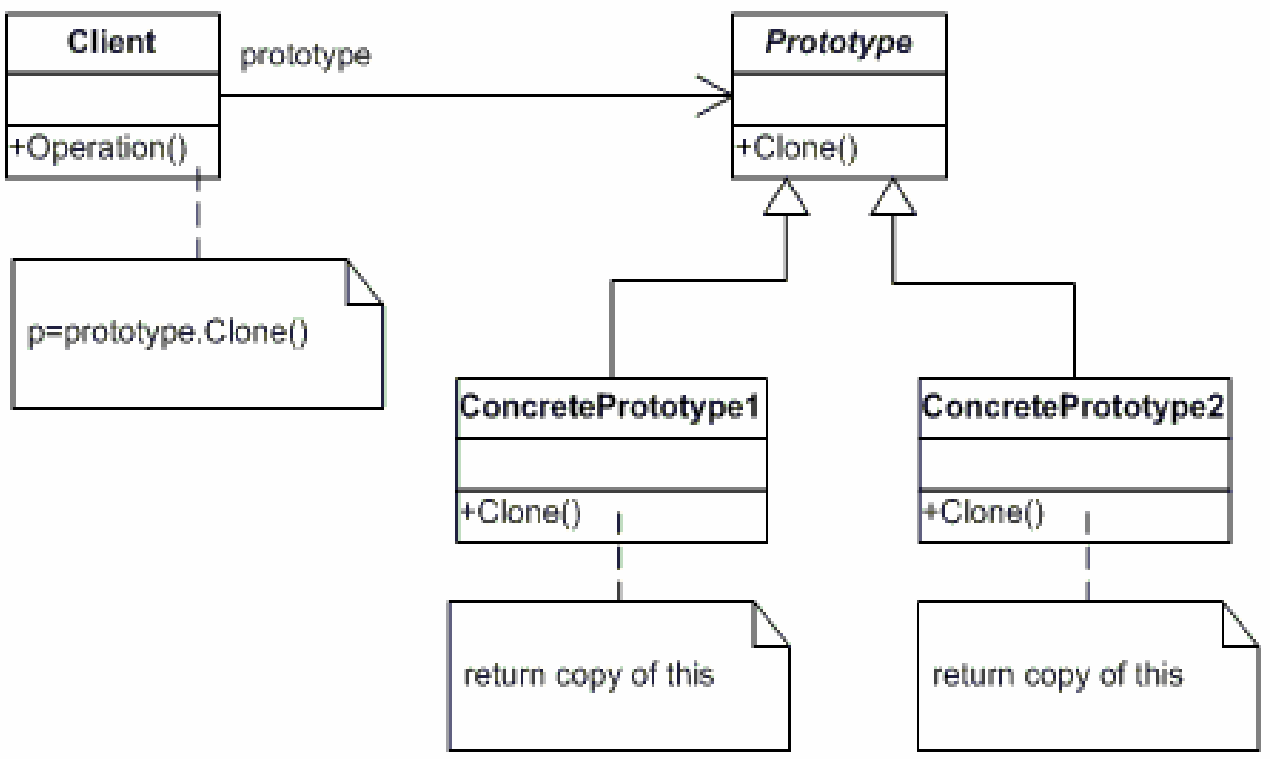
Discussion

Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

- The overhead of component creation shall be localized and minimized. Instead of creating entirely new instances of a component type, a "prototypical" instance of each type shall be designed to "clone" itself.
- Decisions about the type of component to create shall be deferred until application run-time.

Prototype



Participants

The classes and/or objects participating in the Prototype pattern are:

* Prototype (ColorPrototype)
  - declares an interace for cloning itself
* ConcretePrototype (Color)
  - implements an operation for cloning itself
* Client (ColorManager)
  - creates a new object by asking a prototype to clone itself

Prototype

Andy's Tips

- Java has a cloneable interface
- Deep vs. Shallow copy
- Use serialisation to deep copy. Watch out for circular references.
- Use a prototype registry (loaded from disk) from which to clone. Dynamic loading is a plus.
- Create new instances by copying a prototype – a complex object, which has been streamed/pickled to disk. Example, a graphics library, a character editor. Can dynamically define new types of 'classes' by composing a complex object, then registering it as a prototype! :-) No need for a special class or programmer intervention. E.g. www.dofactory.com has a good C# example of a custom colour repository.
- Initialisation issue – no parameters on the clone() method allowed. Thus add your own Init() or Start() method to the cone class and call it after the clone operation has completed.
- Example

```
p2 = p.clone()
while drag
    p2.draw(newpos)
insert p2 into drawing
```

www.dofactory.com's structural code demonstrates the Prototype pattern in which new objects are created by copying pre-existing objects (prototypes) of the same class.

www.dofactory.com's real-world code demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.

Prototype

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126]. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134].

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Method: creation through inheritance. Protoype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. [GOF, p126]

Non Software Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Notes:

Singleton

A class of which only a single instance can exist.

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

Singleton

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

Problem
Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion
Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

* Ownership of the single instance cannot be reasonably assigned
* Lazy initialization is desirable
* Global access is not otherwise provided for

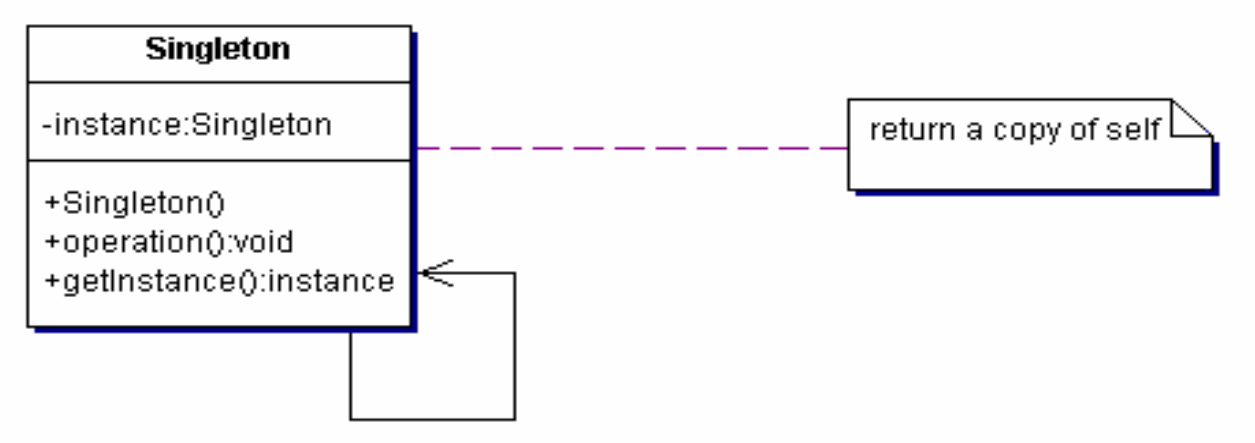
If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Deleting a Singleton class/instance is a non-trivial design problem. See "To kill a Singleton" by John Vlissides (C++ Report, Jun 96, pp10-19) for a discussion.

Singleton




```
GetInstance(){
  if _instance = Null {
    _instance = new Singleton()
  }
  return _instance
}
```

Participants

The classes and/or objects participating in the Singleton pattern are:

* Singleton (LoadBalancer)
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

Singleton

Andy's Tips

- Ensure class has only **one instance** and provide a global point of access to it.
- Can control access strictly.
- Like global without being a global variable.
- Must disable the constructor and use getInstance() instead. To do this in java (see Desifn Patterns in Java p 132.) you create a *private constructor* which effectively disables the constructor. Then make a GetInstance() method public static, (which *has* access to the private constructor).
- BORG pattern variant – allows many instances, but share common data. The argument here is that it's the data that's important to keep common, not how many instances of an object you have. Controversial? Easy to implement in Python – maybe not so easy in other languages, thus the need for Singleton (rather than Borg).
- Thread Safety – make GetInstance() synchronized or mutex/synch the block around the code.
- Usage:

```
s = new Singleton()    // WRONG
s1 = Singleton.GetInstance()
s2 = Singleton.GetInstance()
Assert(s1 == s2)  // the whole point of singleton!
```

www.dofactory.com's real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go throught the one object that has knowledge about the state of the (web) farm.

Non Software Example

The office of the President of the United States is a Singleton. There can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Singleton

Opinions

"Singleton is not going to solve global data problems. When I think of Singleton, I think of a pattern that allows me to ensure a class has only one instance. As far as global data, the only benefit I see is reducing the name-space." [Christopher Parrinello]

"This is not really the point, but I always teach Composite, Strategy, Template Method, and Factory Method before I teach Singleton. They are much more common, and most people are probably already using the last two. The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances." [Ralph Johnson]

Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" - even though this and that are the same thing. [Frieder Knauss]

Rules of thumb

Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, p134]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

State objects are often Singletons. [GOF, p313]

- A single instance of a subsystem shall be enforced, and the subsystem itself shall be responsible for that enforcement.
- The single instance shall be globally accessible.
- The single instance shall be initialized only if, and when, it is accessed.

Notes:

Adapter

Match interfaces of different classes

Definition

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem

An "off the shelf" component offers compelling functionality that you would like reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

Create an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

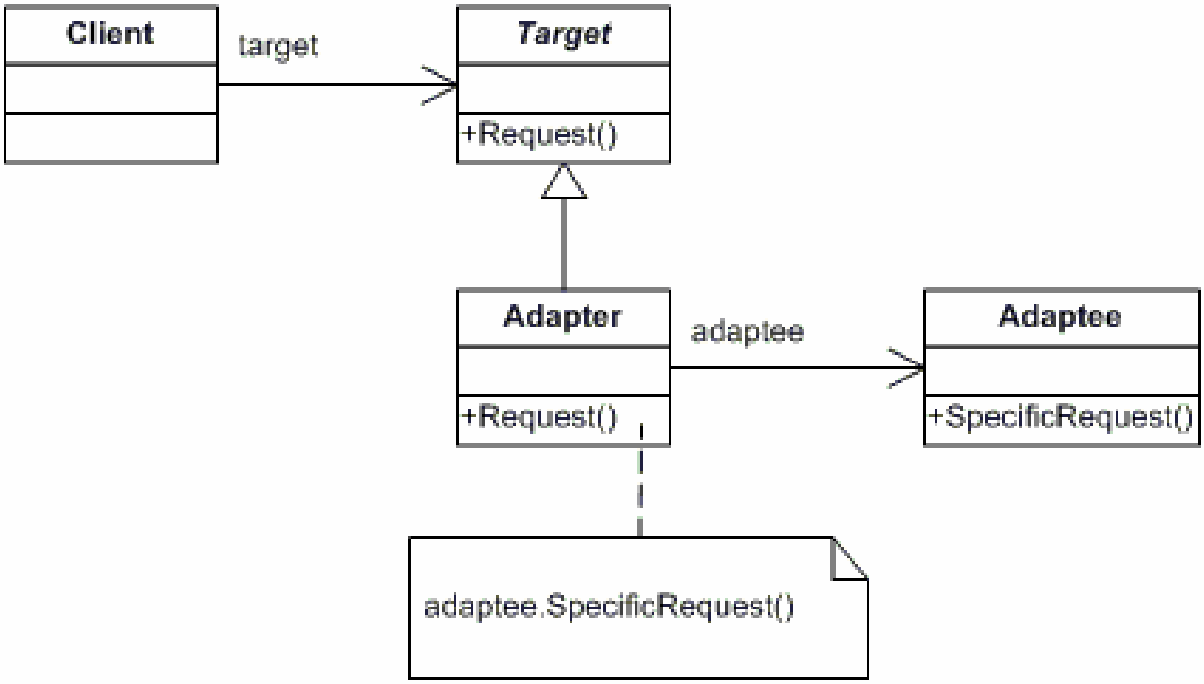
Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class. It effectively offers an impedence-matching facility.

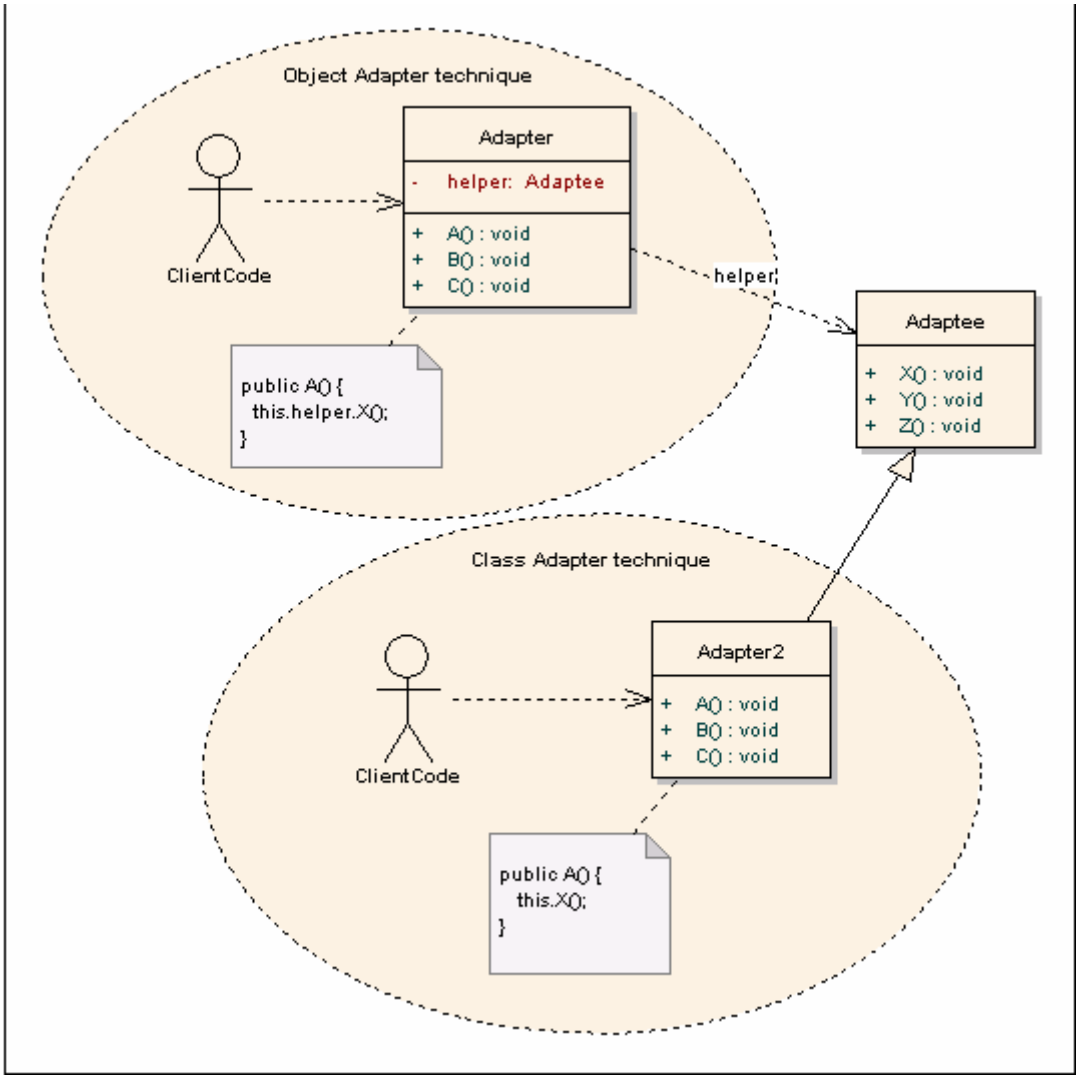
Structure

This diagram is from javacoder.net. It is using aggregation is its delegation mechanism. The inheritance shown is only for interface - not for implementation.

- A legacy component shall be reused in a new design, and the interface "impedance mismatch" shall be reconciled.
- A reusable component shall cooperate with unrelated (or unforeseen) application components.

Adapter


Client
target
Target
+Request()
Adapter
adaptee
Adaptee
+Request()
+SpecificRequest()
adaptee.SpecificRequest()


Object Adapter technique
Adapter
- helper: Adaptee
+ AO : void
+ BO : void
+ CO : void
ClientCode
helper
public AO {
this.helper.XO;
}
Adaptee
+ XO : void
+ YO : void
+ ZO : void
Class Adapter technique
Adapter2
+ AO : void
+ BO : void
+ CO : void
ClientCode
public AO {
this.XO;
}

Adapter

Andy's Tips

- Convert the interface of class into another interface more suitable.
- "Wrapper"
- Don't want to change the original code you are adapting:
  * e.g. because don't have the source code
  * e.g. or you don't want to affect existing legacy code which uses the code you want to adapt.
- Adds a level of indirection.
- Like strategy, but intent differs (Java Design Patterns p183).
- Object Adapter vs. Class Adapter. Object Adapter adapts through delegation. Class Adapter adapts through subclassing.
- Class Adapter: Why inherit – well, target may have attributes and other methods that we still want to use. Thus in te new adapter class we get to use, the old methods are still available (both a benefit and a liability). Though… class adaption may not work if you are not able to inherit from the old class (e.g. the adapter already iherits from something). Typically the new adapter class inherits from the old class (the one you are adapting) and implements the new adapter interface.
- Object Adapter: Wrapper. Gets its work done by forwarding requests to the helper/adaptee. Delegation

Participants

The classes and/or objects participating in the Adapter pattern are:

* Target (ChemicalCompound)
  - defines the domain-specific interface that Client uses.
* Adapter (Compound)
  - adapts the interface Adaptee to the Target interface.
* Adaptee (ChemicalDatabank)
  - defines an existing interface that needs adapting.
* Client (AdapterApp)
  - collaborates with objects conforming to the Target interface.

Adapter

www.dofactory.com’s structural code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

www.dofactory.com’s real-world code demonstrates the use of a legacy chemical databank. Chemical compound objects access the databank through an Adapter interface.

Non Software Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, p161]

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, 149]

Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, pp219]

Notes:

Bridge

Separates an object's interface from its implementation. "Driver" Pattern.

Definition

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

Intent

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

Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

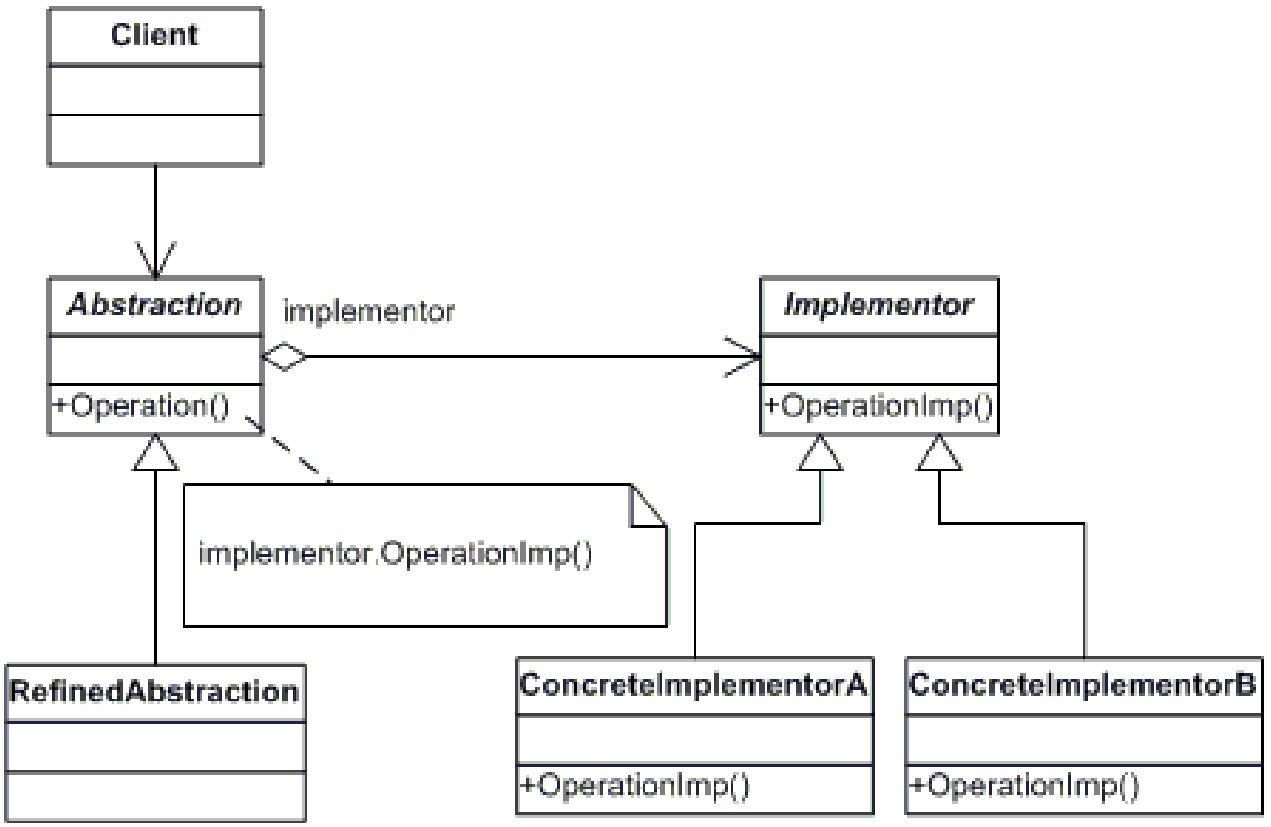
The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Consequences include:

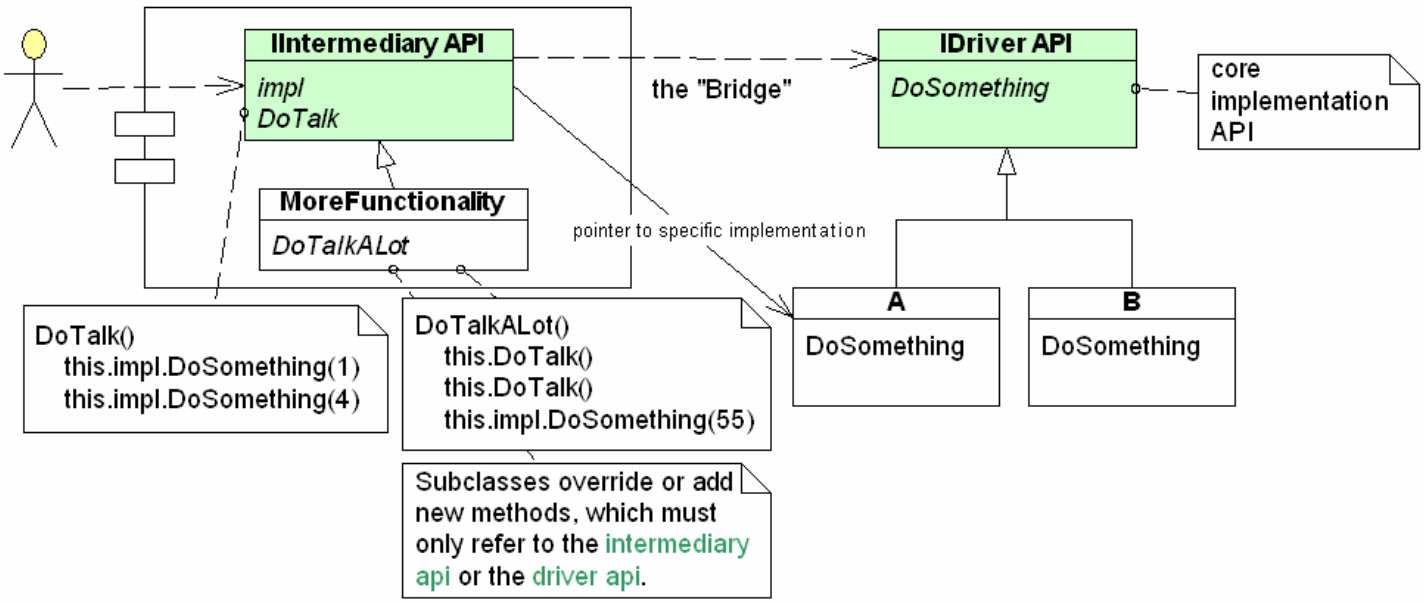
* decoupling the object's interface,
* improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
* hiding details from clients.

Bridge is a synonym for the "handle/body" idiom [Coplien, C++ Report, May 95, p58]. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)." [Coplien, Advanced C++, p62]

Bridge

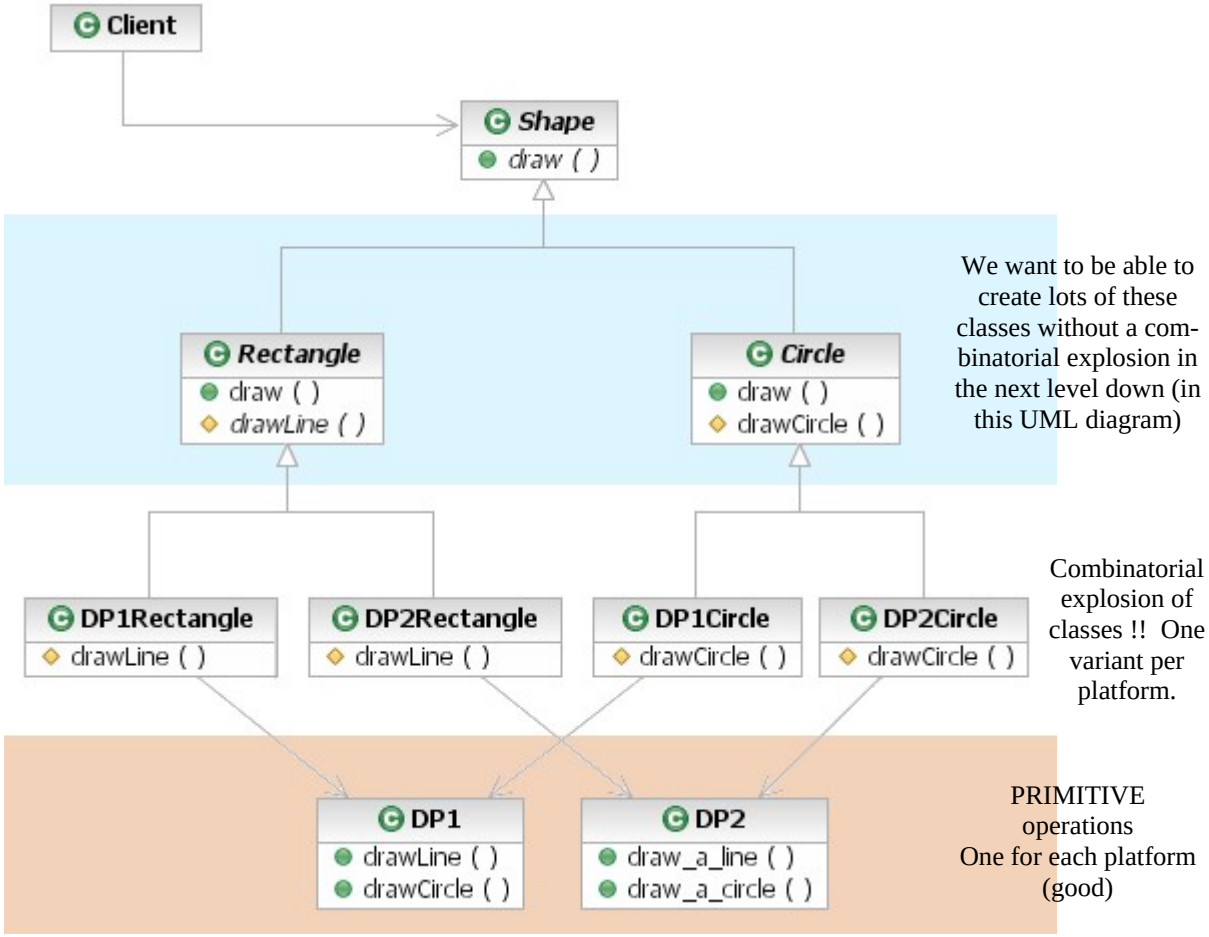


Pure Bridge Pattern



Bridge Pattern - slightly more detail

Bridge



Naïve "BAD" initial design.
(This is not the Bridge Pattern)

What are we trying to achieve?

We are trying to create a way of drawing rectangles and circles so that it works on both DP1 and DP2 where DP means "Drawing Platform" E.g. DP1 = Windows, DP2=Mac

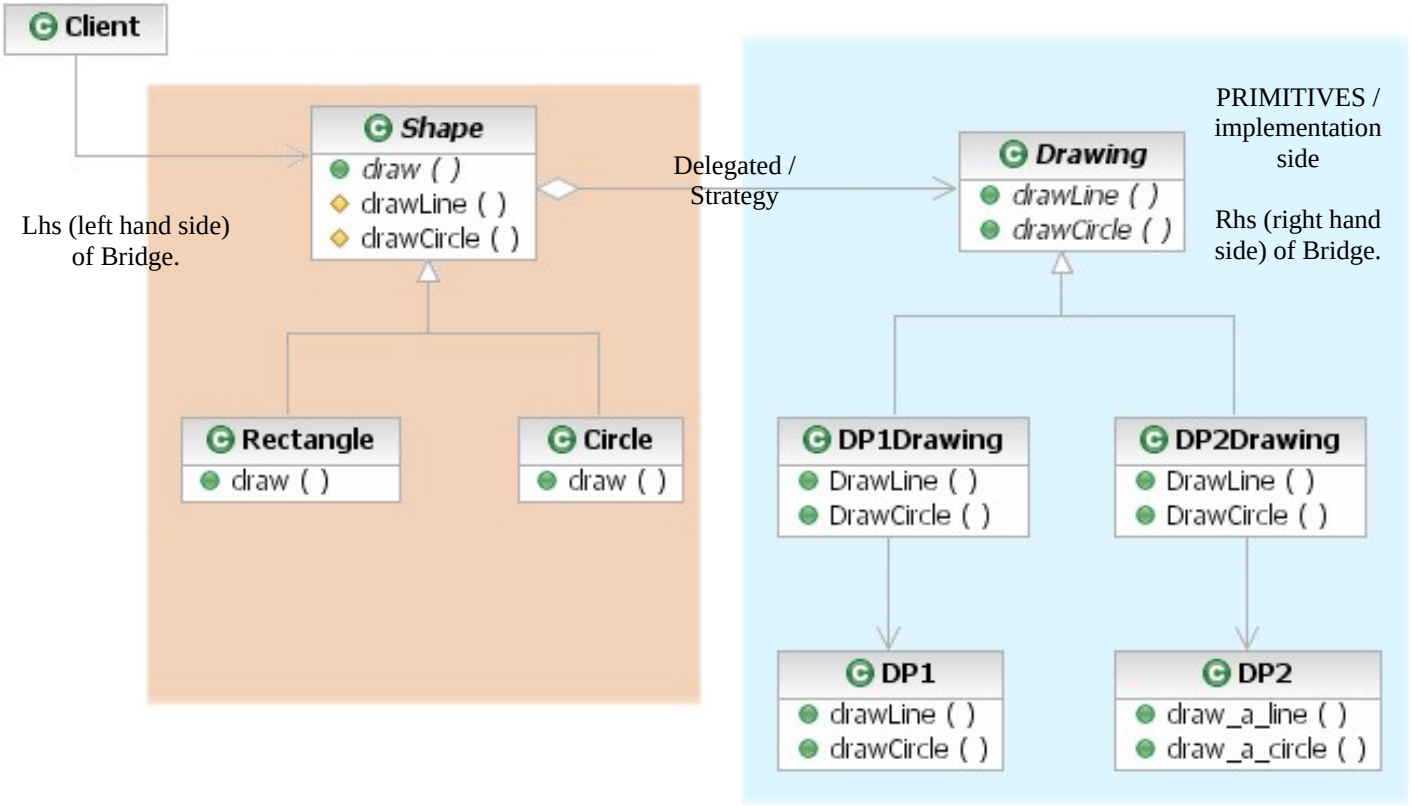
Why is this design good?

Well, at the lowest level of the UML diagram we have a nice bunch of primitive operations defined, one for each platform e.g. Mac/Windows. This seems clean. Also, we have organized our classes in an extensible way.

Why is this design bad?

- Too many classes are being created. Combinatorial explosion of classes !!
- If we want to add a new "Triangle" class, we would have to create 3 new classes e.g. "Triangle", "DP1Triangle" and "DP2Triangle"
- It would be nice if we could just create 1 new class e.g. for Triangle.

Bridge



Better design—using Bridge Pattern

Why is this design better?

Easy to add a new shape e.g. Triangle. Just write it once on the lhs. and we are finished! We don't have to create a DP1Triangle and a DP2Triangle. The single Triangle class we write in terms of calls to the primitive methods on the rhs.

Imagine a combobox shape - this is quite a complex bit of programming. Being able to implement it only once, on the lhs, is a big benefit.

What about the lowest common denominator problem?

How do we optimize and take advantage of a particular driver/rhs with a higher level feature? Bridge relies on a lowest common denominator API for the rhs, something all implementation platforms can easily supply.

POSSIBLE SOLUTION SUGGESTED BY ANDY: Have the rhs. implement a CanDoSmartThingBlah() method and have the lhs conditionally take advantage of it. So if the rhs. can draw a combobox shape natively then our lhs. Draw() code would:

```
if impl.CanDrawCombo()
    impl.DrawCombo()
else {
    do it the hard way
    in terms of multiple calls to rhs. primitives.
    ….
}
```

We would have to add CanDrawCombo() and DrawCombo() to the rhs. Drawing interface.

Bridge

Andy's Tips

- Bridge is closely related to the **Adapter** design pattern. However Adapter is usually as thin as possible on the lhs. vs. the lhs in Bridge is more complex, thick and hierarchical.
- This is a variant on accessing different behavior via an intermediary.
- Designed to separate a class's interface from its implementation, so that you can vary or replace implementations without changing client code (where 'client code' refers to any code that uses the interface in question.
- For this trick to work, the 'client code' must only talk to the interface/ abstraction.
- The abstraction / interface the client uses, in turn uses the interface of the implementation. So there are two interfaces, the left hand side (LHS) and the right hand side (RHS). The implementation of the left hand side interface uses the interface of the right hand side.
- Looks like strategy, Bridge is just strategy with a oversized lhs context. Same as strategy except there is massive subclassing going on on the lhs (the 'context' side). The nature of the lhs methods are more compositional, adaptive and far reaching (not just a simply strategy delegation)
- Abstract Factory often used to create one particular set of implementation classes on the right hand side.
- Examples are device drivers or database drivers. They force you to abstract a common model of the API. We lose some driver specific features.

Often the LHS interface API is a higher level of abstraction e.g. "drawing a box". The RHS implementation API is often a more primitive abstraction e.g. "drawing a line". e.g.

```
DrawRect() {
    // draw a rectangle by drawing a line four times
    this.impl.DrawLine(a,n);
    this.impl.DrawLine(n,m);
    this.impl.DrawLine(m,z);
    this.impl.DrawLine(z,a);
}
```

Bridge

Andy's Tips – part II

There is massive subclassing going on in the lhs. context

The reason is that you are wanting lots of methods and lots of functionality, lots of classes. E.g. you want to have a GUI or DB subsystem, not just a single strategy.

The nature of the lhs and rhs methods

Typically rhs (implementation/driver) calls are more primitive, and one lhs method will call the rhs. many times. e.g. see the DoTalk() method, above.

The lhs methods can be diverse, comprising

- lhs method simply calls rhs method. Method names can change or be the same. Simple delegation with no extra work.
- lhs methods more complex and adapt and do extra lines of code as needed
- Lots of logic in the lhs methods and may have associated helper classes. But in the end they call stuff on the intermediary api.

Insulated from change. Allow lhs and rhs to vary independently.

Client is insulated from changes. Should not talk talk to implementation, even if it is the abstract implementation interface because the abs impl. may change. If the abstract implementation interface does change then this affects only the Intermediary but not the client code. Client code should thus only talk to intermediary.

Similarly, if you change the Intermediary API, then only the client is affected - the r.h.s. (the abstract implementation interface and concrete implementations) are not affected.

In this sense the lhs and rhs can vary independently. Ok - so there are repercussions when things vary - but they are limited, as discussed above.

Final thought on Bridge

You could simplify Bridge and have the client code talk directly to the rhs. abstract implementation interface. You would be reverting to where we started on this "road to Bridge". Nothing wrong with that - but you would lose the 'insulation against change' that Bridge gets you. And with Bridge the lhs can have lots of complex logic and the rhs implementations need only implement the more primitive operations. That is a big win.

}

Bridge

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, 216]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58]. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, C++ Report, May 95, p58]

If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects. [Grand, Patterns in Java, p196]

- Component interfaces shall be decoupled from their implementation(s).
- The client shall be "insulated" from implementation changes. [Changes to implementation details should only require a re-link, not a re-compile.]
- The choice of implementation shall be configurable at run-time.
- Both the "interface" and the "implementation" shall be separately "open for extension, but closed for modification".
- "Interface" components that represent "implementation" components that are "expensive" and "equal" shall share these implementations, instead of duplicating them.

Use the Bridge pattern when:

* you want run-time binding of the implementation,
* you have a proliferation of classes resulting from a coupled interface and numerous implementations,
* you want to share an implementation among multiple objects,
* you need to map orthogonal class hierarchies.

Bridge

Participants

* Abstraction (BusinessObject)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
* RefinedAbstraction (CustomersBusinessObject)
  - extends the interface defined by Abstraction.
* Implementor (DataObject)
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
* ConcreteImplementor (CustomersDataObject)
  - implements the Implementor interface and defines its concrete implementation.

Non Software Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

www.dofactory.com’s structural code demonstrates the Bridge pattern which separates (decouples) the interface from its implementation. The implementation can evolve without changing clients which use the abstraction of the object.

www.dofactory.com’s real-world code demonstrates the Bridge pattern in which a BusinessObject abstraction is decoupled from the implementation in DataObject. The DataObject implementations can evolve dynamically without changing any clients.

Notes:

Composite

A tree structure of simple and composite objects

Definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Intent

Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Discussion

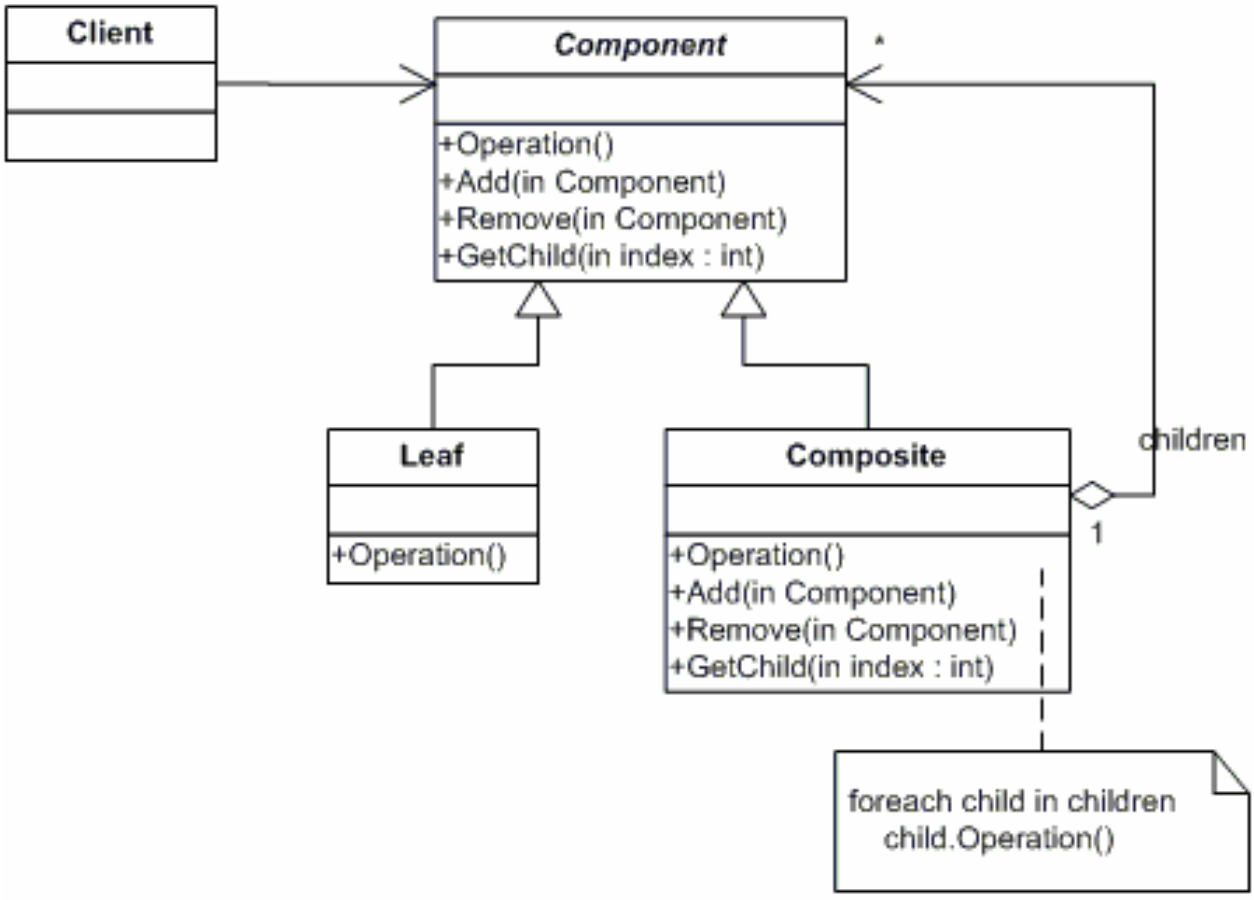
Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.

- The system shall support recursive composition.
- The system shall support "whole-part" hierarchical assemblies of components.
- Clients shall be able to transparently interact with compositions of components (e.g. a sub-tree) and individual components (e.g. a node).

Composite


Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Composite

Andy's Tips

- Tree structures and whole/part hierarchies let clients treat objects uniformly.

- As well as the operations

```
Add(o)
Remove(o)
GetChildren()
```

  you typically have some operation method on all the classes. E.g.

```
Traverse()
```

  where the implementation of the method on the composite class is different than on the leaf class. But the client code can just call the Traverse method (no knowing whether they are dealing with a composite or a leaf) and still get sensible results.

- A big implementation question with **Composite Pattern** is whether to implement methods like AddChild() in the base Component base class, or only on the composite class. If we want a consistent interface to the client (which deals only with components) then we should implement AddChild () in the base class. However do you then implement a vector/arraylist of children in the leaf class, or simply have the leaf class return an empty list? Should the leaf class implement AddChild(child) and if so - what does this mean - after all it's a nonsense operation for a leaf. So should AddChild() be on the base component class after all—perhaps its ok to have it there, its just that it does nothing or raises an exception if called on a leaf.

  Sometimes you might want the client to *know* whether they are dealing with a composite or a leaf. So you might add a IsLeaf() method. Other times you may not want to allow this sort of distinction to be able to be made. See opinions sections (next page) for more on this.

  My compromise solution is:
  - implement only guaranteed common operations on the base class
  - implement IsLeaf() on the base class and allow the client to access deeper functionality e.g. AddChild(c) safely if that's what the client code needs to do.

Composite

Opinions

The opinions below illustrate the debate about what methods to put in the base component class vs. which to put into the subclasses.

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. **At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.** [Don Roberts]

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces. [Bill Burcham]

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use dynamic_cast to figure this out. In those cases where dynamic_cast is too expensive, I will use a Visitor. [Robert Martin]

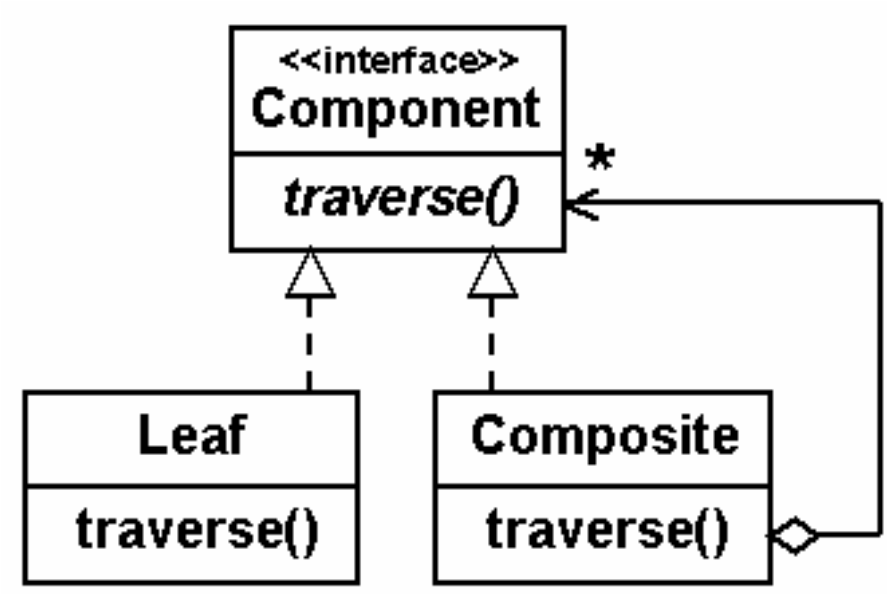
Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof. Just because Composite's structure diagram shows child management operations in the Component base class doesn't mean all implementations of the pattern must do the same. [John Vlissides]

Composite

Participants

The classes and/or objects participating in the Composite pattern are:

* Component (DrawingElement)
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
* Leaf (PrimitiveElement)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
* Composite (CompositeElement)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
* Client (CompositeApp)
  - manipulates objects in the composition through the Component interface.



Composite

Rules of thumb

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes. [GOF, pp173,349]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Non Software Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expresssion. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

www.dofactory.com’s structural code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

www.dofactory.com’s real-world code demonstrates the Composite pattern used in building a graphical tree structure made up of primitive nodes (lines, circles, etc) and composite nodes (groups of drawing elements that make up more complex elements).

Notes:

Decorator

Add responsibilities to objects dynamically

Definition

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

Intent

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

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.

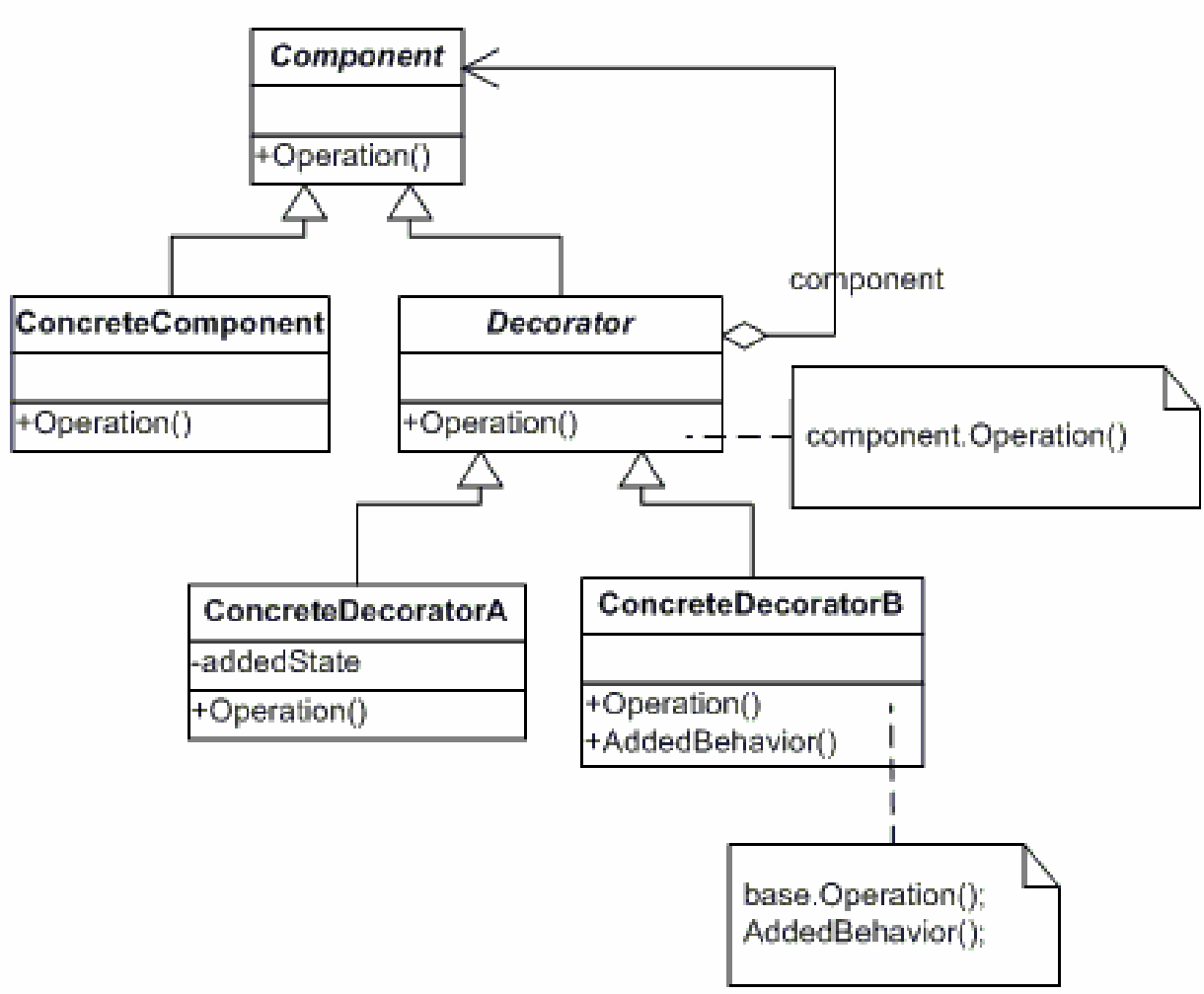
Participants

The classes and/or objects participating in the Decorator pattern are:

* Component (LibraryItem)
  - defines the interface for objects that can have responsibilities added to them dynamically.
* ConcreteComponent (Book, Video)
  - defines an object to which additional responsibilities can be attached.
* Decorator (Decorator)
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* ConcreteDecorator (Borrowable)
  - adds responsibilities to the component.

- Components shall be extensible at run-time.
- Functionality shall be "layer-able". The client shall be capable of configuring any combination of capabilities by simply specifying the layers (or wrappers, or onion skins) to be applied.

Decorator


Component
+Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();

Decorator

Andy's Tips

- Adds functions by wrapping a class
- Add responsibilities to a class more flexibly than inheritance.
- Examples are decorated borders or decorated strems.
- Structure similar to composite. Decorators inherit from the 'composite class'.
- A common operation implemented across a hierarchy.
- Resembles proxy, forward the call to the subordinate decorator object.
- A possible negative: type checking will fail i.e. the client code shouldn't check that the object it is dealing with is of a certain class, because it might be wrapped, and thus be of the decorated type.
- There are two senses in which there is 'order' in decorators.

  1. The first is the order in which decorators are chained together (this can be important e.g. in streams wrapping streams – these things may need to be done in a certain order).

  2. The second sense is *within* the operation method that each decorator implements. Each decorator must at some point call the same operation on the next decorator in the chain. A decorator method can call the next decorator *before* its own code or *after* its own code.

- Example:

```
t = Thing()
d1 = Decorator(t)
d2 = Decorator(d1)
d3 = Decorator(d2)
d3.Operation()
```

Decorator

Discussion

Suppose you have a user interface toolkit and you wish to make a border or scrolling feature available to clients without defining new subclasses of all existing classes. The client "attaches" the border or scrolling responsibility to only those objects requiring these capabilities.

```
Widget*  aWidget = new BorderDecorator(
                   new HorScrollDecorator(
                   new VerScrollDecorator(
                 new TextWidget( 80, 24 ))));

aWidget->draw();
```

Another example could be cascading responsibilities on to an output stream -

```
Stream*  aStream = new CompressingStream(
         new ASCII7Stream(
     new FileStream( "fileName.dat" )));

aStream->putString( "Hello world" );
```

The solution to this problem 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.

Decorator

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, p149]

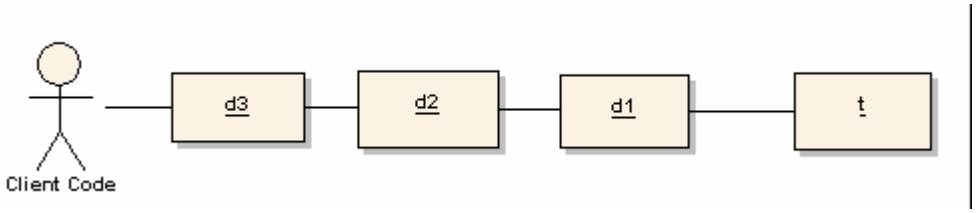
Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. [GOF, p349]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Decorator lets you change the skin of an object. Strategy lets you change the guts. [GOF, p184]



Decorator

Non Software 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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

www.dofactory.com’s structural code demonstrates the Decorator pattern which dynamically adds extra functionality to an existing object.

www.dofactory.com’s real-world code demonstrates the Decorator pattern in which 'borrowable' functionality is added to existing library items (books and videos).

Notes:

Façade

A single class that represents an entire subsystem

Definition
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Intent
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Problem
A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Discussion
Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

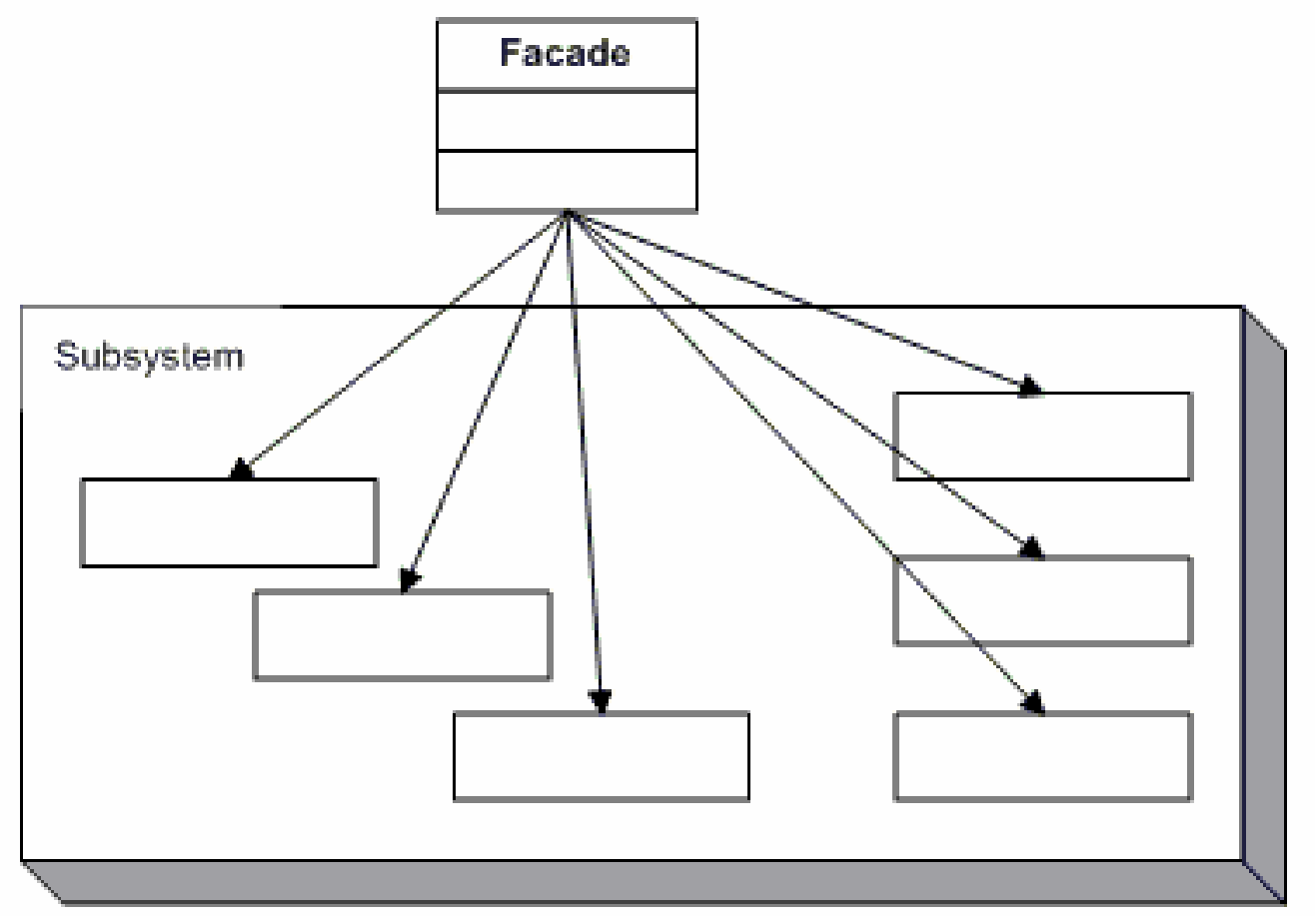
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Participants
The classes and/or objects participating in the Façade pattern are:

* Facade (MortgageApplication)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
* Subsystem classes (Bank, Credit, Loan)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.

- The system shall provide a simple interface to a complex subsystem.
- The system shall provide alternative novice, intermediate, and "power-user" interfaces.
- The system shall decouple subsystems from each other.
- The system shall support "layering" of subsystems.

Façade



Façade

Andy's Tips

- Simplify the interface to a complex system. Shield the client code.
- Façade to façade to façade – layers are possible. See GOF p186
- Vary subsystem without affecting clients. Weak coupling.
- Client code can still bypass the façade and use the subsystem directly if it wants to.
- Façade often a Singleton.
- Similar to Mediator, different intent. GOF 193.
- Negative: Facades can end up being very large API's
- The subsystems do not know about the façade!

Non-software example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

www.dofactory.com's structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

www.dofactory.com's real-world code demonstrates the Facade pattern as a MortgageApplication object which provides a simplified interface to a large subsystem of classes measuring the creditworthyness of an applicant.

Façade

Rules of thumb

Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF. p193]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

Notes:

Flyweight

A fine-grained instance used for efficient sharing

Definition
Use sharing to support large numbers of fine-grained objects efficiently.

Intent
Use sharing to support large numbers of fine-grained objects efficiently.

Problem
Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

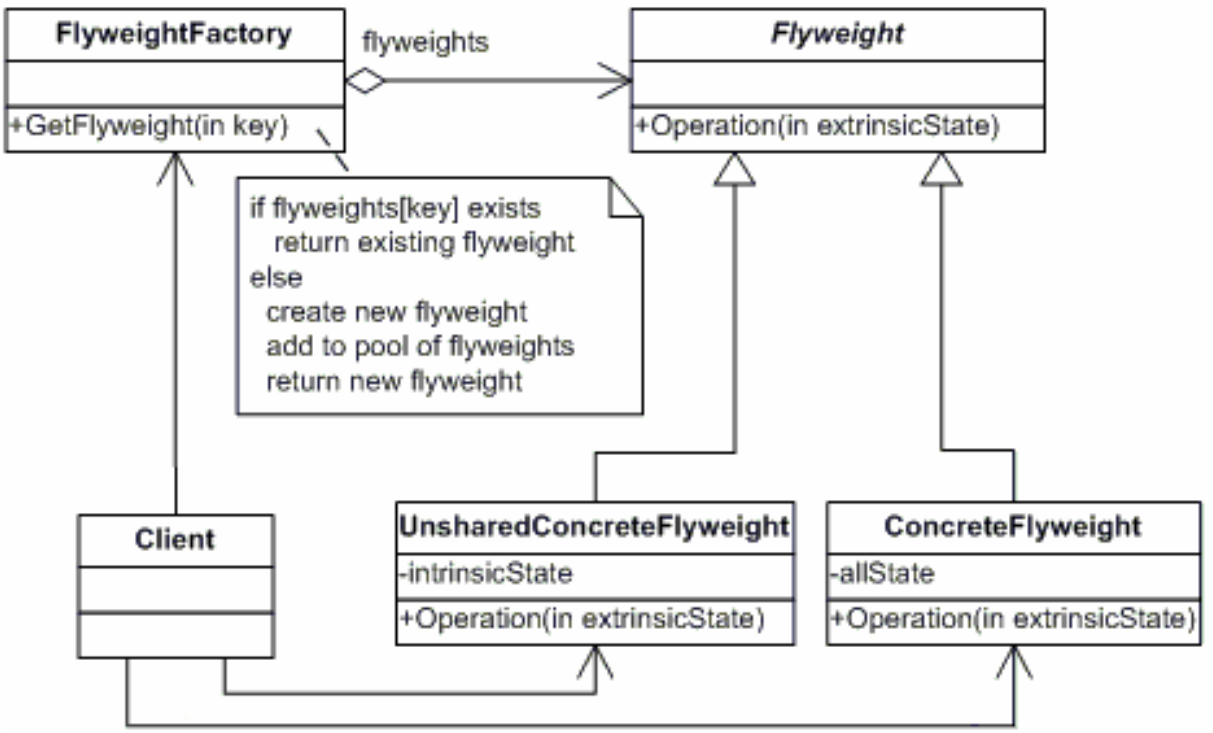
Discussion
The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

Structure
This diagram is from javacoder.net. Note that Row and Column are playing the role of Composite in the Composite pattern. The Context parameters are providing the requisite extrinsic state to each method. No "factory" is modeled here for serving up flyweight objects.

- The system shall employ hundreds of components at very fine levels of granularity without prohibitive cost.
- The system shall support pooling and reuse of a finite supply of "x" across an inexhaustible demand.

Flyweight

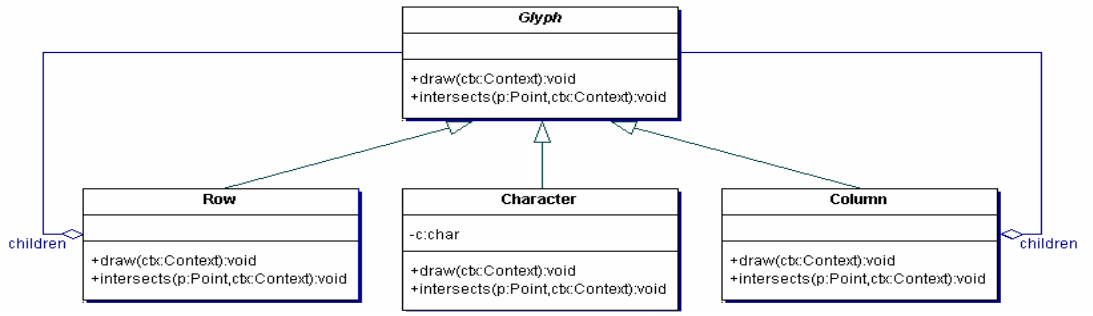

FlyweightFactory
flyweights
Flyweight
+GetFlyweight(in key)
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

Flyweight

Participants

The classes and/or objects participating in the Flyweight pattern are:

* Flyweight (Character)
  - declares an interface through which flyweights can receive and act on extrinsic state.
* ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.
* UnsharedConcreteFlyweight (not used)
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
* FlyweightFactory (CharacterFactory)
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
* Client (FlyweightApp)
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).



www.dofactory.com's structural code demonstrates the Flyweight pattern in which a relatively small number of objects is shared many times by different clients.

www.dofactory.com's real-world code demonstrates the Flyweight pattern in which a relatively small number of Character objects is shared many times by a document that has potentially many characters.

Flyweight

Andy’s Tips

- Basically you have a problem of say, millions of objects - which is expesive to store, so you refactor the common stuff out into a single object (intrinsic state) and the varying stuff into extrinsic state.
- Extrinsic state might be implemented in a very efficient manner e.g. not separate objects but an array or database or binary bits of a number etc.
- Flyweight is not a pattern you would typically use very often.

Example

The Flyweight uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF. p255]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Notes:

Proxy

An object representing another object

Definition

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

Intent

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

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.

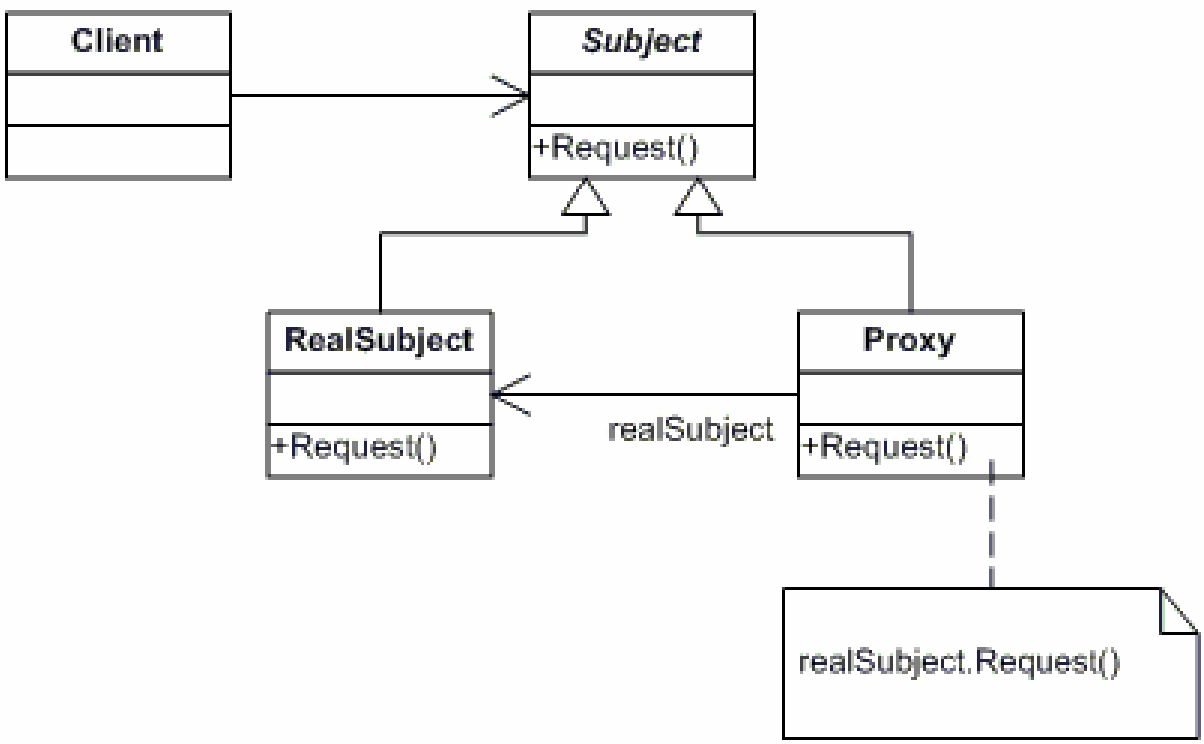
- A **virtual proxy** is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.

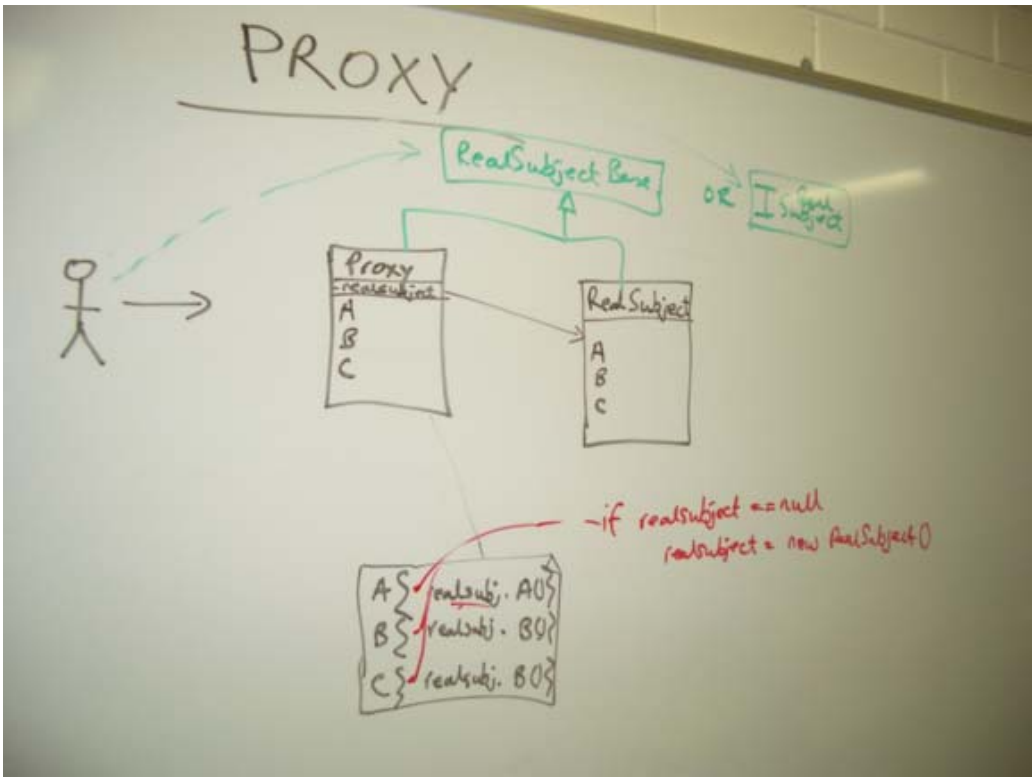
- A **remote proxy** provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.

- 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.

- A **smart proxy** interposes additional actions when an object is accessed. Typical uses include:
    1. 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),
    2. Loading a persistent object into memory when it's first referenced,
    3. Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Proxy


Client
Subject
+Request()
RealSubject
+Request()
Proxy
realSubject
+Request()
realSubject.Request()


PROXY
RealSubject Base
OR
Proxy
realSubject
A
B
C
RealSubject
A
B
C
if realSubject == null
realSubject = new RealSubject()
A
B
C

Proxy

Participants

The classes and/or objects participating in the Proxy pattern are:

* Proxy (MathProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - **provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.**
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilites depend on the kind of proxy:
    + remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    + virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
    + protection proxies check that the caller has the access permissions required to perform a request.
* Subject (IMath)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
* RealSubject (Math)
  - defines the real object that the proxy represents.

- The system shall support "distributed processing" by providing a local representative for a component in a different address space.
- The system shall support "lazy creation" - a component is created only if, and when, the client demonstrates an interest in it.
- The system shall control access to components by interposing an intermediary that evaluates the identity and access privileges of the requestor.
- The system architecture shall be characterized by multiple dimensions of indirection that offer a locus for intelligence that would unduly complicate other components if they were obliged to support the additional responsibility.

Proxy

Andy's Tips

- Basic indirection pattern—client code talks to the proxy instead of the real thing.
- Both the proxy and the real thing either descend off the same base class or implement the same interface, so that the client can use either the proxy or the real thing without knowing. Of course the client code should only talk to the interface or the base class.
- The proxy can do a lazy instantiation of the real subject, if the real subject doesn't exist—which is a nice use of a proxy. E.g. Proxy Class:

```
Proxy.SomeMethodAA() {

  // Lazy instantiation
  if this.realSubject == null
   this.realSubject == new realSubject()

  // Now do the proxy delegation work
  this.realSubject.SomeMethodAA();
}
```

www.dofactory.com's structural code demonstrates the Proxy pattern which provides a representative object (proxy) that controls access to another similar object. www.dofactory.com's real-world code demonstrates the Remote Proxy pattern which provides a representative object that controls access to another object in a different AppDomain.

Non Software Example

The Proxy provides a surrogate or place holder to provide access to an object. Asking technical questions of a Sales Representative. In all likelihood, the Sales Representative will forward the technical request to a Technical Representative and relay the answer to the customer.

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations

Notes:

Chain of Responsibility

A way of passing a request between a chain of objects

Definition

Avoid coupling the sender of a request to its receiver by 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.

Intent

Avoid coupling the sender of a request to its receiver by 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.

Problem

There is a potentially variable number of "handler" objects and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Discussion

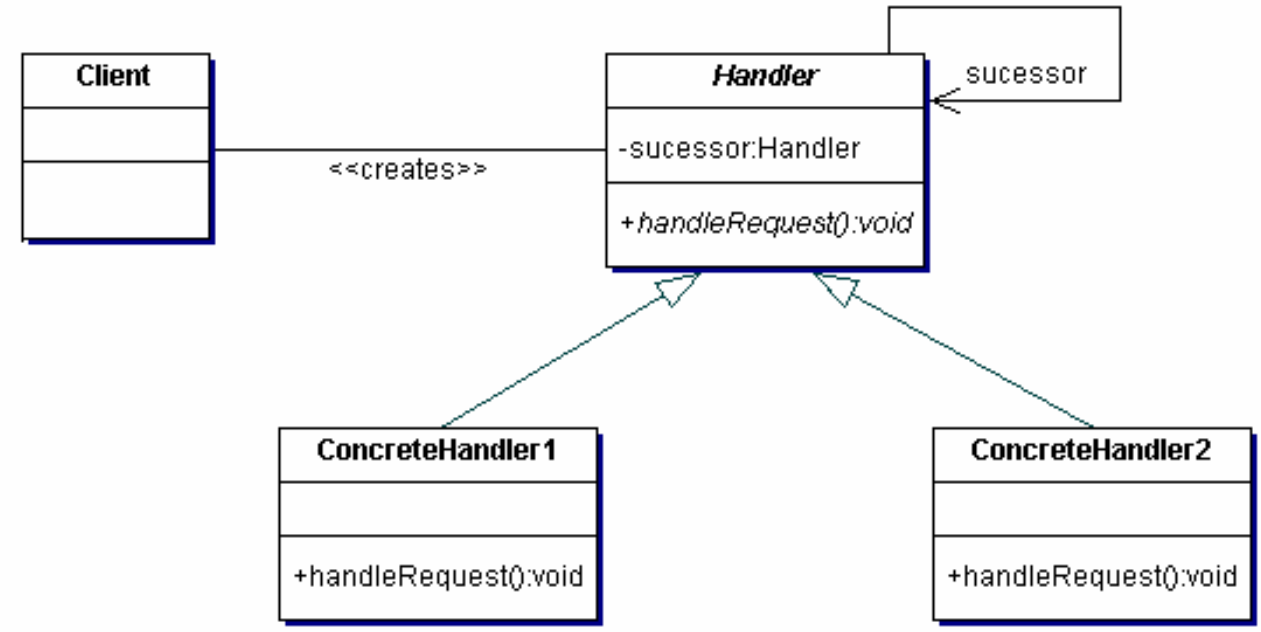
The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a "safety net" to "catch" any requests which go unhandled.

Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Chain of Responsibility



Chain of Responsibility

www.dofactory.com's structural code demonstrates the Chain of Responsibility pattern in which several linked objects (the Chain) are offered the opportunity to respond to a request or hand it off to the object next in line.
www.dofactory.com's real-world code demonstrates the Chain of Responsibility pattern in which several linked managers and executives can respond to a purchase request or hand it off to a superior. Each position has can have its own set of rules which orders they can approve.

Non Software Examples
The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the Chain of Responsibility. Rather than having a separate slot for each coin denomination coupled with a receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Non Software example: The chain of command in the military illustrates the *Chain of Responsibility* pattern. When a request is made of a private, the private can either grant the request or forward the request to his or her superior. The request will be forwarded until someone with the appropriate authority is able to act upon it.

Rules of thumb
Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349]

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor. [GOF, p232]

- The system shall allow the client to issue a request to one of several "handlers" without knowing or specifying the receiver explicitly.
- The system shall allow a suite of "handlers" to be configured at run-time.
- The system shall allow the client to "launch and leave" a request with a "virtual pipeline of handlers".
- "Senders" and "receivers" shall be decoupled from one another.

Chain of Responsibility

Participants
The classes and/or objects participating in this pattern are:

* Handler (Approver)
  - defines an interface for handling the requests
  - (optional) implements the successor link
* ConcreteHandler (Director, VicePresident, President)
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
* Client (ChainApp)
  - initiates the request to a ConcreteHandler object on the chain

Andy's Tips

- If you find your client code is making "probing calls" before issuing the request it really wants to make, apply the chain of responsibility pattern.
- You relieve the client from having to know which object in a collection supports a given behavior.
- Can be implemented as a chain (common) or a hierarchy where messages / requests trickle up from the leaves to the parents to the grandparents etc. E.g. Toolbook's hierarchical message passing, Java 1.0 (user interface event container hierarchy).
- Consequence: reduce the coupling between classes. Objects just need to know only how to handle a request, or alternatively, forward the request to other objects.
- Requests can be forwarded by each individual object in the chain, or alternatively there can be an external loop which calls each object in the chain. External control probably better (more decoupled and less fragile)
- To wire up your chains/hierarchies? Constructors or an AddToChain(o) method. Perhaps use an existing hierarchy or chain/arraylist?

Notes:

Command

Encapsulate a command request as an object. Implement Undo/Redo.

Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

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.

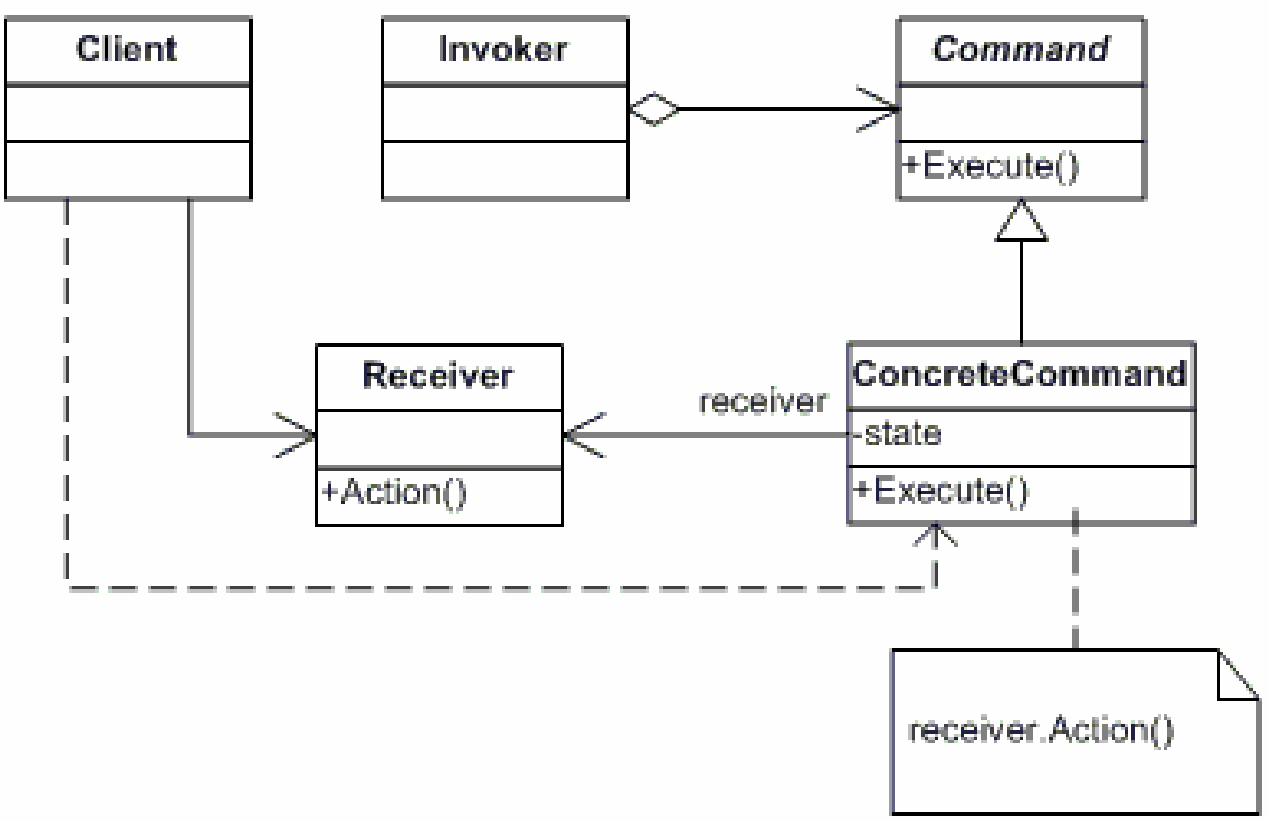
All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual Execute() method whenever the client requires the object's "service".

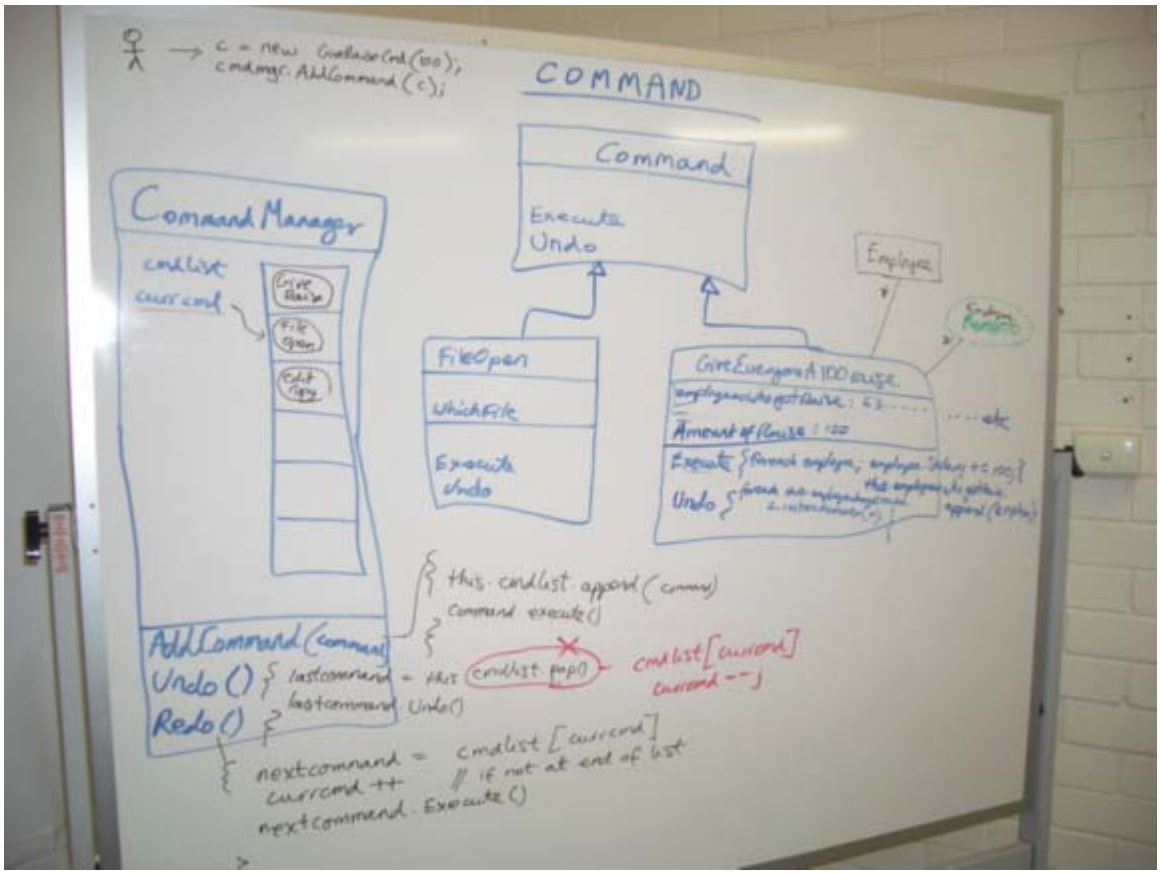
Sequences of Command objects can be assembled into composite (or macro) commands.

Non Software 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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Command


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()


COMMAND
Command
Execute
Undo
Comand Manager
cmdlist
currcmd
FileOpen
WhichFile
Execute
Undo
Employee
AddCommand(command)
Undo()
Redo()
this.cmdlist.append(cmmd)
Command.execute()
cmdlist.pop()
cmdlist[currcmd]
currcmd--
lastcommand = this
lastcommand.Undo()
nextcommand = cmdlist[currcmd]
currcmd++ // if not at end of list
nextcommand.Execute()

Command

www.dofactory.com’s structural code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.
www.dofactory.com’s real-world code – demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C# the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

Rules of thumb
Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349].

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, p242]

MacroCommands can be implemented with Composite. [GOF, p242]

A Command that must be copied before being placed on a history list acts as a Prototype. [GOF, p242]

POSA's Command Processor pattern describes the scaffolding Command needs to support full multilevel undo and redo. [Vlissides, Java Report, Nov 2000, p80]

- The system architecture shall provide a "callback" framework.
- The system shall encapsulate "execute" requests so that they may be created, queued, and subsequently serviced. They may also be logged, archived, loaded, and re-applied.
- The system shall support hierarchical compositions of primitive "command" abstractions.
- The system shall support reuse of "command" abstractions. [Define a "command", and simultaneously attach the encapsulation to a push button, a toolbar icon, a menu item, and a keyboard accelerator - so that all users of the GUI can enjoy their preferred method of interaction.]
- The system shall support "redo"
- The system shall support transactions
- "Senders" and "receivers" shall be decoupled from one another.

Command

Participants
The classes and/or objects participating in this pattern are:

* Command (Command)
  - declares an interface for executing an operation
* ConcreteCommand (CalculatorCommand)
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
* Client (CommandApp)
  - creates a ConcreteCommand object and sets its receiver
* Invoker (User)
  - asks the command to carry out the request
* Receiver (Calculator)
  knows how to perform the operations associated with carrying out the request.

Andy's Tips

- Commands are object oriented replacements for callbacks (a callback is a function that's registered somewhere to be called at a later time).
- A memento can keep the state the command requires to undo its effect.
- To undo, you can either take a snapshot of the way the system was and restore it, or alternatively, just undo the steps the command did. Depending on your situation and how much info there is in the "state", it might be easier one way or the other.

Notes:

Interpreter

A way to include language elements in a program

Definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

Discussion

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

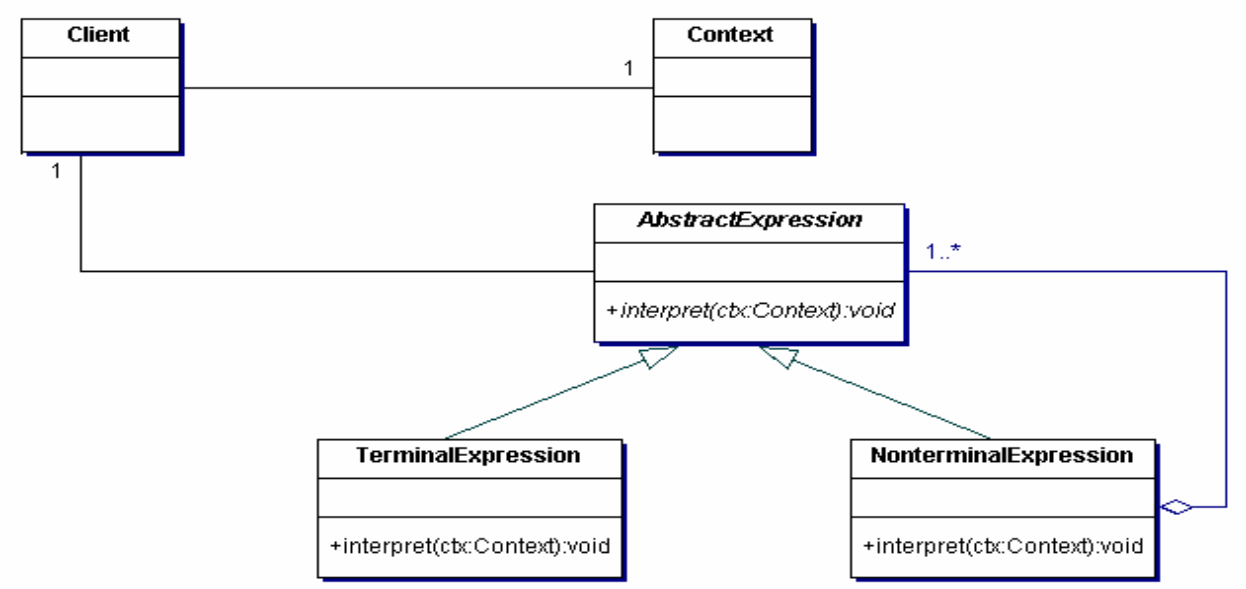
An abstract base class specifies the method interpret(). Each concrete subclass implements interpret() by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

Non Software Example

The Intepreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

> The system shall characterize the domain as a set of rules, then define a language capable of specifying the rules, and a grammar for defining the language, and an "engine" for interpreting the grammar.

Interpreter



Interpreter

Participants
The classes and/or objects participating in this pattern are:

* AbstractExpression (Expression)
  - declares an interface for executing an operation
* TerminalExpression (ThousandExpression,
  HundredExpression,
  TenExpression,
  OneExpression)
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.
* NonterminalExpression (not used)
  - one such class is required for every rule R ::= R1R2...Rn in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
* Context (Context)
  - contains information that is global to the interpreter
* Client (InterpreterApp)
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
    o invokes the Interpret operation

Code Examples
www.dofactory.com's code demonstrates the Interpreter patterns, which using a defined grammer, provides the interpreter that processes parsed statements.

www.dofactory.com's real-world code demonstrates the Interpreter pattern which is used to convert a Roman numeral to a decimal.

Interpreter

Rules of thumb
Interpreter can use State to define parsing contexts. [GOF, p349]

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF, p255]

Notes:

Iterator

Sequentially access the elements of a collection

Intent

Provide a way to access the elements of an aggregate object (“collection”) sequentially—without exposing the details of the underlying internal representation of the collection to the looping code.

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each consistently and transparently. E.g. one class may use an **Array**, another might use an **ArrayList**—thus the way client code would be force to write looping code would be different in each case. It would be nice to have a consistent interface for looping through collections—a **standard traversal protocol**.

Discussion

"An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." [GOF, p257] And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

Participants

The classes and/or objects participating in this pattern are:

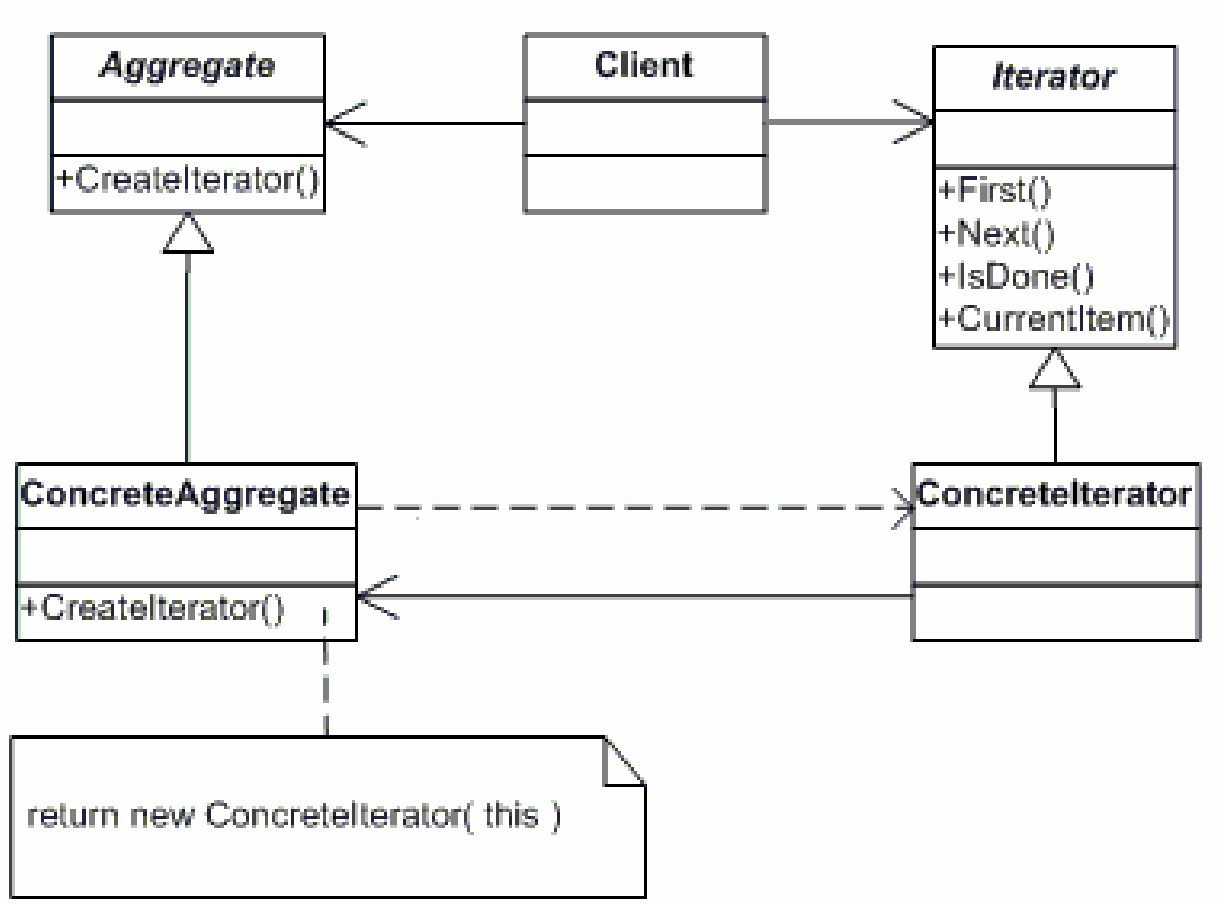
* Iterator (AbstractIterator)
  - defines an interface for accessing and traversing elements.
* ConcreteIterator (Iterator)
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
* Aggregate (AbstractCollection)
  - defines an interface for creating an Iterator object
* ConcreteAggregate (Collection)
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Code Examples

www.dofactory.com’s structural code demonstrates the Iterator pattern which provides for a way to traverse (iterate) over a collection of items without detailing the underlying structure of the collection.

www.dofactory.com’s real-world code demonstrates the Iterator pattern which is used to iterate over a collection of items and skip a specific number of items each iteration.

Iterator


Aggregate
+CreateIterator()
Client
Iterator
+First()
+Next()
+IsDone()
+CurrentItem()
ConcreteAggregate
+CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator

Andy's Tips

- Iterators allow you to have multiple traversals happening at the same time because the "current position" is kept in the iterator object, and you can have as many iterator objects as you like.
- The Iterator class defines the interface for accessing and traversing elements. It could use memento to capture the state of the iteration, but usually a simple property "CurrentItem" will do.
- .NET, Java and Python and other languages have built-in support for iteration through the use of enumeration interfaces e.g. IEnumerable and built in language keyword support e.g. foreach().
- You can have different styles of iterator, iterating in different ways e.g. sequentially, depth first, breadth first etc.
- You can define special **"filtering" iterators** that skip certain items in your collection. This might be a cool way to organize your business logic e.g. iterate over the female customers. Perhaps chain multiple iterators—like a google search within a google search (Decorator?)
- The Aggregate (or collection e.g. ArrayList) should have a consistently named factory method called something like "CreateIterator()" which returns an iterator object—so that you always know where to get the iterator from, and also so that the built in language support (e.g. foreach) also knows how to get the iterator. The Aggregate's factory method e.g. "CreateIterator()" is typically part of an official interface e.g. IEnumerable
- **Internal** vs. **External** iterators: An external iterator is where the client code does the looping, albeit via a consistent api. With an internal iterator is you *pass a method* to the collection (e.g. a C# delegate, or Java object with a known method to call each time, or block of code known as a "closure") and have the collection do the iteration internally, calling the code/method/delegate with each item in the collection as a parameter. Thus the iterator is never exposed to the client.
- Traversing trees via recursion is more easily done as an *internal* iterator because you can take advantage of recursive programming and a call stack. You simply call a user method within the recursive algorithm. Doing recursive iteration using an *external* iterator is impossible (unless your language has a 'yield' keyword) because e.g. MoveNext() returns after each item and the call stack is lost, meaning you have to traverse the tree in a more laborious, non recursive way e.g. keeping track of pointers to children and parents etc.

Iterator

Rules of thumb
The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Non Software Examples

Iterator examples included the receptionist in a doctor's waiting room. The receptionist iterates the aggregate of patients who are seated randomly around the room. The receptionist will send the "next" patient to the doctor.

The channel selector on modern day television sets is another example of an *Iterator.* Rather than a dial containing all possible channels or one button per tuned channel, today's television sets have a "Next" and "Previous" button for tuning channels. The "Channel Surfer" doesn't need to know if Channel 3 or Channel 4 comes after Channel 2.

These non software examples demonstrate the intent of the *Iterator*, since the *Iterator* knows how the objects are ordered.

- The system shall support accessing an aggregate component's contents without exposing its internal representation.
- The system shall support multiple simultaneous traversals of aggregate components without complicating the implementation of the aggregate itself.
- The system shall define a uniform interface for traversing dissimilar aggregate components.
- The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.

Notes:

Mediator

Defines simplified communication between classes

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

Discussion

Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.

Example

An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.

Participants

The classes and/or objects participating in this pattern are:

* Mediator (IChatroom)
  - defines an interface for communicating with Colleague objects
* ConcreteMediator (Chatroom)
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues
* Colleague classes (Participant)
  - each Colleague class knows its Mediator object
    each colleague **communicates with its mediator whenever it would have otherwise communicated with another colleague.**

Mediator - Structure

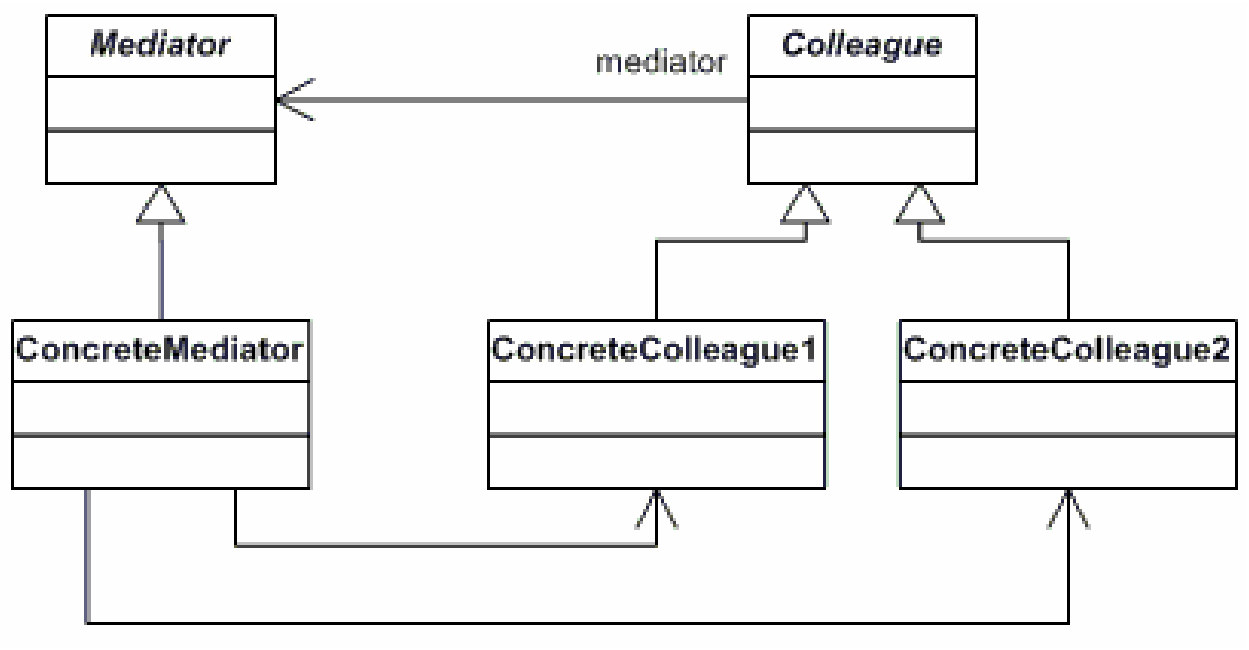
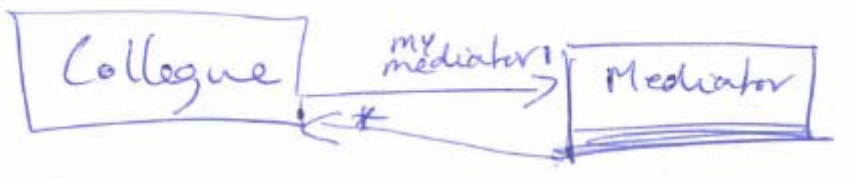


diagram is from javacoder.net.



Andy's simplified UML

Mediator

Code Examples

www.dofactory.com’s structural code demonstrates the Mediator pattern facilitating loosely coupled communication between different objects and object types. The mediator is a central hub through which all interaction must take place.

www.dofactory.com’s real-world code demonstrates the Mediator pattern facilitating loosely coupled communication between different Participants registering with a Chatroom. The Chatroom is the central hub through which all communication takes place. At this point only one-to-one communication is implemented in the Chatroom, but would be trivial to change to one-to-many.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

Non Software Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Mediator

Andy's Tips

- Define an object that encapsulates how a set of objects interact.
- Promote loose coupling by keeping objects from referring to each other explicitly. "Decouples colleagues".
- Isolate complex inter-relationships in the one mediating class. Centralizes control. Lets you vary their interaction independently.
- Each Colleague class knows its Mediator. Each Colleague communicates with its mediator **whenever it would have otherwise communicated with another colleague**.
- Colleagues send and receive requests from a mediator object and the mediator routes requests between the appropriate colleagues.
- Can implement Mediator using the Observer pattern. Colleagues are subjects and Mediator is the observer. E.g. Colleagues/subjects notify the mediator when they change state. The Mediator responds by propogating the effects of the change to the other colleagues.
- A centralized message kernel is a mediator.
- A GUI form (e.g. Swing form, Winform, Delphi TForm etc) could be though of as a mediator since GUI components ("colleagues") on the form don't know about each other (they are, after all, standard components). However their event handlers are housed inside the one form, thus the form (incl. event handler code) is a mediator. E.g. a button click causes text to appear in a label—the mediator makes this happen without the button GUI component knowing anything about the label GUI component.

- The system shall encapsulate complex, many-to-many coupling between "peer" components in a separate component capable of allowing the "peers" to be: disengaged, replaced, and reused.
- The system shall support numerous many-to-many "mappings" to be defined, installed, and exchanged by the client.
- The system shall balance the distribution of intelligence emphasized by "logical" OO design with the centralization of intelligence often required by "physical" large scale design.
- "Senders" and "receivers" shall be decoupled from one another.

Notes:

Memento

Capture and restore an object's internal state

Definition
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Intent
Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

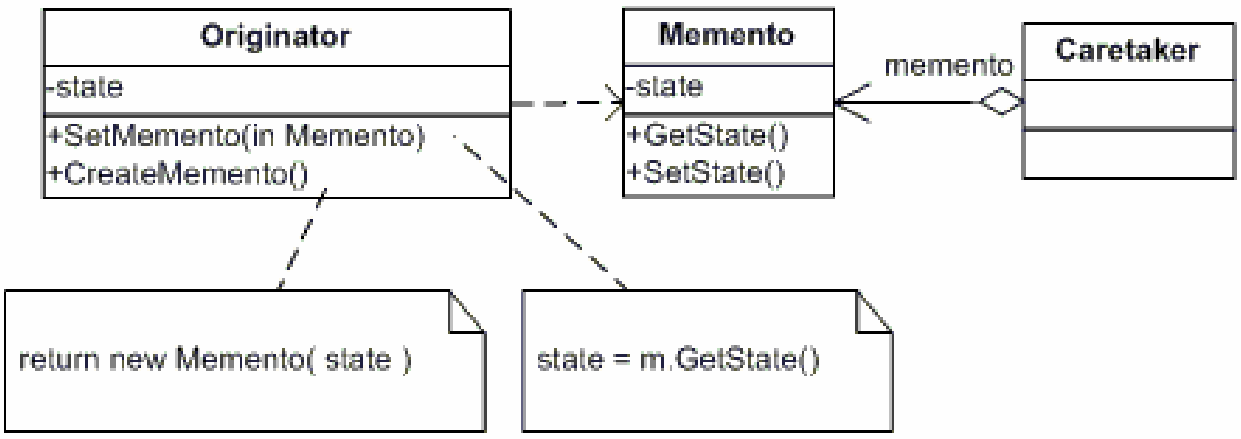
Problem
Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Discussion
The client requests a Memento from the source object when it needs to check-point the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

- The system shall support "undo" or "rollback"
- The system shall support transactions
- The system shall support saving, or "flattening", or "streaming" components without compromising their encapsulation.

Memento


Originator
-state
+SetMemento(in Memento)
+CreateMemento()
Memento
-state
+GetState()
+SetState()
memento
Caretaker
return new Memento(state)
state = m.GetState()

Memento

Participants
The classes and/or objects participating in this pattern are:

* Memento (Memento)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
* Originator (SalesProspect)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
* Caretaker (Caretaker)
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

Code Examples
www.dofactory.com’s structural code demonstrates the Memento pattern which temporary saves and restores another object's internal state.

www.dofactory.com’s real-world code demonstrates the Memento pattern which temporarily saves and then restores the SalesProspect's internal state.

Rules of thumb
Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, 242]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Memento

Non Software Examples

An example of audio mixing equipment was then proposed. Since there are several switches set in different positions, it is common to photograph the equipment, so that the switches can be returned to their original positions if perturbed.

Yet another example was proposed at the poster session. When a restaurant patron enters an establishment wearing a coat, it is common for the coat to be left in the coat room while the patron dines. The patron is given a claim check, so that upon leaving the restaurant, he or she is returned to the same state: HAS_COAT. In this example, the claim check is used to ensure that the coat the patron has upon leaving the restaurant is the same coat that he or she had upon entering.

Andy's Tips

- Capture an object's state and restore it later, without violating encapsulation
- The Originator (the class who's state you are capturing) sees a wide interface to the memento (the class containing the state). The Caretaker (or whoever else holds a reference to the memento object) sees a narrow interface.
- The Caretaker never operates or examines the contents of the memento (hence the narrow interface)

Notes:

Observer

A way of notifying change to a number of classes

Definition

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

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [GOF p. 297]

Problem

You want to achieve decoupled communication between classes, without breaking layering or encapsulation. You want to be able to notify an arbitrary and dynamically changing list of objects that something has happened.

Discussion

Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

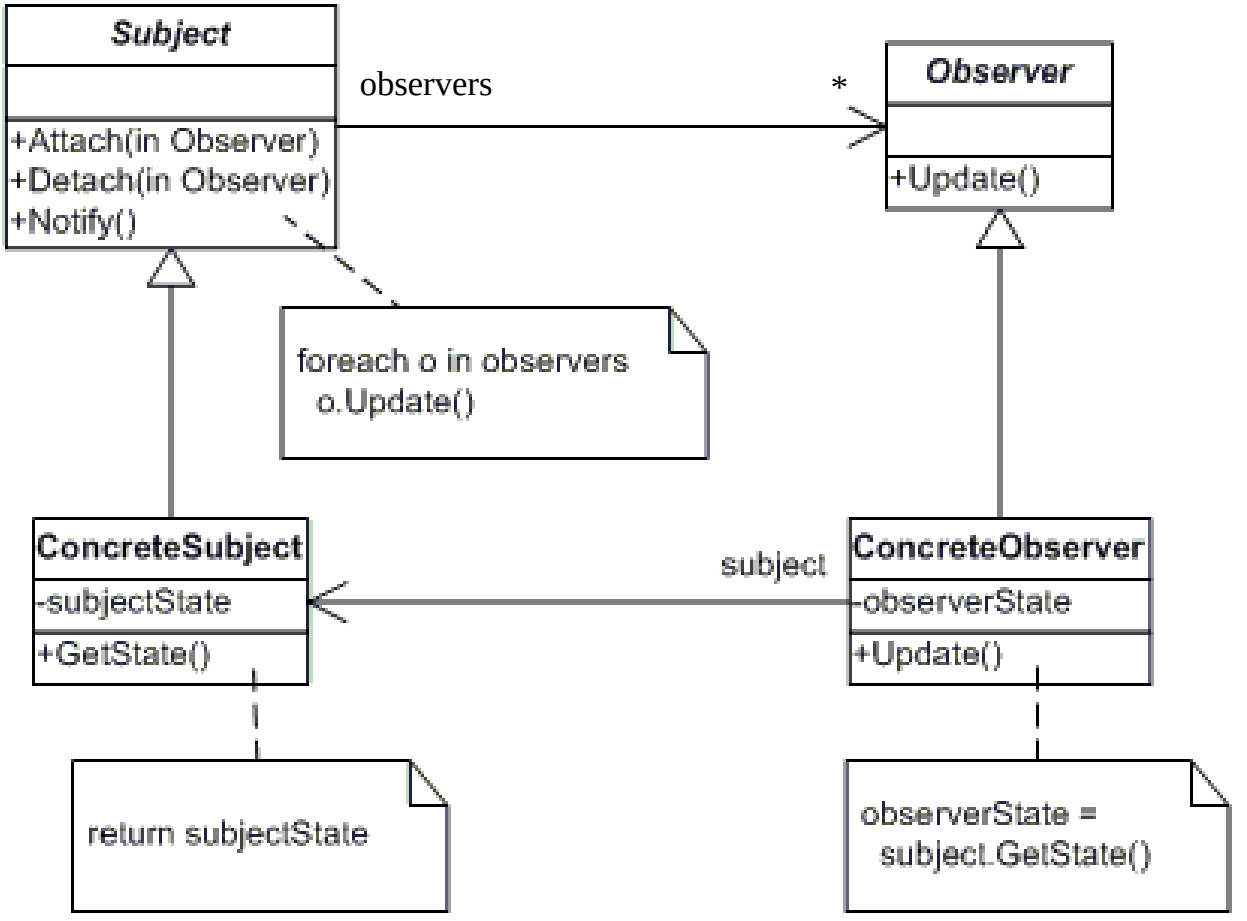
This allows the number and "type" of "view" Observer objects to be configured dynamically, instead of being statically specified at compile-time.

Participants

The classes and/or objects participating in this pattern are:

* Subject (Stock)
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
* ConcreteSubject (IBM)
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
* Observer (IInvestor)
  - defines an updating interface for objects that should be notified of changes in a subject.
* ConcreteObserver (Investor)
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's state
  - implements the Observer updating interface to keep its state consistent with the subject's state.

Observer


Subject
+Attach(in Observer)
+Detach(in Observer)
+Notify()
observers
*
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Observer

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

You can have a “push” or “pull” designs. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Non Software Examples

Consumers (observers) who send in warranty registration cards to a company (subject) will be notified in the event of a recall, new version release or any other news related to the product. This example uses explicit registration, and the collaborations may be easier to see.

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

The CNN news agency was presented as an example of the *Observer* pattern. The news agency acts as the subject, and the viewing public are the observers. When something in the world changes, everyone is notified. The discussion of this example centered on whether or not the subject knows the observers and whether or not registration is essential to the pattern. Turning on a television set could be considered to be registering. This broadcast method of distributed communication is common in data communications.

Code Samples

www.dofactory.com’s structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

www.dofactory.com’s real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

Observer

Andy's Tips

- The subject's Notify() method simply loops through the list of observers and calls each observer's Update() method. **This is the only knowledge the subject has about the observer**. The observer, on the other hand, often knows all about the subject it is observing, including knowledge of how to extract specific data from the subject.
- The subject's Notify() method can be implemented in the base "Subject" class (as can the arraylist holding the list of observers). Thus the mechanism for being a Subject need only be written once.
- Observer is an abstract coupling—the subject doesn't know the concrete class of the observer, thus notifications can be safely done between different layers of the system e.g. between model and GUI.
- .Net has the multicast delegate mechanism for doing observer. Java has the listener pattern for doing observer. Alternatively it is reasonably easy to build your own observer implementation if you want a specific type of observer-like behaviour.
- Watch out for dangling references—there should be a notification between the subject and observer when subject or observer is deleted.
- There can often be unnecessary notifications to the observers—the naïve implementation of Observer is blind to the cost of updates/notification. A solution is to build a mechanism to switch on/off updates (only sending a single change broadcast after a series of consecutive changes has occurred).
- You can achieve the intent of the observer design pattern (of decoupled communication) by using an event. The subject defines an event and fires it on occasion. The observer installs a method in that event and thus gets notified whenever the event fires. A multicast delegate allows multiple methods to be installed / setup on an event, and is the same as the observer pattern. A non multicast event would be a weaker form of the observer pattern, since you could only have one observer at a time.
- Implementation variant: having a single Observer monitoring multiple Subjects
- Observer pattern typically used in hooking GUI to model. Model objects are the subject, GUI components (or their controllers) are the Observers.

Notes:

State

Alter an object's behavior when its state changes

Definition
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Intent
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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.

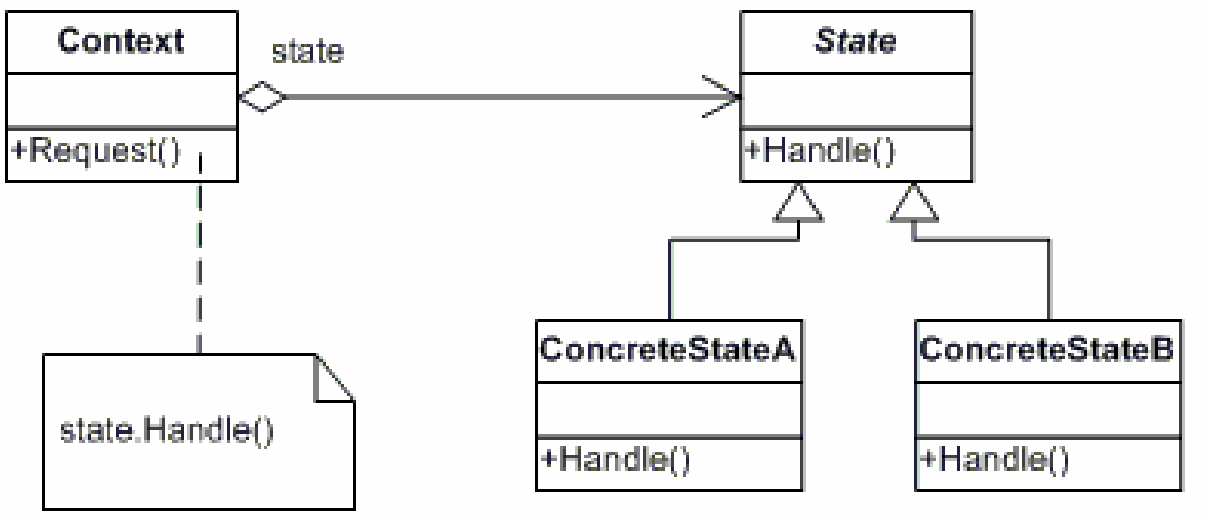
Discussion
The State pattern is a solution to the problem of how to make behavior depend on state.

* 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. [GOF, p308]

- Components shall be capable of "morphing" their behavior at run-time.
- The system architecture shall be characterized by a "finite state machine". The state machine will need to support application logic at each state transition, not simply the transition itself. [A "table-driven" approach routinely supports only the latter.]
- "case" statements are a maintenance nightmare - they shall be avoided.

State


Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

State

More Discussion
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. [GOF, p308]

The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects. [Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 1999, p253]

Participants
The classes and/or objects participating in this pattern are:

* Context (Account)
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
* State (State)
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
* Concrete State (RedState, SilverState, GoldState)
  - each subclass implements a behavior associated with a state of Context

Code Examples
www.dofactory.com's structural code demonstrates the State pattern which allows an object to behave differently depending on its internal state. The difference in behavior is delegated to objects that represent this state.

www.dofactory.com's real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

State

Non Software 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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb
State objects are often Singletons. [GOF, p313]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Interpreter can use State to define parsing contexts. [GOF, p349]

State is like Strategy except in its intent. [Coplien, C++ Report, Mar 96, p88]

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, C++ Report, Mar 96, p88]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58; see discussion under Bridge pattern]. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, C++ Report, May 95, p58]

Notes:

Strategy

Encapsulates an algorithm inside a class

Definition

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

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.

Problem

If clients have potentially generic algorithms embedded in them, it is difficult to: reuse these algorithms, exchange algorithms, decouple different layers of functionality, and vary your choice of policy at run-time. These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions.

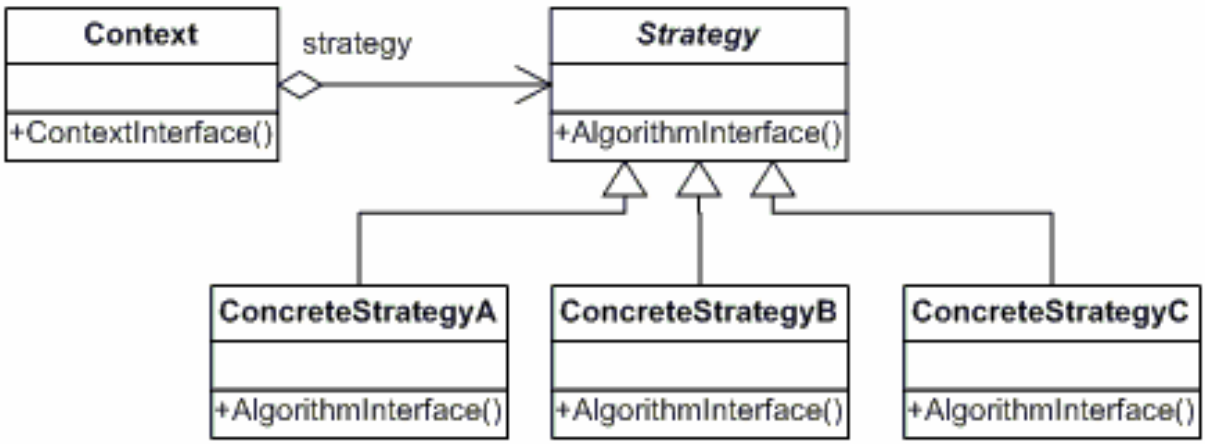
Discussion

Replace the many, monolithic, conditional constructs with a Strategy inheritance hierarchy and dynamic binding.

* Identify the protocol that provides the appropriate level of abstraction, control, and interchangeability for the client.
* Specify this protocol in an abstract base class.
* Move all related conditional code into their own concrete derived classes.
* Configure the original application with an instance of the Strategy hierarchy, and delegate to that "contained" object whenever the "algorithm" is required.

- Clients shall be decoupled from "choice of algorithm".
- Clients shall be decoupled from complex, algorithm-specific data structures.
- The choice, or implementation, of algorithm shall be configurable at run-time.
- "case" statements are a maintenance nightmare - they shall be avoided.

Strategy


Context
strategy
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Strategy

More Discussion

"Strategies can provide different implementations of the same behavior. The client can choose among Strategies with different time and space trade-offs." [GOF, p318] This sounds exactly like the intent of the Bridge pattern. The apparent overlap can be explained by observing that both patterns employ the same composition-delegation structure. The distinction is in intent: Strategy is a "behavioral" pattern (the focus is on delegating responsibilities), Bridge is a "structural" pattern (the focus is on establishing relationships).

The Strategy pattern should only be used when the variation in behavior is relevant to clients. If this criteria is not satisfied, the additional abstraction and effort necessary to implement the Strategy pattern is not justified.

There are two implementations for Strategy. The first is described above. It uses abstract coupling and composition-delegation. The second uses C++ templates. It is demonstrated in the second example that follows, and requires the client to "bind" the choice of algorithm at compile time.

Non Software 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 tlme. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Code Examples

www.dofactory.com’s structural code demonstrates the Strategy pattern which encapsulates functionality in the form of an object. This allows clients to dynamically change algorithmic strategies.

www.dofactory.com’s real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

Strategy

Participants
The classes and/or objects participating in this pattern are:

* Strategy (SortStrategy)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
* ConcreteStrategy (QuickSort, ShellSort, MergeSort)
  - implements the algorithm using the Strategy interface
* Context (SortedList)
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Rules of thumb
State is like Strategy except in its intent. [Coplien, Mar96, p88]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58]. They differ in intent - that is, they solve different problems.

Strategy lets you change the guts of an object. Decorator lets you change the skin. [GOF, p184]

Strategy is to algorithms as Builder is to creation. [Icon, p8-13]

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, Mar96, p88]

Strategy objects often make good Flyweights. [GOF, p323]

Strategy is like Template Method except in its granularity. [Coplien, Mar96, p88]

Notes:

Template Method

Defer the exact steps of an algorithm to a subclass

Definition

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.

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.

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.

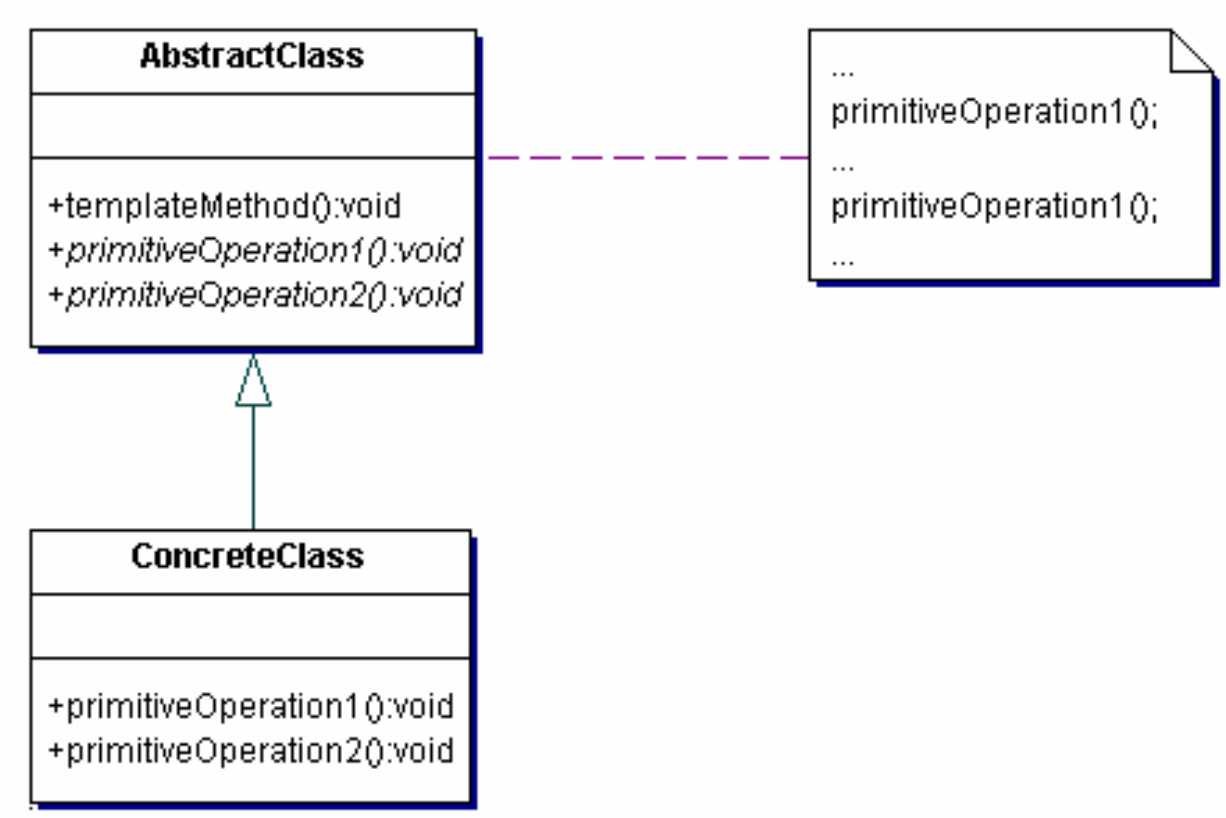
Discussion

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".

Template Method



Template Method

Participants
The classes and/or objects participating in this pattern are:

* AbstractClass (DataObject)
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
* ConcreteClass (CustomerDataObject)
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm

Code Examples
www.dofactory.com's structural code demonstrates the Template method which provides a skeleton calling sequence of methods. One or more steps can be deferred to subclasses which implement these steps without changing the overall calling sequence.

www.dofactory.com's real-world code demonstrates a Template method named Run() which provides a skeleton calling sequence of methods. Implementation of these steps are deferred to the CustomerDataObject subclass which implements the Connect, Select, Process, and Disconnect methods.

Non Software 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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb
Strategy is like Template Method except in its granularity. [Coplien, C++ Report, Mar 96, p88]

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]

Template Method

- Individual steps of an algorithm shall be definable or replaceable.
- The system shall provide "hooks" wherever future functionality or extensibility may be required.
- The "invariant" parts of an algorithm shall be implemented in one place, and reused all other places.
- The "standard" parts of an algorithm shall be implemented and enforced in one place.
- Reuse shall be emphasised by architecting a "don't call us, we'll call you" framework.
- The system shall be "open for extension, but closed for modification".

Notes:

Visitor

Defines a new operation to a class without change

Definition

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.

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.

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

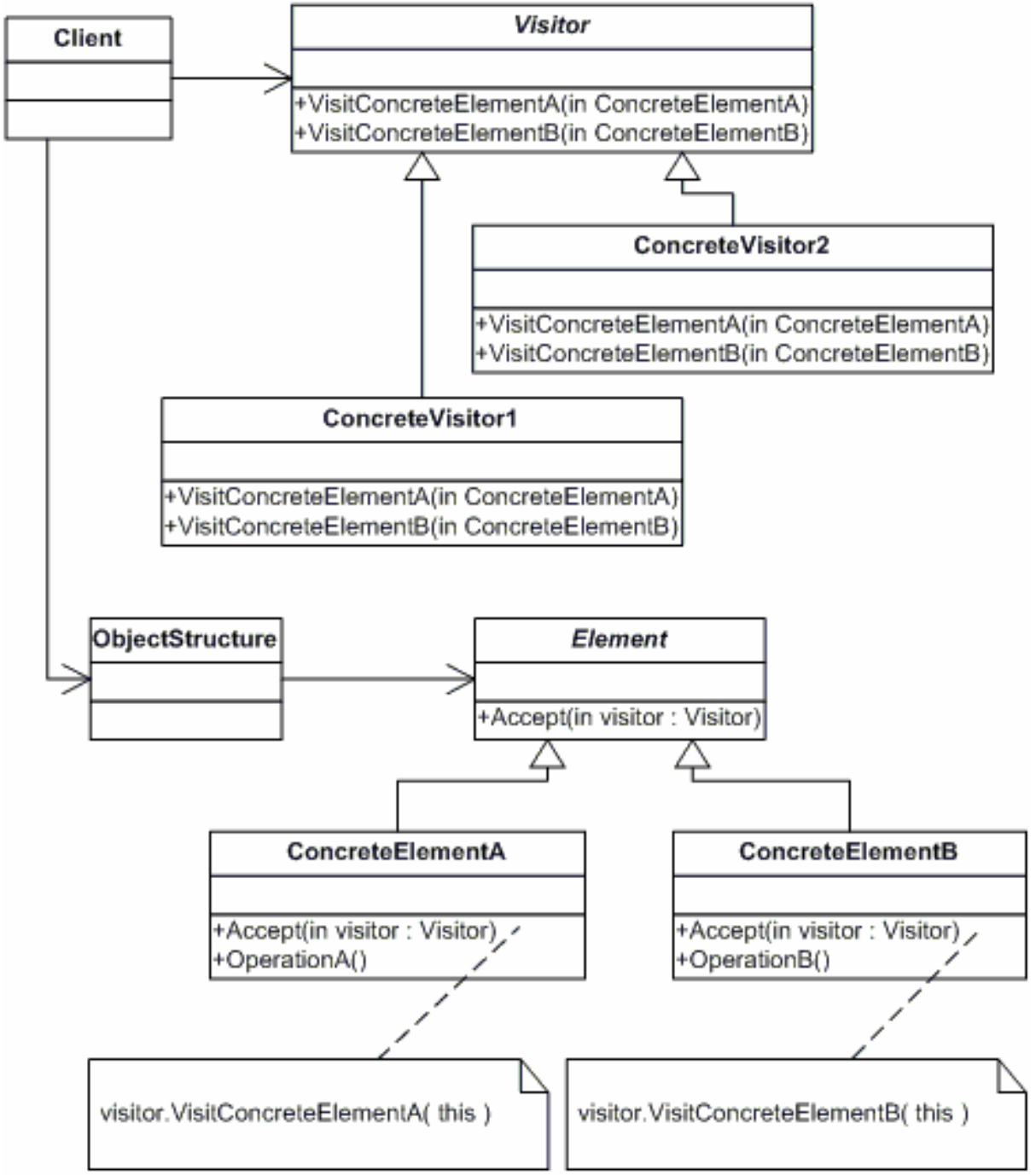
Discussion

Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.

Visitor


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

Visitor

More Discussion

Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.

Everything for "elements" and "visitors" is now set-up. When the client needs an operation to be performed, (s)he creates an instance of the Vistor object, calls the accept() method on each Element object, and passes the Visitor object. The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch.

The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.

An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/ rationale is the goal of promoting non-traditional behavior to full object status.
Structure
This diagram is from javacoder.net. The NodeVisitor hierarchy is not modeled here. The first dispatch is accept(), and the second dispatch is visit(). Inside each visit() method, we now know the exact type of both the Node entity (the object being passed) and the NodeVisitor entity (the object being messaged).

Visitor

Participants
The classes and/or objects participating in this pattern are:

* Visitor (Visitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
* ConcreteVisitor (IncomeVisitor, VacationVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
* Element (Element)
  - defines an Accept operation that takes a visitor as an argument.
* ConcreteElement (Employee)
  - implements an Accept operation that takes a visitor as an argument
* ObjectStructure (Employees)
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

Code Examples
www.dofactory.com's structural code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

www.dofactory.com's real-world code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

Andy's Tips

- Visitor is basically a way of getting different code called (using double dispatch), depending on the item you are "visiting". It haves you having big if-then-else statements.
- Visitor uses iterator (either internal or external) to drive the visitor across the items of the collection

Visitor

Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters. [Johnson, Huni, Engel, 1995, p8]

The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts. [Vlissides, "Type Laundering", C++ Report, Feb 97, p48]

Non Software Example

The *Visitor* example involves an outside consultant coming into a department to meet with employees. While the visitor is escorted to each cubicle, she does nothing. Once she arrives at a cubicle, she introduces herself, and the employee introduces himself (double dispatch). Following the introduction, the consultant springs into action interviewing the employee (sending messages and obtaining results). The basic idea is that the consultant is not required to have knowledge of the organization's structure to do her job. The escort supplies this knowledge.

- The system shall allow the operations that can be performed on "whole-part" hierarchical assemblies of components to be defined and added without having to change (and potentially break) any existing code.
- The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.
- The design shall support the recovery of "lost type information" without resorting to the overhead associated with RTTI.
- "case" statements are a maintenance nightmare - they shall be avoided.
- The system shall be "open for extension, but closed for modification".

Visitor

JavaPro has an article by James Cooper
The November 2000 issue of JavaPro has an article by James Cooper (author of a Java companion to the GoF) on the Visitor design pattern. He suggests it "turns the tables on our object-oriented model and creates an external class to act on data in other classes ... while this may seem unclean ... there are good reasons for doing it."

His primary example. Suppose you have a hierarchy of Employee-Engineer-Boss. They all enjoy a normal vacation day accrual policy, but, Bosses also participate in a "bonus" vacation day program. As a result, the interface of class Boss is different than that of class Engineer. We cannot polymorphically traverse a Composite-like organization and compute a total of the organization's remaining vacation days. "The Visitor becomes more useful when there are several classes with different interfaces and we want to encapsulate how we get data from these classes."

[It also is useful when you want to perform the right algorithm, based on the type of two (or more) objects (aka "double dispatch"). Example algorithms might be: process a "collision" between different kinds of SpaceGame objects, compute the distance between different kinds of shapes, or compute the intersection between different kinds of shapes.]

His Benefits for Visitor include:
* Add functions to class libraries for which you either do not have the source or cannot change the source
* Obtain data from a disparate collection of unrelated classes and use it to present the results of a global calculation to the user program
* Gather related operations into a single class rather than force you to change or derive classes to add these operations
* Collaborate with the Composite pattern

Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended.

Notes:

Layers

Architectural Pattern

The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Motivation

A layered approach is considered better practise than monolithic alternatives. It accentuates decoupling, aides development by teams, and supports incremental coding and testing. Using semi-independent parts also enables the easier exchange of individual parts at a later date.

Example

Networking protocols specify agreements at a variety of abstraction levels, ranging from the the details of bit transmission, to packetizing, to routing, to high-level application logic. Each layer deals with a specific aspect of communication and uses the services of the next lower layer. The OSI 7-layer model has the following architectural model.

Application	layer 7	Provides miscellaneous protocols for common activities
Presentation	layer 6	Structures information and attaches semantics
Session	layer 5	Provides dialog control and synchronization facilities
Transport	layer 4	Breaks messages into packets and guarantees delivery
Network	layer 3	Selects a route from sender to receiver
Data Link	layer 2	Detects and corrects errors in bit sequences
Physical	layer 1	Transmits bits: velocity, bit-code, connections, etc.

Implementation

The following steps describe a step-wise refinement approach to the definition of a layered architecture. This is not necessarily the best method for all applications - often a bottom-up or "yo-yo" approach is better. See also the discussion in step 5.
Not all the following steps are mandatory - it depends on your application. For example, the results of several implementation steps can be heavily influenced or even strictly prescribed by a standards specification that must be followed.

1. Define the abstraction criterion for grouping tasks into layers. This criterion is often the conceptual distance from the platform. Sometimes you encounter other abstraction paradigms, for example the degree of customization for specific domains, or the degree of conceptual complexity. For example, a chess game application may consist of the following layers, listed from bottom to top:

- Elementary units of the game, such as a bishop
- Basic moves, such as castling
- Medium-term tactics, such as the Sicilian defense

- Overall game strategies

In American Football these levels may correspond respectively to linebacker, blitz, a sequence of plays for a two-minute drill, and finally a full game plan.
In the real world of software development we often use a mix of abstraction criterions. For example, the distance from the hardware can shape the lower levels, and conceptual complexity governs the higher ones. An example layering obtained using a mixed-mode layering principle like this is as follows, ordered from top to bottom:

- User-visible elements
- Specific application modules
- Common services level
- Operating system interface level
- Operating system (being a layered system itself, or structured according to the Microkernel pattern [p171])
- Hardware

2. Determine the number of abstraction levels according to your abstraction criterion. Each abstraction level corresponds to one layer of the pattern. Sometimes this mapping from abstraction levels to layers is not obvious. Think about the trade-offs when deciding whether to split particular aspects into two layers or combine them into one. Having too many layers may impose unnecessary overhead, while too few layers can result in a poor structure.

3. Name the layers and assign tasks to each of them. The task of the highest layer is the overall system task, as perceived by the client. The tasks of all other layers are to be helpers to higher layers. If we take a bottom-up approach, then lower layers provide an infrastructure on which higher layers can build. However, this approach requires considerable experience and foresight in the domain to find the right abstractions for the lower layers before being able to define specific requests from higher layers.

4. Specify the services. The most important implementation principle is that layers are strictly separated from each other, in the sense that no component may spread over more than one layer. Argument, return, and error types of functions offered by Layer J should be built-in types of the programming language, types defined in Layer J, or types taken from a shared data definition module. Note that modules that are shared between layers relax the principles of strict layering.
It is often better to locate more services in higher layers than in lower layers. This is because developers should not have to learn a large set of slightly differ-

ent low-level primitives - which may even change during concurrent development. Instead the base layers should be kept "slim" while higher layers can expand to cover a broader spectrum of applicability. This phenomenon is also called the "inverted pyramid of reuse".

5. Refine the layering. Iterate over steps 1 to 4. It is usually not possible to define an abstraction criterion precisely before thinking about the implied layers and their services. Alternatively, it is usually wrong to define components and services first and later impose a layered structure on them according to their usage relationships. Since such a structure does not capture an inherent ordering principle, it is very likely that system maintenance will destroy the architecture. For example, a new component may ask for the services of more than one other layer, violating the principle of strict layering.
The solution is to perform the first four steps several times until a natural and stable layering evolves. "Like almost all other kinds of design, finding layers does not proceed in an orderly, logical way, but consists of both top-down and bottom-up steps, and a certain amount of inspiration..." [Joh95]. Performing both top-down and bottom-up steps alternately is often called "yo-yo" development, mentioned at the start of the this section.

6. Specify an interface for each layer. If Layer J should be a "black box" for Layer J+1, design a "flat interface" that offers all of Layer J's services, and perhaps encapsulate this interface in a Facade object [GHJV95]. A flat interface is an API of function specifications that does not betray a hint of internal components and relationships. A "white-box" approach is that in which Layer J+1 sees the internals of Layer J. And a "gray-box" approach provides a compromise - layer J+1 is aware of the fact that Layer J consists of some number of components, and addresses them separately, but does not see the internal workings of individual components.
Good design practise tells us to use the black-box approach whenever possible, because it supports system evolution better than other approaches. Exceptions to this rule can be made for reasons of efficiency, or a need to access the innards of another layer. The latter occurs rarely, and may be helped by the Reflection pattern [p193], which supports more controlled access to the internal functioning of a component. Arguments over efficiency are debatable, especially when inlining can simply do away with a thin layer of indirection.

7. Structure individual layers. Traditionally, the focus was on the proper relationships between layers, but inside individual layers there was often free-wheeling chaos. When an individual layer is complex it should be broken into separate components. This subdivision can be helped by using finer-grained

patterns. For example, you can use the Bridge pattern [GHJV95] to support multiple implementations of services provided by a layer. The Strategy pattern [GHJV95] can support the dynamic exchange of algorithms used by a layer.

8. Specify the communication between adjacent layers. The most often used mechanism for inter-layer communication is the *push model*. When Layer J invokes a service of Layer J-1, any required information is passed as part of the service call. The reverse is known as the *pull model* and occurs when the lower layer fetches available information from the higher layer at its own discretion. The Publisher-Subscriber [p339] and Pipes and Filters patterns [p53] give details about push and pull model information transfer. However, such models may introduce additional dependencies between a layer and its adjacent higher layer. If you want to avoid dependencies of lower layers on higher layers introduced by the pull model, use callbacks, as described in the next step.

9. Decouple adjacent layers. There are many ways to do this. Often an upper layer is aware of the next lower layer, but the lower layer is unaware of the identity of its users. This implies a one-way coupling only: changes in Layer J can ignore the presence and identity of Layer J+1 provided that the interface and semantics of the Layer J services being changed remain stable. Such a one-way coupling is perfect when requests travel top-down and return values are sufficient to transport the results in the reverse direction.
For bottom-up communication, you can use callbacks and still preserve a top-down one-way coupling. Here the upper layer registers callback functions with the lower layer. This is especially effective when only a fixed set of possible events is sent from lower to higher layers. During start-up the higher layer tells the lower layer what functions to call when specific events occur. The lower layer maintains the mapping from events to callback functions in a registry. The Reactor pattern [Sch94] illustrates an object-oriented implementation of the use of callbacks in conjunction with event demultiplexing. The Command pattern [GHJV95] shows how to encapsulate functions into first-class objects.
You can also decouple the upper layer from the lower layer to a certain degree. Here is an example of how this can be done using object-oriented techniques. The upper layer is decoupled from specific implementation variants of the lower layer by coding the upper layer against an interface (or pure abstract base class). The lower-level implementations can then be easily exchanged, even at run-time. A Layer 2 requestor talks to a Layer 1 provider but does not know which implementation of Layer 1 it is talking to.
For communicating stacks of layers where messages travel both up and down, it is often better explicitly to connect lower levels to higher levels. We therefore again introduce base classes, for example classes L1Provider, L2Provider, and

L3Provider, and additionally L1Parent, L2Parent, and L1Peer. Class L1Parent provides the interface by which Layer 1 classes access the next higher layer, for example to return results, send confirmations, or pass data streams. An analogous argument holds for L2Parent. L1Peer provides the interface by which a message is sent to the layer 1 peer module in another protocol stack in another process or machine. A Layer 1 implementation class therefore inherits from two base classes: L1Provider and L1Peer. A second-level implementation class inherits from L2Provider an L1Parent, as it offers the services of Layer 2 and can serve as the parent of a Layer 1 object. A third-level implementation class finally inherits from L3Provider and L2Parent.
If your programming language separates inheritance and subtyping at the language level, as Java [AG96] does, the above base classes can be transformed into interfaces by pushing data into subclasses and implementing all methods there.

10. Design an error-handling strategy. Error handling can be rather expensive for layered architectures with respect to processing time and, notably, programming effort. An error can either be handled in the layer where it occurred, or be passed to the next higher layer. In the latter case, the lower layer must transform the error into an error description meaningful to the higher layer. As a rule of thumb, try to handle errors at the lowest layer possible. This prevents higher layers from being swamped with many different errors and voluminous error-handling code. At a minimum, try to condense similar error types into more general error types, and only propagate these more general errors. If you do not do this, higher layers can be confronted with error messages that apply to lower-level abstractions that the higher layer does not understand. And who hasn't seen totally cryptic error messages being popped up to the highest layer of all - the user?

Variants

Relaxed layered architecture. Each layer may use the services of all (or some of the) layers below it. The gain in flexibility and performance in this approach is paid for by loss of maintainability. This is often a high price to pay, and you

should consider carefully before giving in to the demands of developers asking for shortcuts.

Virtual machines. We can speak of each lower layer as a *virtual machine* that insulates its higher layers from low-level details, or varying hardware. For example, Java is predicated on the Java Virtual Machine (JVM). Java code can "write once, run anywhere" because it relies on the illusion of a universal platform created by a plethora of JVMs - each one platform-specific in its implementation, but perfectly common in its interface.

Notes:

Null Object Pattern

Taken from http://occs.cs.oberlin.edu/~jwalker/nullObjPattern/

Intent

Provide an object as a surrogate for the lack of an object of a given type. The Null Object provides intelligent do nothing behavior, hiding the details from its collaborators.

Also Known as

Stub, Active Nothing

Motivation

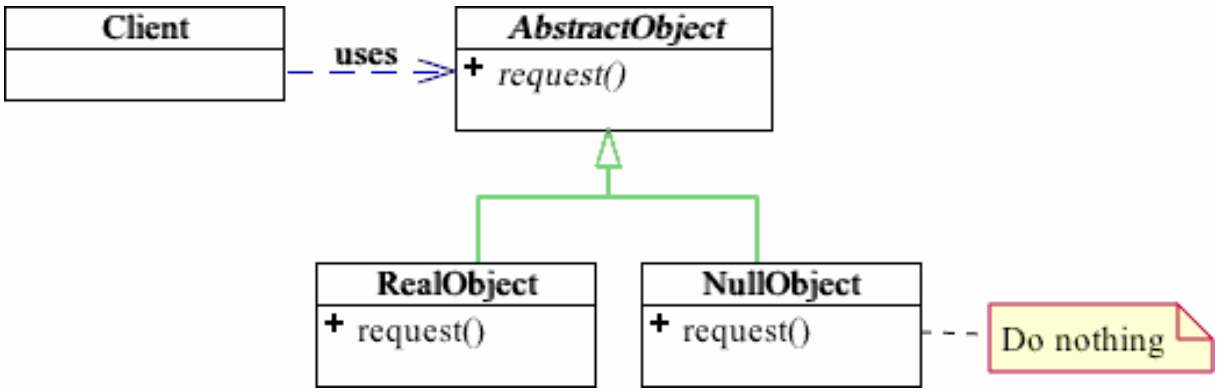
Sometimes a class that requires a collaborator does not need the collaborator to do anything. However, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

Consider for example a simple screen saver which displays balls that move about the screen and have special color effects. This is easily achieved by creating a Ball class to represent the balls and using a Strategy pattern [GHJV95, page 315] to control the ball's motion and another Strategy pattern to control the ball's color. It would then be trivial to write strategies for many different types of motion and color effects and create balls with any combination of those. However, to start with you want to create the simplest strategies possible to make sure everything is working. And these strategies could also be useful later since you want as strategies as possible strategies.

Now, the simplest strategy would be no strategy. That is do nothing, don't move and don't change color. However, the Strategy pattern requires the ball to have objects which implement the strategy interfaces. This is where the Null Object pattern becomes useful. Simply implement a NullMovementStrategy which doesn't move the ball and a NullColorStrategy which doesn't change the ball's color. Both of these can probably be implemented with essentially no code. All the methods in these classes do "nothing". They are perfect examples of the Null Object Pattern.

The key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed. As compared to using a special "null" value which doesn't actually implement the abstract interface and which must constantly be checked for with special code in any object which uses the abstract interface.
It is sometimes thought that Null Objects are over simple and "stupid" but in

Null Object Pattern



truth a Null Object always knows exactly what needs to be done without interacting with any other objects. So in truth it is very "smart."

Applicability

Use the Null Object pattern when:
an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists.
some collaborator instances should do nothing.
you want to abstract the handling of null away from the client.

Participants

* Client
  - o requires a collaborator.
* AbstractObject
  - o declares the interface for Client's collaborator.
  - o implements default behavior for the interface common to all classes, as appropriate.
* RealObject
  - o defines a concrete subclass of AbstractObject whose instances provide useful behavior that Client expects.
* NullObject
  - o provides an interface identical to AbstractObject's so that a null object can be substituted for a real object.
  - o implements its interface to do nothing. What exactly it means to do nothing depends on what sort of behavior Client is expecting.
  - o when there is more than one way to do nothing, more than one NullObject class may be required.

Collaborations

* Clients use the AbstractObject class interface to interact with their collaborators. If the receiver is a RealObject, then the request is handled to provide real behavior. If the receiver is a NullObject, the request is handled by doing nothing or at least providing a null result.

Consequences

The Null Object pattern:

* defines class hierarchies consisting of real objects and null objects. Null objects can be used in place of real objects when the object is expected to do nothing. Whenever client code expects a real object, it can also take a null object.

* makes client code simple. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know (and shouldn't care) whether

they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write testing code which handles the null collaborator specially.

* encapsulates the do nothing code into the null object. The do nothing code is easy to find. Its variation with the AbstractObject and RealObject classes is readily apparent. It can be efficiently coded to do nothing. It does not require variables that contain null values because those values can be hard-coded as constants or the do nothing code can avoid using those values altogether.

* makes the do nothing code in the null object easy to reuse. Multiple clients which all need their collaborators to do nothing will all do nothing the same way. If the do nothing behavior needs to be modified, the code can be changed in one place. Thereafter, all clients will continue to use the same do nothing behavior, which is now the modified do nothing behavior.

* makes the do nothing behavior difficult to distribute or mix into the real behavior of several collaborating objects. The same do nothing behavior cannot easily be added to several classes unless those classes all delegate the behavior to a class which can be a null object class.

* can necessitate creating a new NullObject class for every new AbstractObject class.

* can be difficult to implement if various clients do not agree on how the null object should do nothing as when your AbstractObject interface is not well defined.

* always acts as a do nothing object. The Null Object does not transform into a Real Object.

Implementation

There are several issues to consider when implementing the Null Object pattern:

1. Null Object as Singleton. The Null Object class is often implemented as a Singleton [GHJV95, page 127]. Since a null object usually does not have any state, its state can't change, so multiple instances are identical. Rather than use multiple identical instances, the system can just use a single instance repeatedly.
2. Clients don't agree on null behavior. If some clients expect the null object to do nothing one way and some another, multiple NullObject classes will be

required. If the do nothing behavior must be customized at run time, the NullObject class will require pluggable variables so that the client can specify how the null object should do nothing (see the discussion of pluggable adaptors in the Adapter pattern [GHJV95, page 142]). This may generally be a symptom of the AbstractObject not having a well defined (semantic) interface.

3. Transformation to Real Object. A Null Object does not transform to become a Real Object. If the object may decide to stop providing do nothing behavior and start providing real behavior, it is not a null object. It may be a real object with a do nothing mode, such as a controller which can switch in and out of read-only mode. If it is a single object which must mutate from a do nothing object to a real one, it should be implemented with the State pattern [GHJV95, page 305] or perhaps the Proxy pattern [GHJV95, page 207]. In this case a Null State may be used or the proxy may hold a Null Object.

4. Null Object is not Proxy. The use of a null object can be similar to that of a Proxy [GHJV95, page 207], but the two patterns have different purposes. A proxy provides a level of indirection when accessing a real subject, thus controlling access to the subject. A null collaborator does not hide a real object and control access to it, it replaces the real object. A proxy may eventually mutate to start acting like a real subject. A null object will not mutate to start providing real behavior, it will always provide do nothing behavior.

5. Null Object as special Strategy. A Null Object can be a special case of the Strategy pattern [GHJV95, page 315]. Strategy specifies several ConcreteStrategy classes as different approaches for accomplishing a task. If one of those approaches is to consistently do nothing, that ConcreteStrategy is a NullObject. For example, a Controller is a View's Strategy for handling input, and NoController is the Strategy that ignores all input.

6. Null Object as special State. A Null Object can be a special case of the State pattern [GHJV95, page 305]. Normally, each ConcreteState has some do nothing methods because they're not appropriate for that state. In fact, a given method is often implemented to do something useful in most states but to do nothing in at least one state. If a particular ConcreteState implements most of its methods to do nothing or at least give null results, it becomes a do nothing state and as such is a null state. [Woolf96]

7. Null Object as Visitor host. A Null Object can be used to allow a Visitor [GHJV95, page 331] to safely visit a hierarchy and handle the null situation.

8. The Null Object class is not a mixin. Null Object is a concrete collaborator class that acts as the collaborator for a client which needs one. The null behavior is not designed to be mixed into an object that needs some do nothing behavior. It is designed for a class which delegates to a collaborator all of the behavior that may or may not be do nothing behavior. [Woolf96]

Related Patterns

Singleton [GHJV95, page 127] is often used to implement a Null Object since multiple instances would act exactly the same and have no internal state that could change.

Flyweight [GHJV95, page 195] can be used when multiple null objects are implemented as instances of a single NullObject class.

Strategy [GHJV95, page 315] patterns often have one Null Object representing the strategy of doing nothing.

State [GHJV95, page 305] often uses Null Object to represent the state in which the client should do nothing.

Iterators [GHJV95, page 257] may have Null Object as a special case which doesn't iterate over anything.

Adapters [GHJV95, page 142] may have Null Object as a special case which pretend to adapt another object without actually adapting anything.

Bruce Anderson has also written about the Null Object pattern, which he also refers to as "Active Nothing." [Anderson95]

NullObject is a special case of the Exceptional Value pattern in The CHECKS Pattern Language [Cunningham95]. An Exceptional Value is a special Whole Value (another pattern) used to represent exceptional circumstances. It will either absorb all messages or produce Meaningless Behavior (another pattern). A NullObject is one such Exceptional Value.

Notes:

J2EE Patterns Catalog

http://java.sun.com/blueprints/patterns/catalog.html

Each pattern in this catalog includes sample code from JavaTM BluePrints reference applications such as the Java Pet Store sample application.

Pattern Name	Description
Business Delegate [ACM01]	Reduce coupling between Web and Enterprise JavaBeansTM tiers
Composite Entity [ACM01]	Model a network of related business entities
Composite View [ACM01]	Separately manage layout and content of multiple composed views
Data Access Object (DAO) [ACM01]	Abstract and encapsulate data access mechanisms
Fast Lane Reader	Improve read performance of tabular data
Front Controller [ACM01]	Centralize application request processing
Intercepting Filter [ACM01]	Pre- and post-process application requests
Model-View-Controller	Decouple data representation, application behavior, and presentation
Service Locator [ACM01]	Simplify client access to enterprise business services
Session Facade [ACM01]	Coordinate operations between multiple business objects in a workflow
Transfer Object [ACM01]	Transfer business data between tiers
Value List Handler [ACM01]	Efficiently iterate a virtual list
View Helper [ACM01]	Simplify access to model state and data access logic

Business Delegate

Brief Description
In distributed applications, lookup and exception handling for remote business components can be complex. When applications use business components directly, application code must change to reflect changes in business component APIs.

These problems can be solved by introducing an intermediate class called a business delegate, which decouples business components from the code that uses them. The Business Delegate pattern manages the complexity of distributed component lookup and exception handling, and may adapt the business component interface to a simpler interface for use by views.

Detailed Description
See the Core J2EETM Patterns

Detailed Example
Sample application business delegate class AdminRequestBD handles distributed lookup and catches and adapts exceptions in the sample application order processing center (OPC).

* The AdminRequestBD business delegate manages distributed component lookup and handles exceptions.

The structure diagram in Figure 1 shows the ApplRequestProcessor servlet using AdminRequestBD to find and use distributed business components.

The code sample below shows the constructor for AdminRequestBD. It uses class ServiceLocator to acquire the remote home interface of session facade OPCAdminFacade . (See the Service Locator and Session Facade design patterns.) It uses the remote home interface to create a OPCAdminFacade remote component interface, which it maintains in a private field.

The block that locates and creates the enterprise bean reference catches exceptions related to finding the home interface and to creating the component interface. Any exception that occurs is then wrapped in an AdminBDException , which effectively hides the implementation details of the business delegate from its clients.

Business Delegate

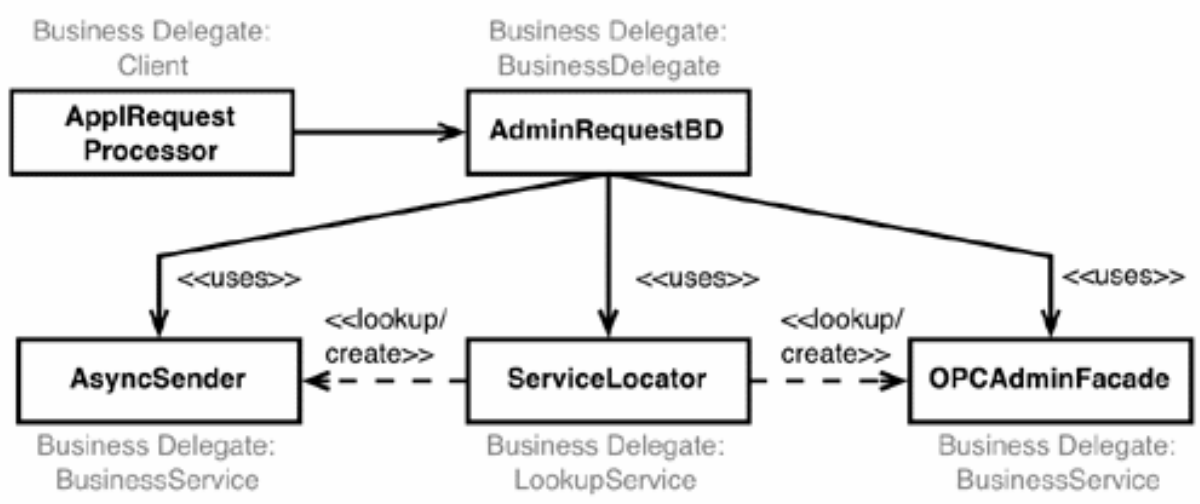


Figure 1. AdminRequestBD locates and adapts other business components

Business Delegate

```
   public AdminRequestBD() throws AdminBDException {
       try {
            OPCAdminFacadeHome home = (OPCAdminFacadeHome)
ServiceLocator.getInstance().getRemoteHome(OPC_ADMIN_NAME, OP-
CAdminFacadeHome.class);
            opcAdminEJB = home.create();
        } catch (ServiceLocatorException sle) {
            throw new AdminBDException(sle.getMessage());
        } catch (CreateException ce) {
            throw new AdminBDException(ce.getMessage());
        } catch (RemoteException re) {
            throw new AdminBDException(re.getMessage());
        }
    }
```

* The OPC request processor uses AdminRequestBD for simple access to business components components.

OPC servlet class ApplRequestProcessor receives service requests from the admin client in the form of XML messages transmitted using HTTP. One of these request is for statistics about orders that have a given status.

Method ApplRequestProcessor.getOrders receives part of an XML DOM tree representing a Web service request. It extracts the status code from the document and uses AdminRequestBD.getOrdersByStatus to retrieve a list of order information. The interface to that list is transfer object interface OrdersTO . (See the Transfer Object pattern.) The code in the request processor that retrieves the order information appears in the following code sample.

```
public class ApplRequestProcessor extends HttpServlet {
...
    String getOrders(Element root) {
        try {
            AdminRequestBD bd = new AdminRequestBD();
            NodeList nl = root.getElementsByTagName("Status");
            String status = getValue(nl.item(0));
            OrdersTO orders = bd.getOrdersByStatus(status);
...
}
```

Because it has already created the reference to an OPCAdminFacadeEJB, the AdminRequestBD object can simply forward the call to the enterprise

bean's method getOrdersByStatus, as follows:

```
public class AdminRequestBD {
...
    public OrdersTO getOrdersByStatus(String status)
        throws AdminBDException {

        try {
            return opcAdminEJB.getOrdersByStatus(status);
        } catch (RemoteException re) {
            throw new AdminBDException(re.getMessage());
        } catch (OPCAdminFacadeException oafee) {
            throw new AdminBDException(oafee.getMessage());
        }
    }
...
}
```

Notice again that the method catches any exceptions that the enterprise bean may throw and re-throws an exception type that is specific to the business delegate's interface. This hides the business delegate's implementation details from the client.

Notes:

Relationship between patterns – Rules of thumb

From
Huston Design Patterns
http://home.earthlink.net/~huston2/dp/patterns.html

Creational Patterns

1 Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

2 Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

3 Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

4 Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

5 Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

6 Builder is to creation as Strategy is to algorithm. [Icon, p8-13]

7 Builder often builds a Composite. [GOF, p106]

8 Factory Methods are usually called within Template Methods. [GOF, p116]

9 Factory Method: creation through inheritance. Prototype: creation through

delegation.

10 Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

11 Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

12 Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. [GOF, p126]

Structural Patterns

1 Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

2 Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, p161]

3 Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

4 Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, p149]

5 Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

6 Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes. [GOF, pp173,349]

Relationship between patterns

7 Composite can let you compose a Mediator out of smaller pieces through recursive composition. [Vlissides, Apr96, p18]

8 Decorator lets you change the skin of an object. Strategy lets you change the guts. [GOF, p184]

9 Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

10 Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

11 Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

12 Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

13 Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF, p193]

14 Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

15 Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

16 Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Relationship between patterns

17 Flyweight explains when and how State objects can be shared. [GOF, p313]

Behavioral Patterns

1 Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it. [GOF, p222]

2 Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

3 Chain of Responsibility can use Command to represent requests as objects. [GOF, p349]

4 Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor. [GOF, p232]

5 Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

6 Command can use Memento to maintain the state required for an undo operation. [GOF, p242]

7 MacroCommands can be implemented with Composite. [GOF, p242]

8 A Command that must be copied before being placed on a history list acts as a Prototype. [GOF, p242]

9 Interpreter can use State to define parsing contexts. [GOF, p349]

10 The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Relationship between patterns

11 Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF. p255]

12 Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

13 Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

14 Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

15 On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

16 Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa). [GOF, p193]

17 Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

18 State is like Strategy except in its intent. [Coplien, Mar96, p88]

19 Flyweight explains when and how State objects can be shared. [GOF, p313]

20 State objects are often Singletons. [GOF, p313]

21 Strategy lets you change the guts of an object. Decorator lets you change

the skin. [GOF, p184]

22 Strategy is to algorithm. as Builder is to creation. [Icon, p8-13]

23 Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, Mar96, p88]

24 Strategy objects often make good Flyweights. [GOF, p323]

25 Strategy is like Template Method except in its granularity. [Coplien, Mar96, p88]

26 Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]

27 The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters. [Johnson, Huni, Engel, 1995, p8]

Notes:

Patterns for Architects (houses and spaces, not software)

Meeting Spaces

The following pattern excepts from Christopher Alexander's book, *A Pattern Language*, offer insight into the nature of good meeting spaces for study groups:

Sitting Circle (185)

...A group of chairs, a sofa and a chair, a pile of cushions -- these are the most obvious things in everybody's life -- and yet to make them work, so people become animated and alive in them, is a very subtle business. Most seating arrangements are sterile, people avoid them, nothing ever happens there. Others seem somehow to gather life around them, to concentrate and liberate energy. What is the difference between the two?

...therefore

Place each sitting space in a position which is protected, not cut by paths or movements, roughly circular, made so that the room itself helps suggest the circle -- not too strongly -- with paths and activities around it, so that people naturally gravitate toward the chairs when they get into the mood to sit. Place the chairs and cushions loosely in the circle, and have a few too many.

Pools Of Light (252)

...Uniform illumination --the sweetheart of the lighting engineers-- serves no useful purpose whatsoever. In fact, it destroys the social nature of space, and makes people feel disoriented and unbounded.

...therefore

Place the lights low, and apart, to form individual pools of light which encompass chairs and tables like bubbles to reinforce the social character of the spaces which they form. Remember that you can't have pools of light without the darker places in between.

Different Chairs (251)

...People are different sizes; they sit in different ways. And yet there is a tendency in modern times to make all chairs alike.

...therefore

Never furnish any place with chairs that are identically the same. Choose a variety of different chairs, some big, some small, some softer than others, some rockers, some very old, some new, with arms, without arms, some wicker, some wook, some cloth.

What is a Pattern?

For those of you unfamiliar with what this "pattern" thing is, Brad Appleton sums it up very well:

Fundamental to any science or engineering discipline is a **common vocabulary for expressing its concepts**, and a language for relating them together. The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. **Patterns help create a shared language for communicating insight and experience** about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge that defines our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design.

You can easily relate this description to fields other than software architecture and design. Patterns help us capture our experiences and relate them to form a more complete body of knowledge in a chosen discipline. They also serve as a common vocabulary for streamlining the communication of complex problems and solutions. In this way, patterns can be a powerful way to organize, relate, and communicate our experiences.

Course Benefits

After completing this course, students will know how to:

- Describe the history of patterns development.
- Appreciate the benefits of a patterns approach to programming design.
- List and classify various standard patterns.
- Describe the design, purpose and behaviour of each of the original GOF design patterns
- Implement patterns in various programming languages.
- Research, understand, and implement other design pattern catalogs such as .NET and Java

Prerequisites:

Understanding of object oriented programming techniques, experience with an object-oriented programming language such as Java, C#, or C++, some exposure to UML diagramming models.

Notes:

END OF PATTERNS HANDOUT