COMP 6471
Software Design Methodologies

Fall 2011

Dr Greg Butler

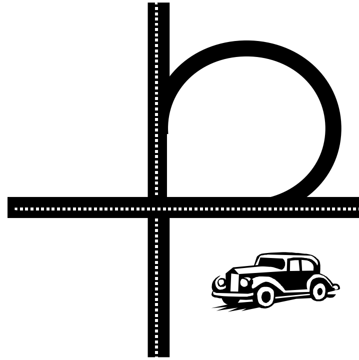
http://www.cs.concordia.ca/~gregb/home/comp6471-fall2011.html

Week 8 Outline

- Software Design Patterns

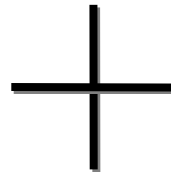
Overview of Patterns

- Present *solutions* to common software *problems* arising within a certain *context*

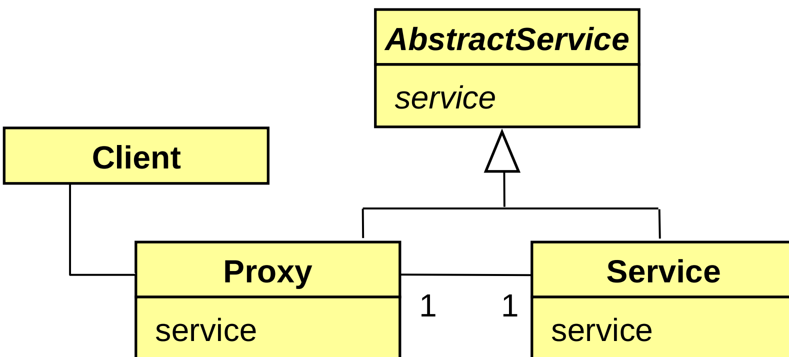


- Help resolve key software design forces

- ***Flexibility***
- ***Extensibility***
- ***Dependability***
- ***Predictability***
- ***Scalability***
- ***Efficiency***



- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs



The Proxy Pattern

- Generally codify expert knowledge of design strategies, constraints & "best practices"

A Partial Bibliography

- « A System of Pattern » Bushmann et All
- « Design Patterns » Gamma et All
- « Concurrent Programming in Java » D. Lea.
- « Distributed Objects » Orfali et All
- « Applying UML and Patterns » Larman
- « Head First Design Patterns » Freeman and Freeman

Patterns

- « Patterns help you build on the collective experience of skilled software engineers. »
- « They capture existing, well-proven experience in software development and help to promote good design practice »
- « Every pattern deals with a specific, recurring problem in the design or implementation of a software system »
- « Patterns can be used to construct software architectures with specific properties… »

Becoming a Chess Master

- First learn the rules.
  - *e.g.*, names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn the principles.
  - *e.g.*, relative value of pieces, strategic value of center squares, pins, etc.
- However, to become a master of chess, one must study the games of other masters.
  - These games contain patterns that must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns.

Becoming a Software Design Master

- First learn the rules.
  - *e.g.*, the algorithms, data structures and languages of software
- Then learn the principles.
  - *e.g.*, structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters.
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns.

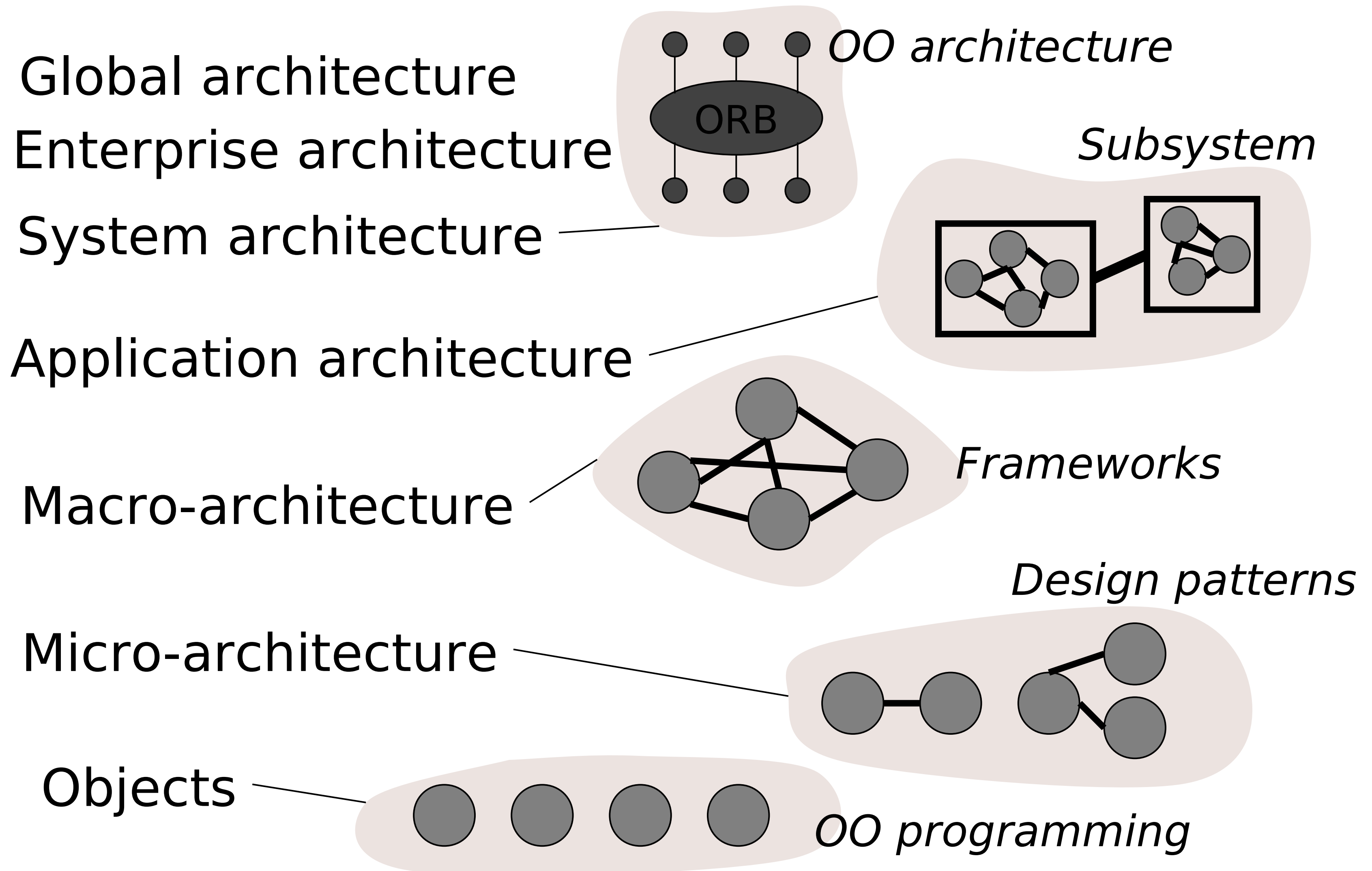
Why Use Design Patterns?

- If you’re a software engineer, you should know about them anyway.

- There are many architectural patterns published, and the GoF design patterns are a prerequisite to understanding them, *e.g.*
  - Mowbray and Malveau – CORBA Design Patterns
  - Schmidt et al – Pattern-Oriented Software Architecture

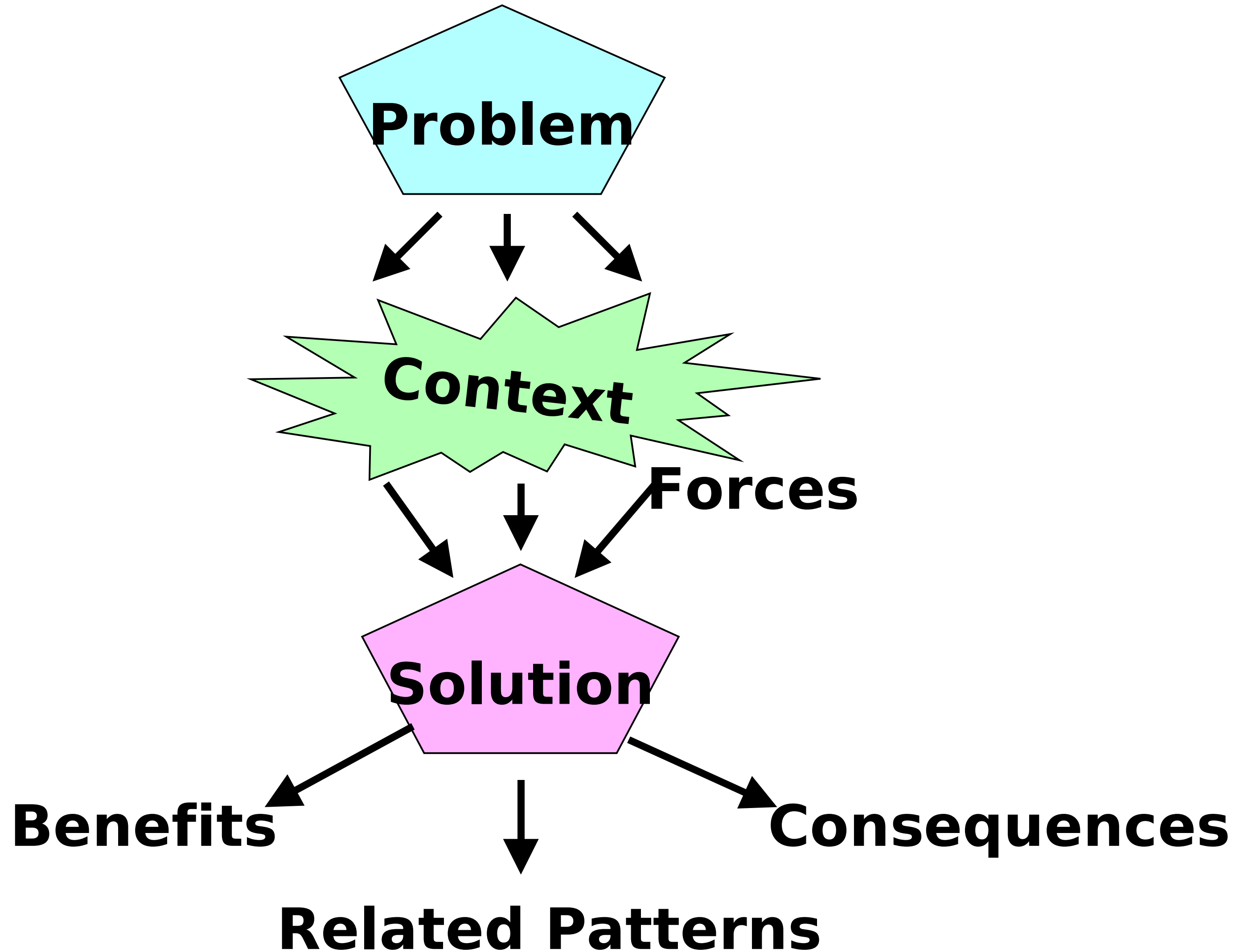
- Design patterns help you *break out* of first-generation OO thought patterns.

The seven layers of architecture*



* Mowbray and Malveau

How Patterns Arise



Structure of a Pattern

- Name
- Intent
- Motivation
- Applicability
- Structure
- Consequences
- Implementation
- Known Uses
- Related Patterns

Design Patterns

The design pattern concept can be viewed as an **abstraction of imitating useful parts** of other software products.

The design pattern is **description of communicating objects and classes** that are customized to solve a general design problem in a particular context.

Classification of Design Patterns

Creational patterns defer some part of object creation to a subclass or another object.

Structural patterns composes classes or objects.

Behavioral patterns describe algorithms or cooperation of objects.

Creational Design Patterns

Factory Method define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory provides an interface for creating families of related objects without specifying their concrete classes.

Structural Design Patterns

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

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

Proxy provides a surrogate or representative for another object to control access to it.

Behavioral Design Patterns

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

Strategy defines a family of algorithms, encapsulates each one, and makes them interchangable. Strategy lets algorithm vary independently from clients that use it.

Some Key Patterns

- The following patterns are a good "basic" set of design patterns.
- Competence in recognizing and applying these patterns *will* improve your low-level design skills.
- (The slides are necessarily brief and do not follow the structure just given above!)

Singleton

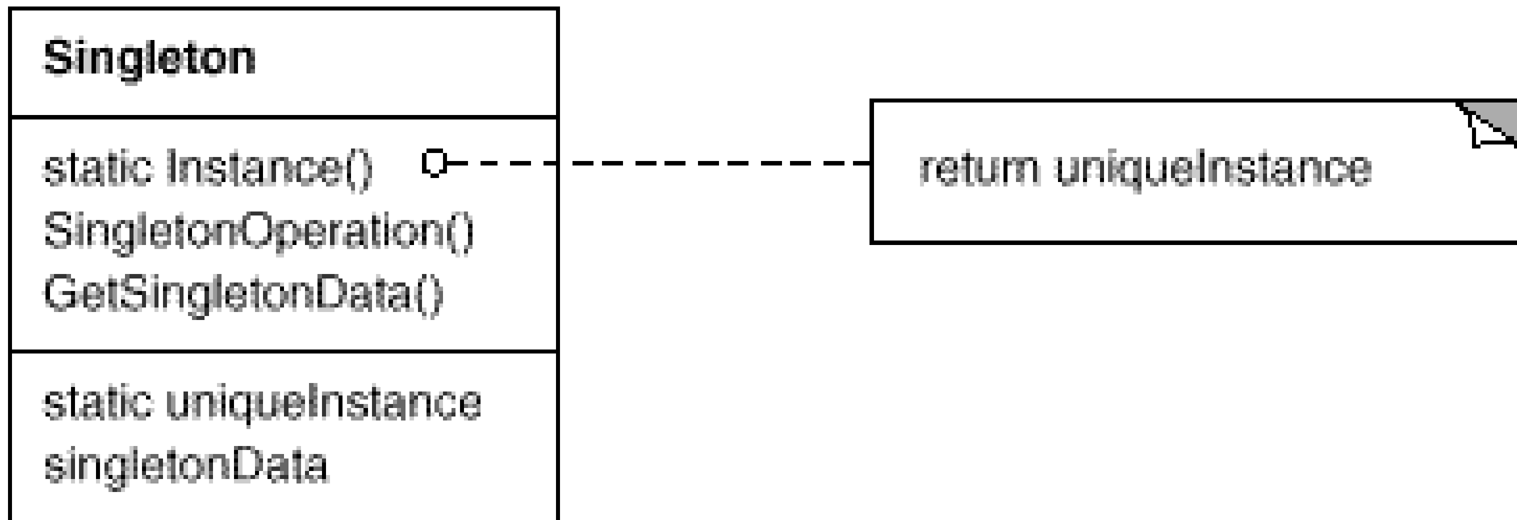
Intent: Ensure a class only has one instance, and provide a global point of access to it.

- It's easy to create one instance of an object.

  ...but how do you ensure that only one instance can be created?

- Sometimes it really does matter that an object is unique. For example, a system may have many printers, but should have only one print spooler. Multiple file managers would only get in each others' way, etc.

Singleton Structure



The Classic Singleton Implementation

```
public class ClassicSingleton
{
      private static ClassicSingleton instance = null;

      protected ClassicSingleton()
      {
         // exists only to prevent direct instantiation
      }

      public static ClassicSingleton getInstance()
      {
            if (instance == null)
            {
                  instance = new ClassicSingleton();
            }

            return instance;
      }
}
```

Warning: This code is not thread-safe!

Command

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

Translation:

Implement a programmable remote control. :-)

Command

Consider what happens when you order a meal in a restaurant:

1) You tell the server what you want to eat.
2) The server writes your instructions on an order pad.
3) The server delivers the order to the kitchen.
4) The chef reads the order and produces the appropriate meal.

What this amounts to is that the top sheet of paper on the order pad (the "Order") encapsulates a request for a specific meal:

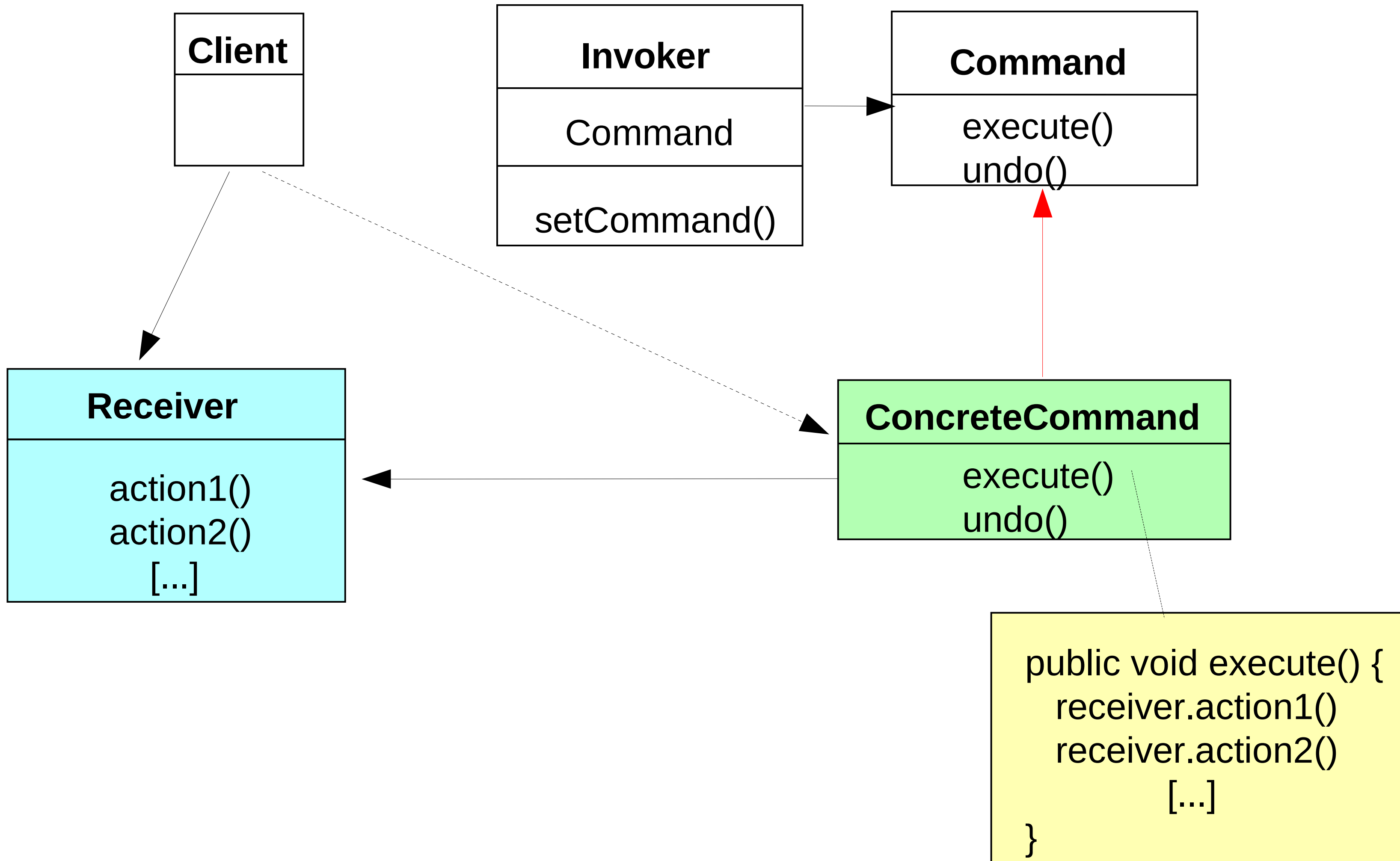
- The server doesn't need to know how to cook the meal.
- The chef doesn't need to know how the order was obtained.

Command

Now let's describe the same transaction in more general terms:

- A client creates a Command object. This contains whatever commands the client wishes to use, and specifies the Receiver object which will eventually run the commands.
- The Command object includes a method called execute(). When run, this method will run the client's chosen commands.
- The Command object is passed to an Invoker, which will store it (using a method called setCommand()) until it's needed (and until the commands are ready to be run).
- Eventually the Invoker will call the Command's execute() method. This will cause the Command's Receiver to run the commands originally specified by the client.
- Translations:
    - client = restaurant customer
    - Command object = order
    - Invoker = server
    - setCommand() = server writing down an order
    - Receiver = chef
    - execute() = chef preparing the meal based on the order

Command



Command Sequence

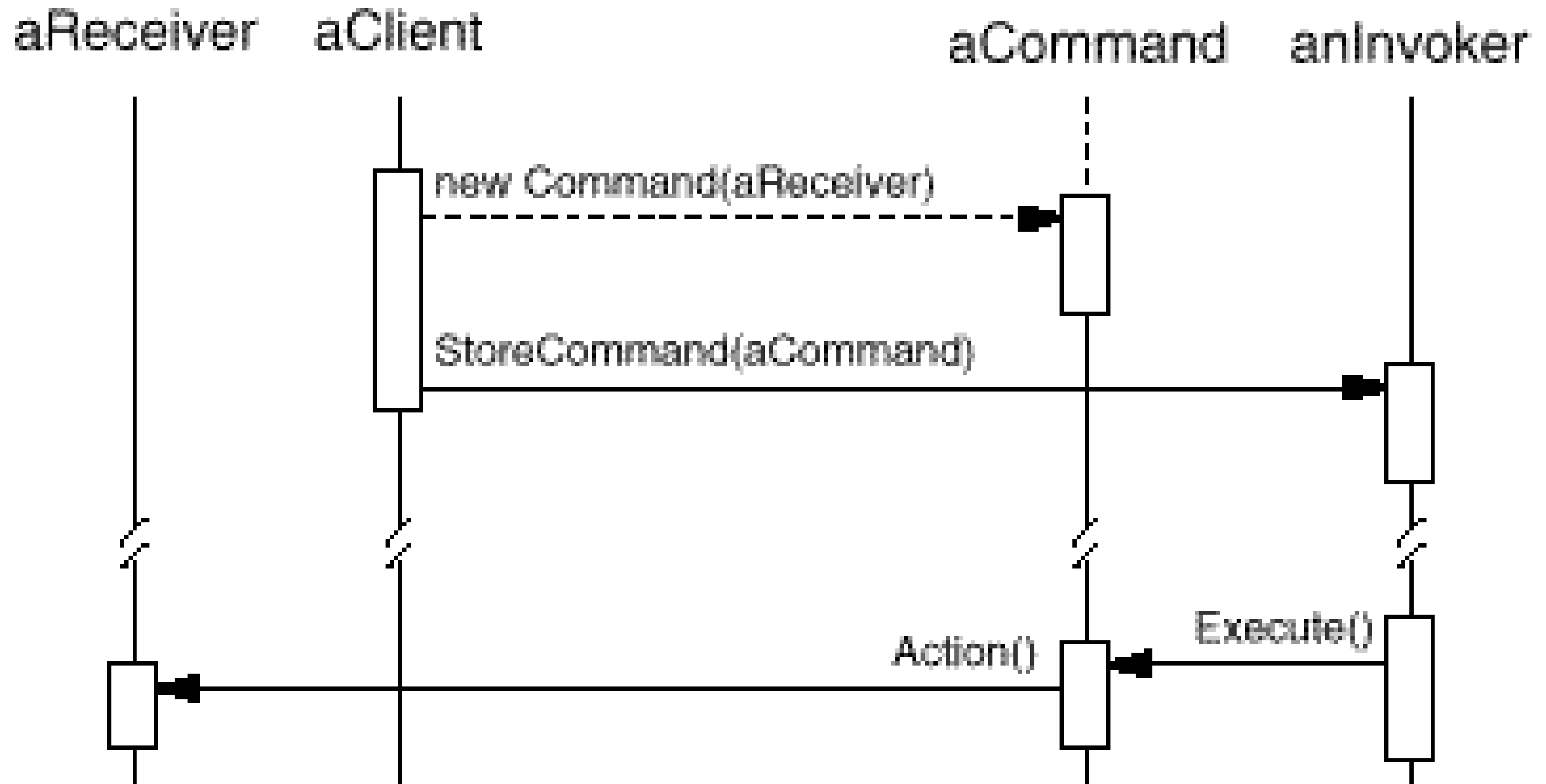
1) The Client creates a new ConcreteCommand object, and specifies its Receiver. (You tell the server what you want to eat.)

2) The Client calls the Invoker's setCommand() method to store the ConcreteCommand. (The server writes your instructions on an order pad.)

3) The Invoker (eventually) calls the ConcreteCommand's execute() method. (The server delivers the order to the kitchen.)

4) The ConcreteCommand's execute() method calls methods in the Receiver in order to fulfill the Client's request. (The chef reads the order and produces the appropriate meal.)

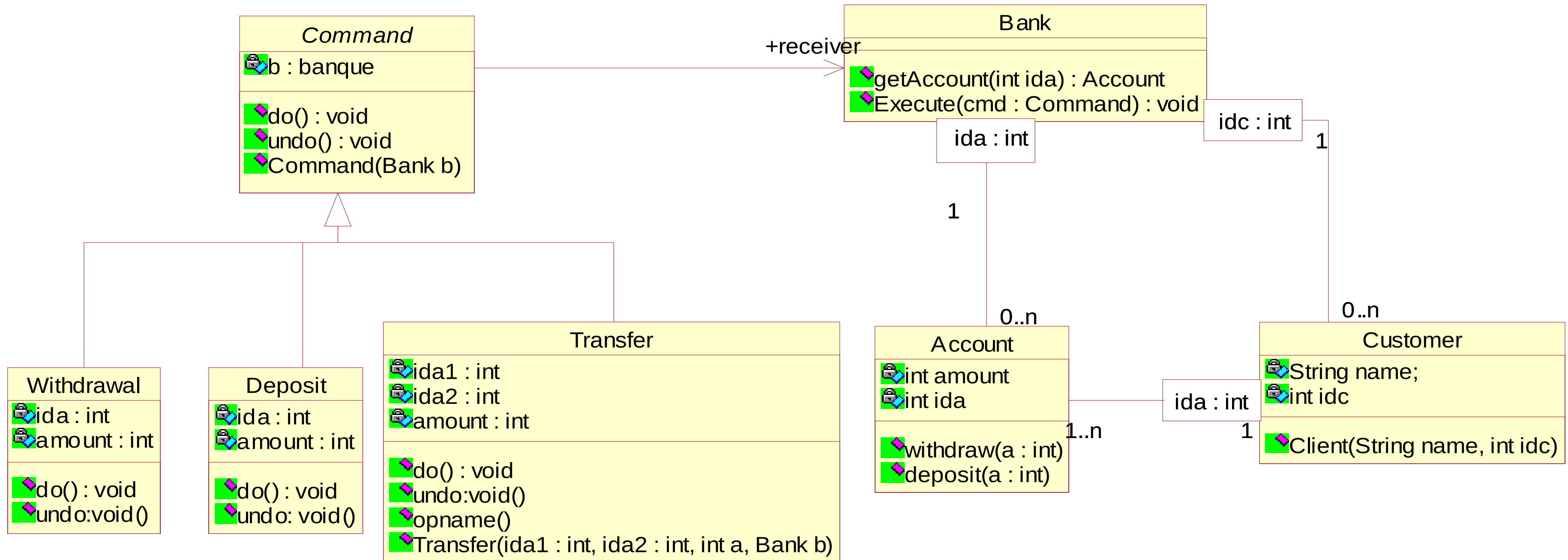
Command Sequence



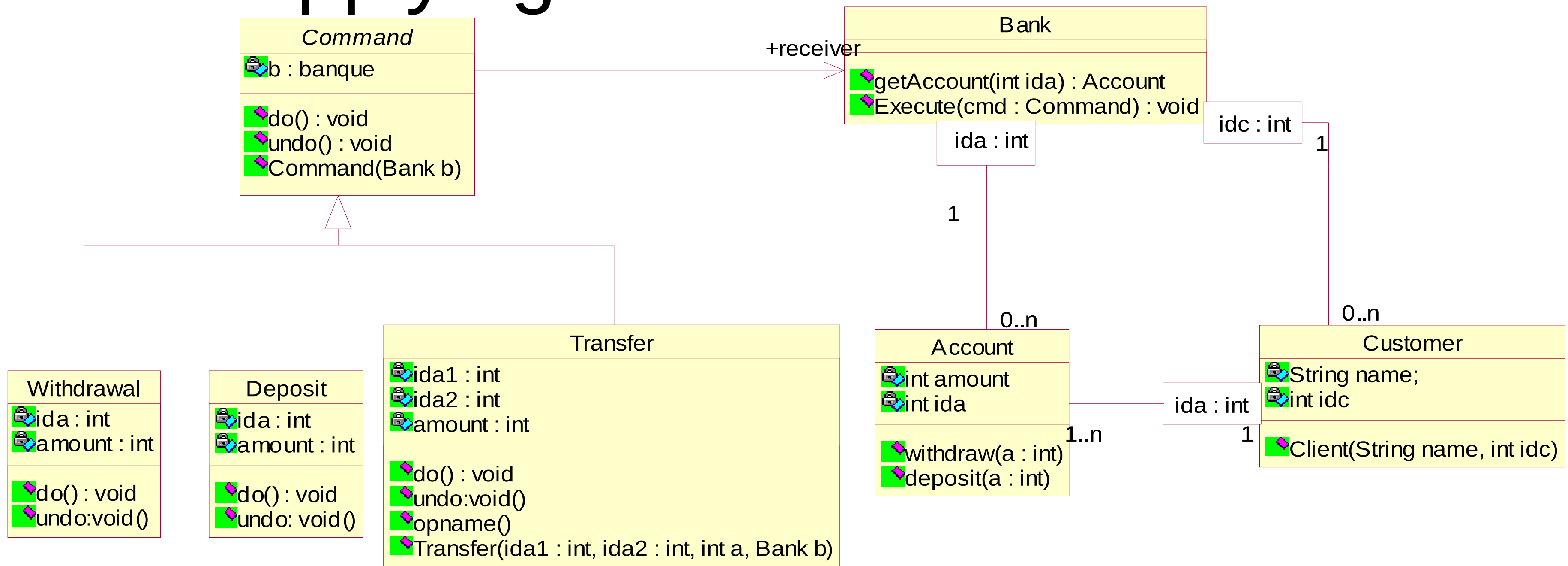
Command Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- It's easy to add new Commands, because you don't have to change existing classes.

Applying the Command Pattern



Applying the Command Pattern



A typical transfer operation would look like this:

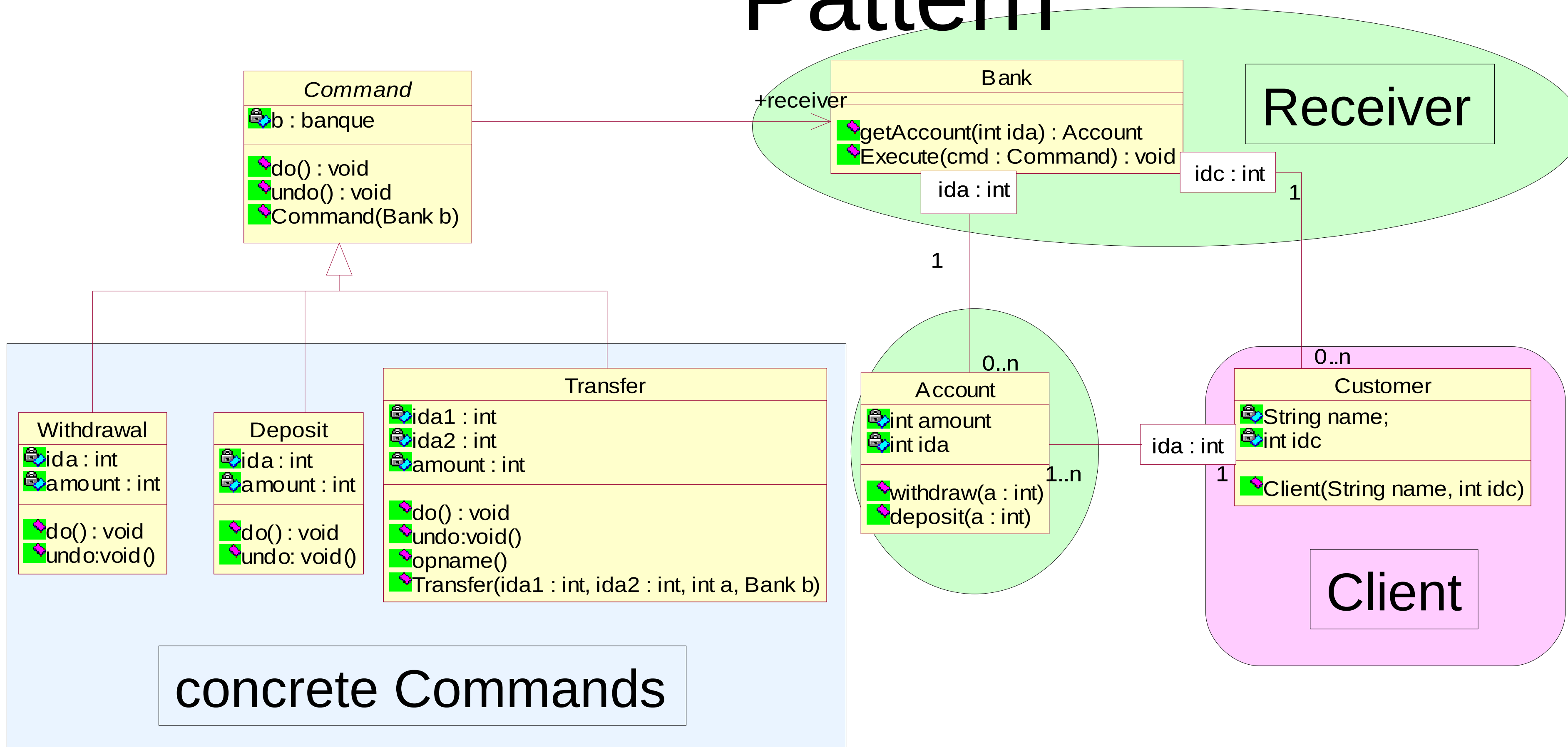
(0) a Customer creates a new Transfer object,
e.g. Transfer T = new Transfer(account1,account2,amount,bank);

(1) Customer calls bank.Execute(T);

(2) bank.Execute() calls T.do();

(3) T.do() calls bank.getAccount(account1), Bank.getAccount(account2), Account1.withdraw(amount), Account2.deposit(amount)

Applying the Command Pattern



NOTE: Client is also Invoker! Account and Bank combined are the Receiver

Factory

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

Translation:

Instantiate new objects, without using `new` directly — wait until run-time to decide what kind of object to instantiate.

– Also known as "Factory Method" or "Virtual Constructor".

Factory

Applicability: use when

- a class cannot anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass to delegate

Factory

- A good analogy for this is a pasta maker. A pasta maker will produce different types of pasta, depending what kind of disk is loaded into the machine.
- All disks should have certain properties in common, so that they will work with the pasta maker. This specification for the disks is the Abstract Factory, and each specific disk is a Factory.
- You will never see an Abstract Factory, because one can never exist, but all Factories (pasta maker disks) inherit their properties from the Abstract Factory.
- In this way, all disks will work in the pasta maker, since they all comply with the same specifications. The pasta maker doesn't care what the disk is doing, nor should it. You turn the handle on the pasta maker, and the disk makes a specific shape of pasta come out.
- Each individual disk contains the information of how to create the pasta, and the pasta maker does not.

Factory

Going from pasta to pizza :-), consider the following code:

```
Pizza orderPizza() {
     Pizza pizza = new Pizza();
     pizza.prepare();
     pizza.bake();
     pizza.cut();
     pizza.box();
     return pizza;
}
```

This works just fine — as long as we always want the same **kind** of pizza. :-)

How would we handle different toppings?

Factory

Different toppings, take 1:

```
Pizza orderPizza(String type) {
     Pizza pizza;
     if (type.equals("cheese")) {
        pizza = new CheesePizza();
     } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
     } [...other types go here...]
     pizza.prepare();  // each type knows how
     pizza.bake();     // to prepare itself :-)
     pizza.cut();
     pizza.box();
     return pizza;
}
```

This assumes that each type of pizza must implement the Pizza interface. It works, but...

Factory

```
if (type.equals("cheese")) {
    pizza = new CheesePizza();
} else if (type.equals("pepperoni")) {
    pizza = new PepperoniPizza();
} else if (type.equals("veggie")) {
    pizza = new VeggiePizza();
} else if (type.equals("mexican")) {
    pizza = new MexicanPizza();
} [...]
```

You can see what happens here if we want to add new types of pizzas, or to eliminate existing types.

This code does work, but it really isn't a good design. How can we improve it?

Factory

Step 1 is to take the creation code out of the order method altogether:

```
Pizza orderPizza(String type) {
     Pizza pizza;
     pizza = factory.createPizza(type);
     pizza.prepare();
     pizza.bake();
     pizza.cut();
     pizza.box();
     return pizza;
}
```

Now all we need to do is figure out how to implement `factory.createPizza()` :-)

Factory

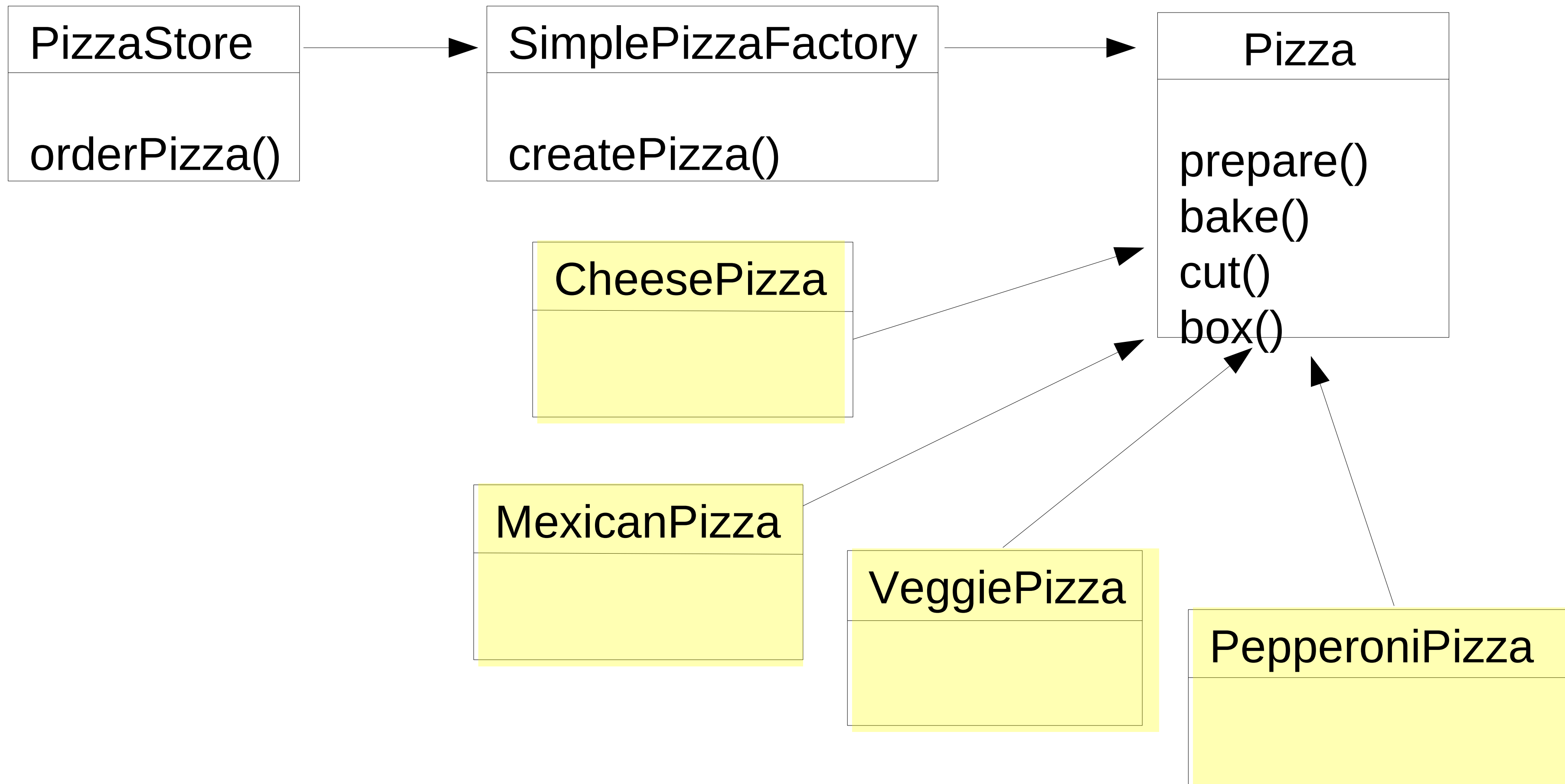
Now let's put that in context:

```
public class PizzaStore {
     SimplePizzaFactory factory;

     public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
     }

      Pizza orderPizza(String type) {
           Pizza pizza = factory.createPizza(type);
           pizza.prepare();
           pizza.bake();
           pizza.cut();
           pizza.box();
           return pizza;
      }
      // other methods go here
}
```

Factory



This is a SimpleFactory, which is more of an idiom than a full pattern.

Factory

Now let's expand:

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("veggie");

ChicagoPizzaFactory chicagoFactory =
   new ChicagoPizzaFactory();
PizzaStore chicagoStore =
   new PizzaStore(chicagoFactory);
chicagoStore.order("veggie");
```

...and likewise for `CaliforniaPizzaFactory`, etc.

How do we ensure that all the different stores are consistent?

Factory

PizzaStore revisited:

```
public abstract class PizzaStore {
     public Pizza orderPizza(String type) {
          Pizza pizza;
          pizza = createPizza(type);  // it's back :-)
          pizza.prepare();
          pizza.bake();
          pizza.cut();
          pizza.box();
          return pizza;
      }
      protected abstract Pizza createPizza(String type)
      // other methods go here
}
```

In this formulation, the PizzaStore must be subclassed — and each subclass must define its own createPizza() method.

Factory

Consider the NY-style createPizza() method:

```
public Pizza createPizza(String type) {
    if (type.equals("cheese")) {
        return new NYStyleCheesePizza();
    } else if (type.equals("pepperoni")) {
        return new NYStylePepperoniPizza();
    } [...other types go here...]
}
```

...and likewise for Chicago-style:

```
public Pizza createPizza(String type) {
    if (type.equals("cheese")) {
        return new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni")) {
        return new ChicagoStylePepperoniPizza();
    } [...other types go here...]
}
```

Factory

The NYPizzaStore class contains only the createPizza method (which it must define, since it's abstract; the orderPizza() method is inherited from the base class PizzaStore):

```
public class NYPizzaStore extends PizzaStore
{
   public Pizza createPizza(String type)
   {
        if (type.equals("cheese")) {
           return NYStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
           return NYStylePepperoniPizza();
        }  [...other types go here...]
        else {
           return null;
        }
   }
}
```

...and likewise for the ChicagoPizzaStore class.

Design Guidelines

- No variable should hold a reference to a concrete class.
  - If you use `new`, you'll be holding a reference to a concrete class; to avoid that, use a factory.
- No class should derive from a concrete class.
  - If you derive from a concrete class, you're depending on a concrete class. Derive from an interface or abstract class instead.
- No method should override an implemented method of any of its base classes.
  - If you override an implemented method, your base class wasn't really an abstraction.

Note that it's **impossible** to follow ALL of these suggestions ALL of the time! ...but like any rule, the most important thing about understanding them is knowing WHY and WHEN to break them.

Abstract Factory

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

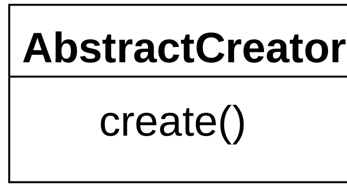
Translation:

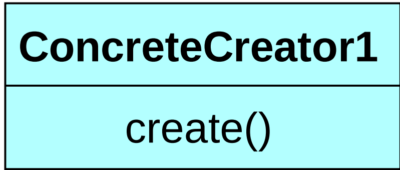
Instantiate **groups** of related objects, without using `new` directly — wait until run-time to decide what kinds of objects to instantiate

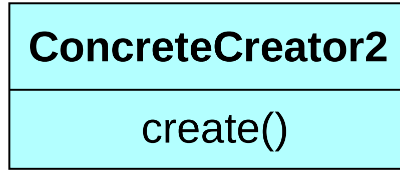
Compare vs. Factory Method:

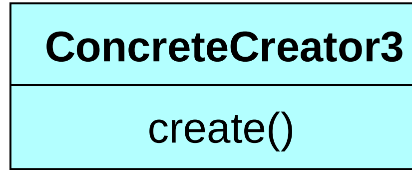
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory lets a class defer instantiation to subclasses.

Factory, Revisited

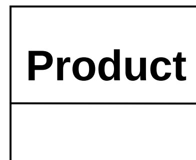








...

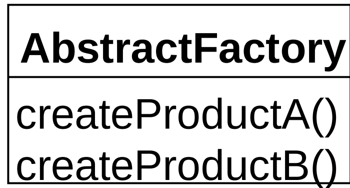


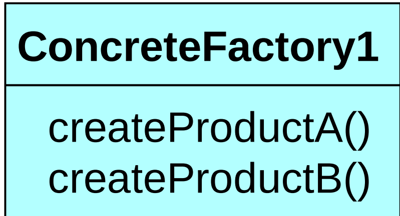
...

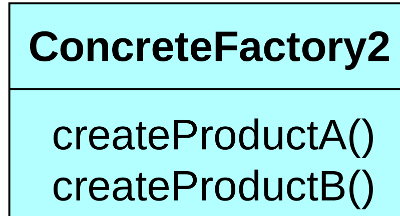
...

... ...

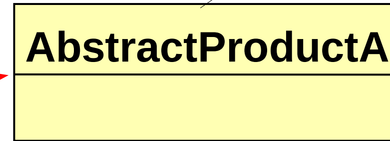
Abstract Factory

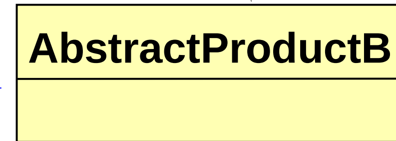






...





...

...

Abstract Factory

- As the diagrams show, the principal difference between Factory (Method) and Abstract Factory is that Abstract Factory has a separate abstraction for each **family** of related products.

- Factory (Method) doesn't need the extra layer of abstraction because it typically creates only one (product) object at a time (*e.g.* one pizza per order).

- Factory (Method) uses inheritance to create objects, whereas Abstract Factory uses composition.

- Typically, an Abstract Factory actually uses Factory (Method) internally!

Abstract Factory

- To see Abstract Factory in action, let's return to our favourite pizza stores:
  - Suppose we want to ensure that all the ingredients used in every store are consistent (fresh, high quality, etc.).
  - But suppose we also want to allow for regional differences (*e.g.* New York uses Marinara sauce but Chicago uses red plum tomato sauce; New York uses mushrooms on vegetarian pizza, but Chicago uses spinach instead, etc.).
  - In this situation, each combination of pizza type and style uses the same **categories** of ingredients, but not the **same** ingredients.

Abstract Factory

Let's start by (surprise! :-) creating an interface:

```
public interface PizzaIngredientFactory
{
   public Dough createDough();
   public Sauce createSauce();
   public Cheese createCheese();
   public Veggies[] createVeggies();
   public Pepperoni createPepperoni();
   public Clams createClam();
}
```

Now we can implement this interface for each region.

Abstract Factory

Now let's implement the New York style factory:

```
public class NYPizzaIngredientFactory implements
         PizzaIngredientFactory {
   public Dough createDough() {
      return new ThinCrustDough();  // thick crust for Chicago
   }
   public Sauce createSauce() {
      return new MarinaraSauce();  // red plum sauce for Chicago
   }
   public Veggies[] createVeggies() {
      Veggies v = { new Garlic(), new Onion(),
                    new Mushroom(), new RedPepper() };
      return v;  // other regions use different vegetables
   }
   // ...and similarly for createCheese(), createPepperoni()
   //    and createClam()
}
```

Abstract Factory

The next step is to modify the Pizza class:

```
public abstract class Pizza {
    String name;  Dough dough;  Sauce sauce;
    Veggies veggies[];  Cheese cheese;
    Pepperoni pepperoni;  Clams clam;

    abstract void prepare();
    void bake() { ... };
    void cut() { ... };
    void box() { ... };
    void setName(String name) { this.name = name; }
    String getName() { return name; }
}
```

The important changes are the addition of the ingredients as data members, plus the fact that prepare is now abstract — it will be implemented by each concrete pizza store.

Abstract Factory

Next, let's look at a concrete pizza store:

```
public class NYPizzaStore extends PizzaStore{
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();
        if (item.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York style cheese pizza");
        } else if (item.equals("veggie")) {
            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York style veggie pizza");
        }
        // [ ...similar code for other pizza types ]
        return pizza;
    }
}
```

Abstract Factory

Tracing an order through to completion: a New York style cheese pizza is born:

- First we need an appropriate pizza store:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

- Next, we can take an order:

```
nyPizzaStore.orderPizza("cheese");
```

- A pizza must be created (in orderPizza()):

```
Pizza pizza = createPizza("cheese");
```

- The new pizza needs ingredients (in createPizza()):

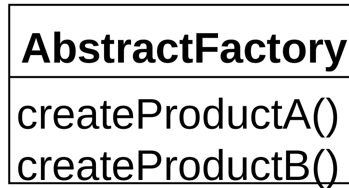
```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

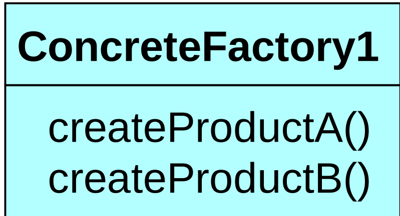
- The pizza must be prepared:

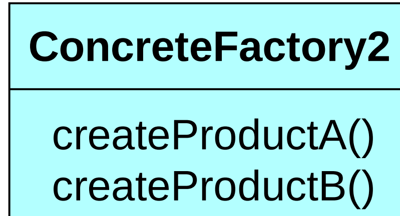
```
void prepare() {
   dough  = factory.createDough();  // thin crust
   sauce  = factory.createSauce();  // Marinara
   cheese = factory.createCheese(); // Reggiano
}
```

- Finally we're ready for the pizza to be baked, cut and boxed.

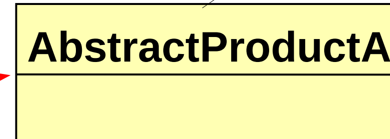
Abstract Factory

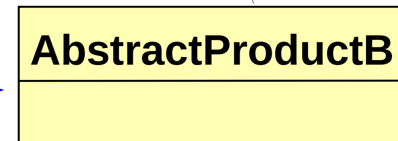






...





...

...

Proxy Pattern

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

Translation:

Pay no attention to that man behind the curtain. :-)

Translation of the translation:

Communicate between some other object and the user, while pretending to be the other object.

Proxy Pattern

Recall the dictionary definition of the word 'proxy':

1. A person authorized to act for another; an agent or substitute.
2. The authority to act for another.
3. The written authorization to act in place of another.

(source: The American Heritage Dictionary, as found by a Google search on the word 'proxy' :-)

Proxy Pattern

Proxies come in several flavours:

- A **remote** proxy is a local stand-in for a non-local object.

  The typical web proxy is a good example, although technically "remote" could just mean "in another address space on the same machine". Some people call this kind of proxy an **ambassador**.

- A **virtual** proxy creates an expensive object on demand.

  The classic example is a document processor, in which images are loaded only when actually required.

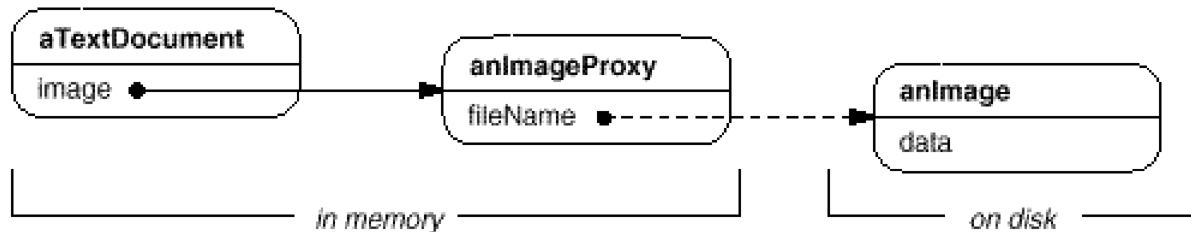
- A **protection** proxy enforces (security) access rights to another object.

  For example, a good secretary or receptionist. :-)

- A **smart reference** is replacement for an ordinary pointer, that supplements the pointer's capabilities.

  Possibilities include reference counting, managing persistent objects, (synchronization) locking, etc.

(Virtual) Proxy Example



DocumentEditor

Graphic

Draw()
GetExtent()
Store()
Load()

Image

Draw()
GetExtent()
Store()
Load()

imageImp
extent

image

ImageProxy

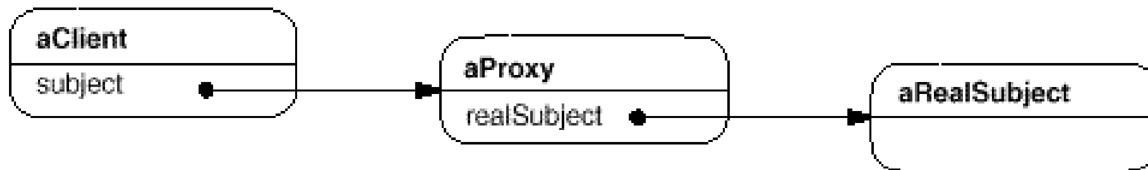
Draw()
GetExtent()
Store()
Load()

fileName
extent

```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

Proxy Structure



Client

Subject

Request()

...

RealSubject

Request()

...

realSubject

Proxy

Request()

...

...

realSubject->Request();

...

Another Virtual Proxy

- From an operating system's point of view, copying a page of memory is expensive, and not always necessary.
- Most modern operating systems implement some form of "copy-on-write" when a new process is created: each page inherited from the creator of the new process are actually **shared** with the creator until either process writes to the page; at that time, the page is copied before the write is allowed to proceed.

Proxy Consequences

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

Proxy Notes

- There are many variants on the general Proxy pattern, but they all have one thing in common: a client invokes a method on some subject, but the method call is intercepted and actually handled by the proxy, usually working together with the real subject.
- Proxy works well with Factory Method: when a client tries to instantiate a subject, the Factory can instantiate the proxy at the same time, and return the proxy instead.
- Factory is similar to the Adapter pattern (coming up next :-), but Adapter **changes** the interface of another object while Proxy deliberately implements the **same** interface.

Adapter Pattern

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

Translation:
Make a square peg actually fit into a round hole. :-)

Adapter in the Real World

- Suppose that you're travelling in Europe with your North American laptop computer. Your battery is getting low, so you pull out your AC adapter to charge it.
- ...but of course you can't plug it in, because Europe has a different standard for AC current, meaning your plug won't fit.
- ...so you end up having to use an adapter for your adapter. :-)
- Note that an adapter between North American and European AC standards doesn't just change the shape of the plug; it also changes the voltage and frequency (110-120 volts at 60 Hz vs. 220-240 volts at 50 Hz).
- ...which is relevant, because software Adapters also cause behavioural change :-)

Adapter Example

- Back to the old drawing editor. :-)   Specifically, suppose you're implementing a drawing editor. Basic shapes such as lines and polygons are easy, but text can be difficult to write.
- ...but suppose that you have (legal :-) access to somebody else's TextView class.
- ...but TextView wasn't designed with your application in mind, and won't fit into your collection of Shape classes because of an incompatible interface.
- As you may surmise, it's Adapter to the rescue. :-)

Adapter Example

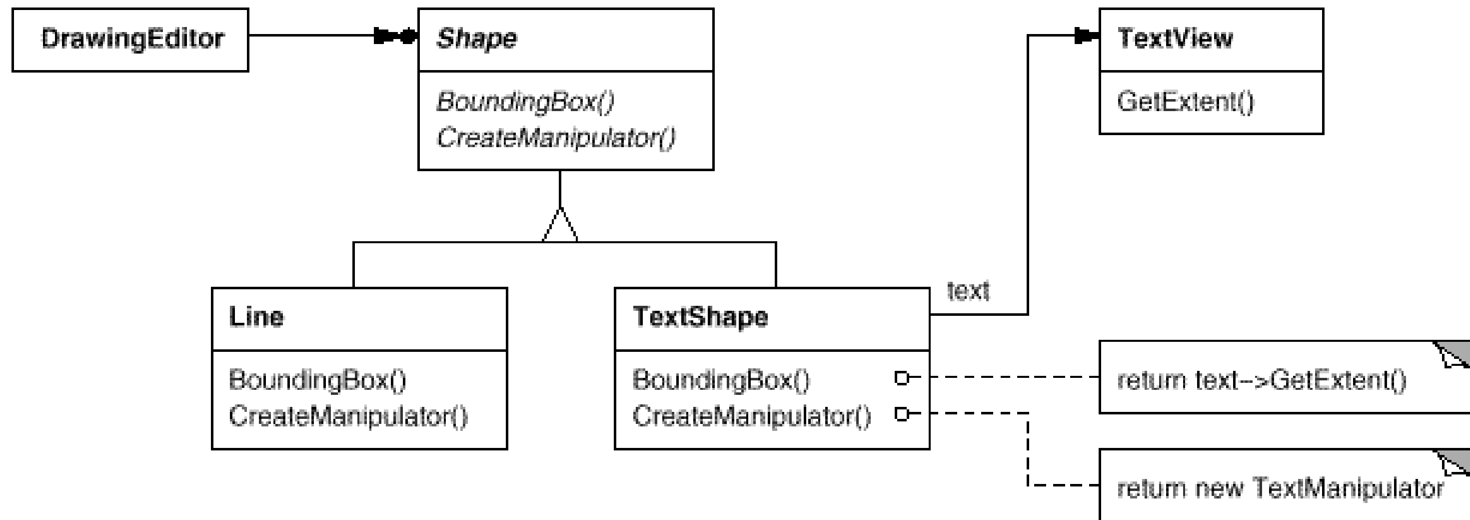
- There are two general approaches in this kind of situation:

  (0) define a TextShape that inherits the interface from Shape and the implementation from TextView

  (1) define a TextShape that contains a TextView instance, and implementing TextShape in terms of the services provided by TextView

- Option (1) is the **class** version of the Adapter pattern, and option (2) is the **object** version of Adapter.

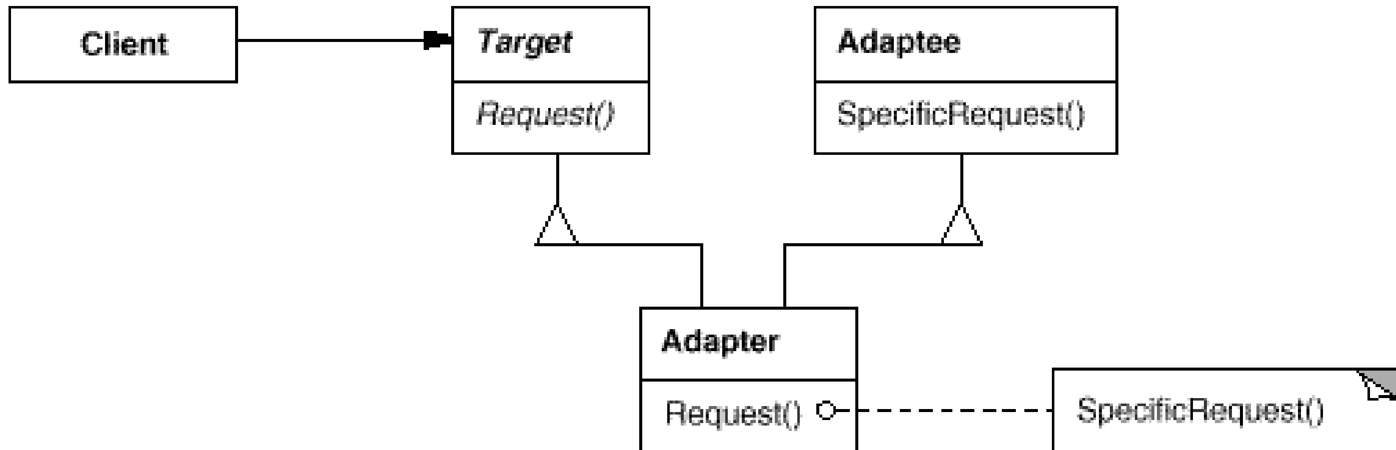
(Class) Adapter Example



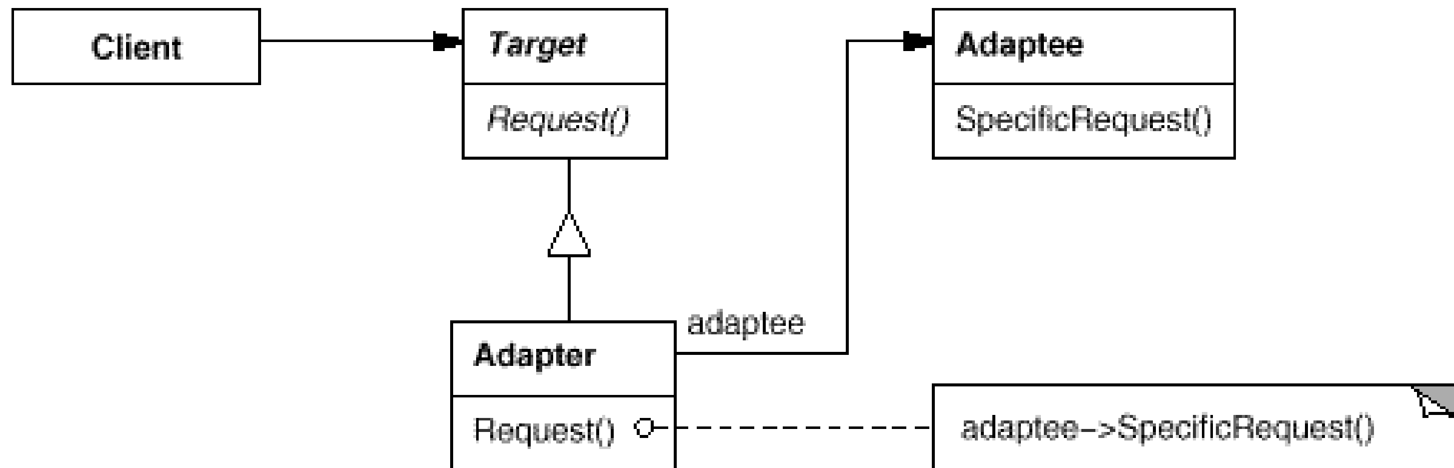
Notice how TextShape inherits from both Shape and TextView (in C++).

In Java, TextShape would have to implement Shape and extend TextView; this is sort of a bit of both styles :-).

(Class) Adapter Structure



(Object) Adapter Structure



(Class) Adapter Consequences

A **class** adapter...

- adapts Adaptee to target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behaviour, since Adapter is a subclass of Adaptee. (Note: this often makes a class adapter easier to write, since less code is required.)
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

(Object) Adapter Consequences

An **object** adapter...

- lets a single Adapter work with many Adaptees — that is, the Adaptee itself and all of its subclasses. The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behaviour. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Visitor Pattern

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

Translation:

Traverse a tree or other data structure and do stuff with each node in turn. :-)

Visitor Example

- Recall the menu/submenu tree we discussed while looking at Composite.
- Suppose that restaurant customers have questions about the ingredients and nutrition value of the various menu items. To answer them, we could add methods such as getHealthRating(), getCalories(), getProtein(), etc. — but this would complicate the interface enormously, making it much more fragile with respect to future change.
- Instead, suppose we add a single getState() method. A client could then traverse the structure, calling getState() on each item and interpreting the result.
- Yes, this does imply that Visitor is most useful in conjunction with Composite. ;-)

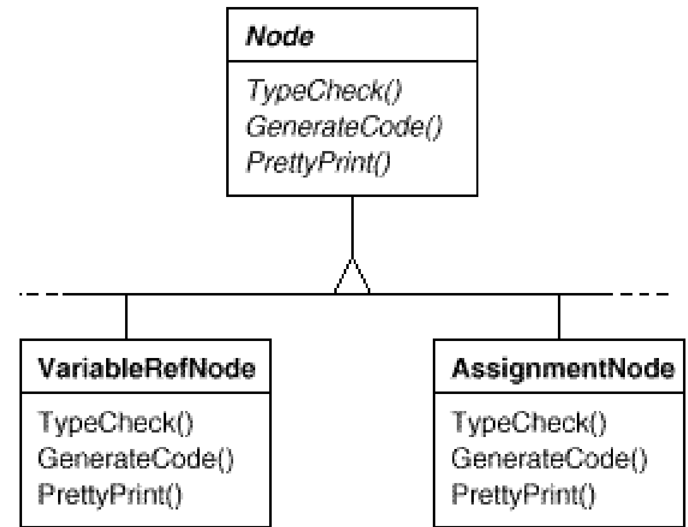
To Visit, or Not to Visit?

- Visitor clearly has two big drawbacks right up front:
  - It breaks encapsulation of the classes in the structure being visited.
  - It makes changes to the structure more difficult, since the shape of the structure is now known and used from outside.
- Despite these problems, there are also some advantages:
  - Operations can be added to the structure without having to modify the structure itself.
  - All the operations are now centralized in one place.
  - Adding new operations is easier than it would be otherwise.

Another Visitor Example

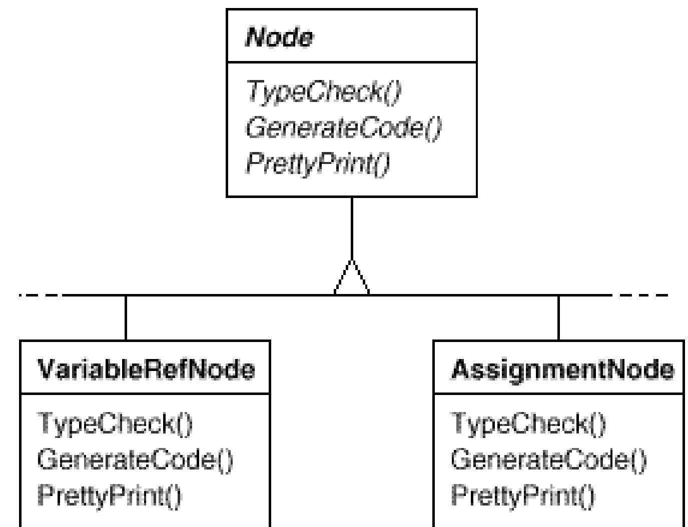
Consider the parse tree of a hypothethical compiler. There will probably be different types of nodes corresponding to different types of source language entities.

For example, perhaps there will be a node type for assignment statements, one for variable references, one arithmetic expressions, etc., as in this picture:



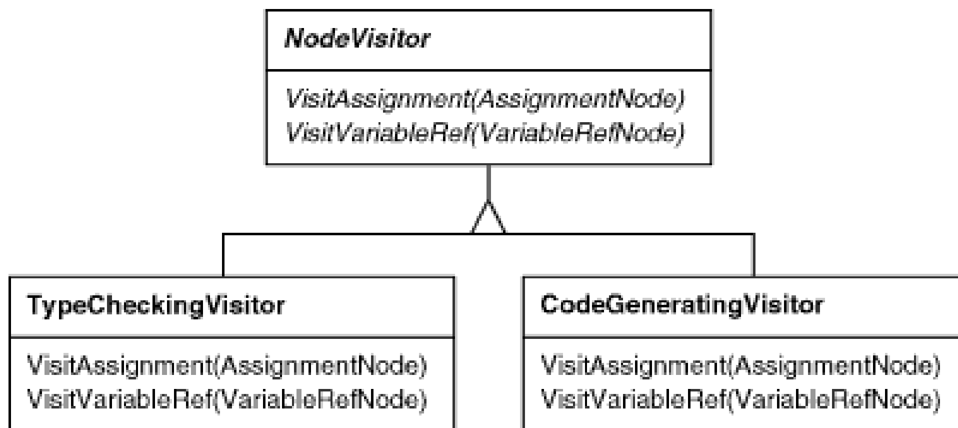
Another Visitor Example

As with the menu example, the problem here is the proliferation of methods in each node. What happens if these need to be modified, or new ones added?

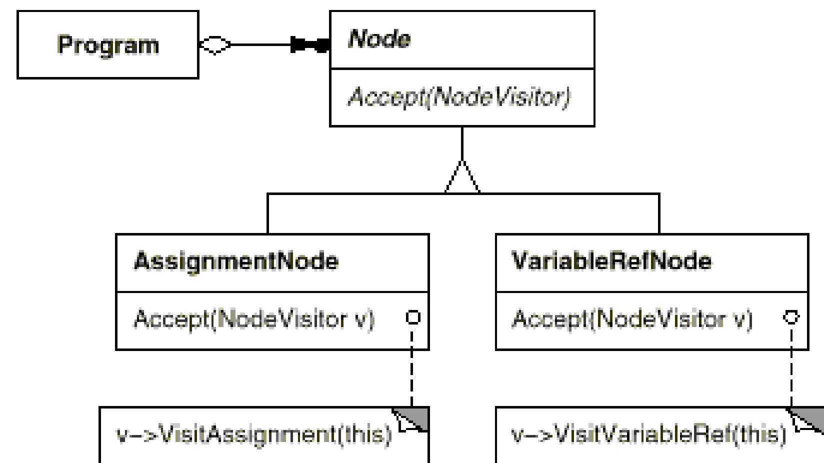


Another Visitor Example

The Visitor pattern suggests creating a separate interface for the node operations:

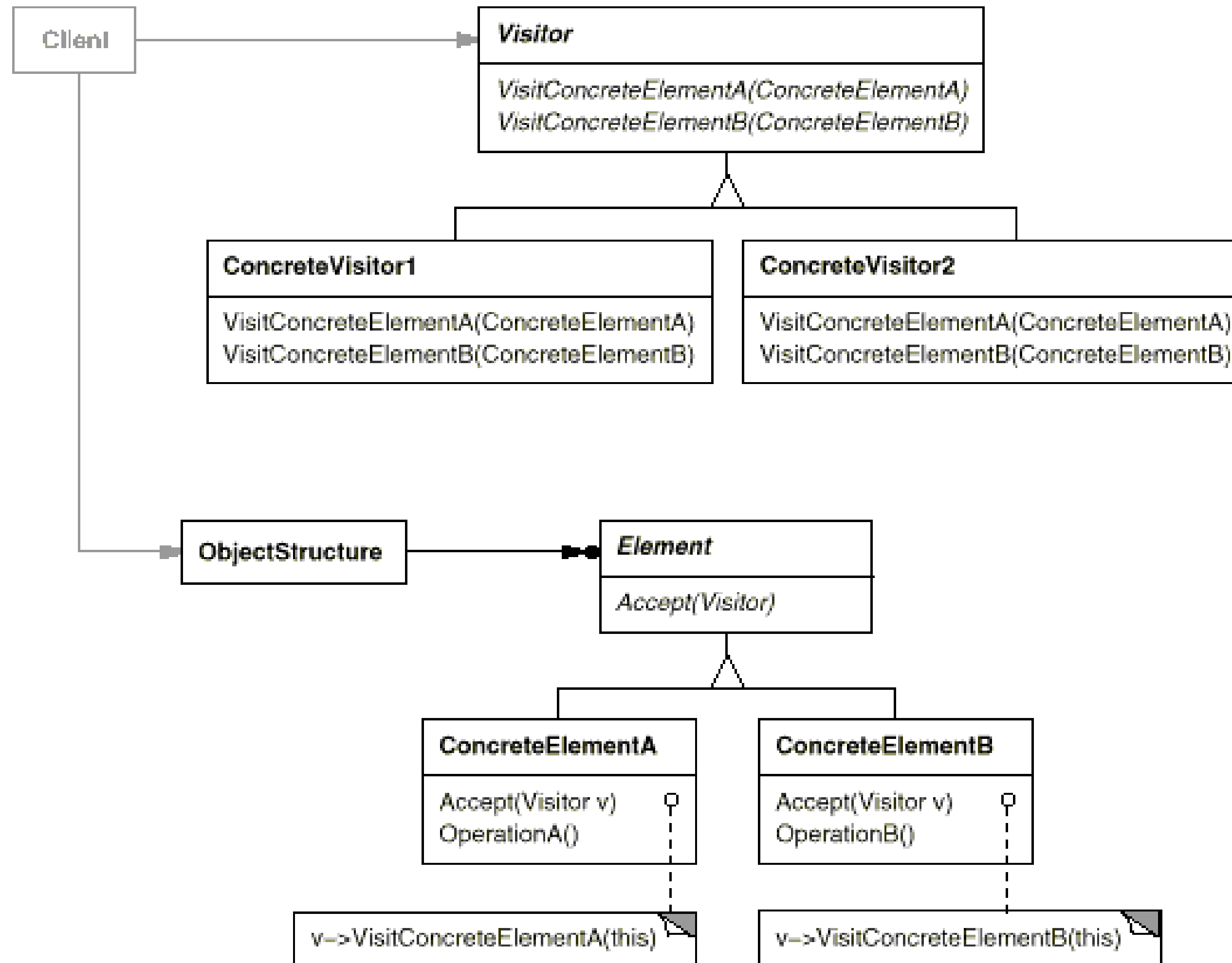


The original nodes would now look like this:



To use this structure, we traverse the nodes and invoke the Accept() method in each one. The traversal is usually done either by the object structure itself or by an iterator.

Visitor Structure



Visitor Applicability

Use Visitor when...

- an object structure contains many classes with different interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure and one wants to avoid "polluting" their classes with these operations.
- **the classes defining the object structure rarely change, but you often want to define new operations over the structure.** Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly.

Visitor Consequences

- Visitor pattern makes adding new operations easy.
- A visitor gathers related operations and separates unrelated ones.
- Adding new Concrete Element classes is hard.
- Visiting can occur across class hierarchies (compared to iterators which can only operate within a single class and its descendants at a time).
- Visitors can accumulate state information as they traverse the object structure.
- Visitor breaks the object structure's encapsulation.

Decorator

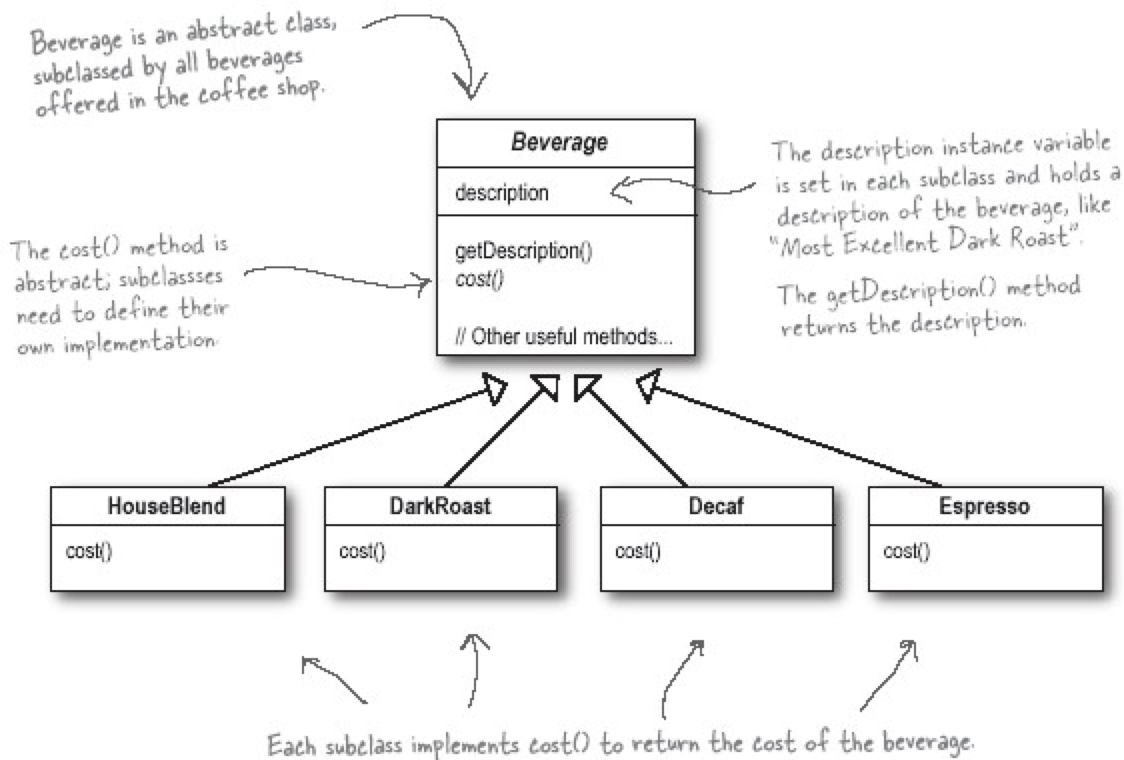
Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Translation:

Add additional capabilities to an object, without changing the class it belongs to.

Decorator

"Starbuzz Coffee" represents their products using the following classes:

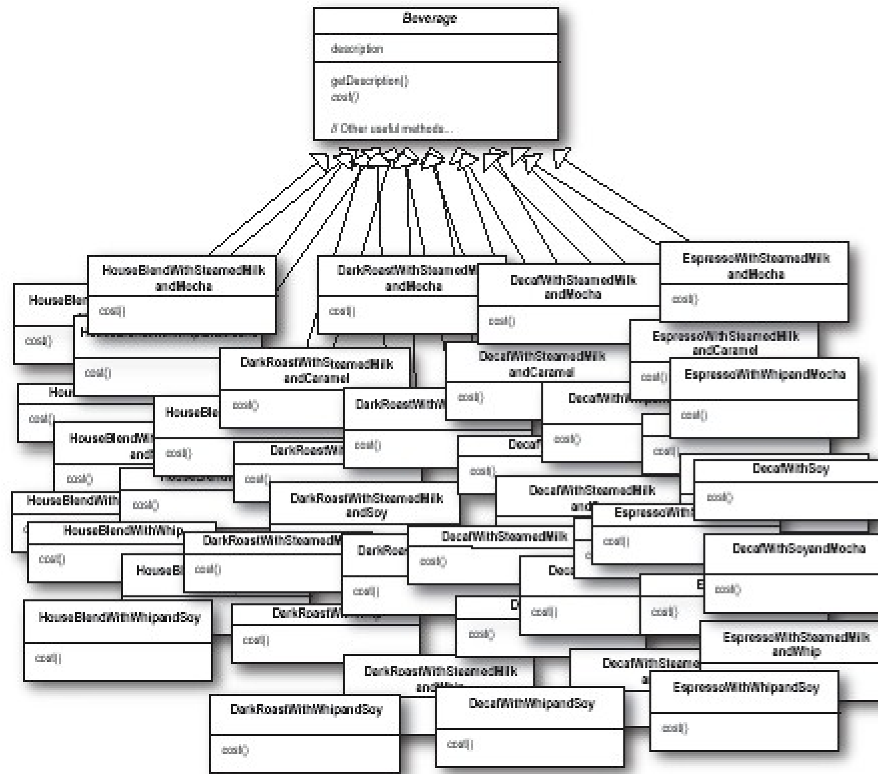


Now they want to model the condiments (milk, soy, mocha, etc.) that they offer.

How would you do that?

Decorator

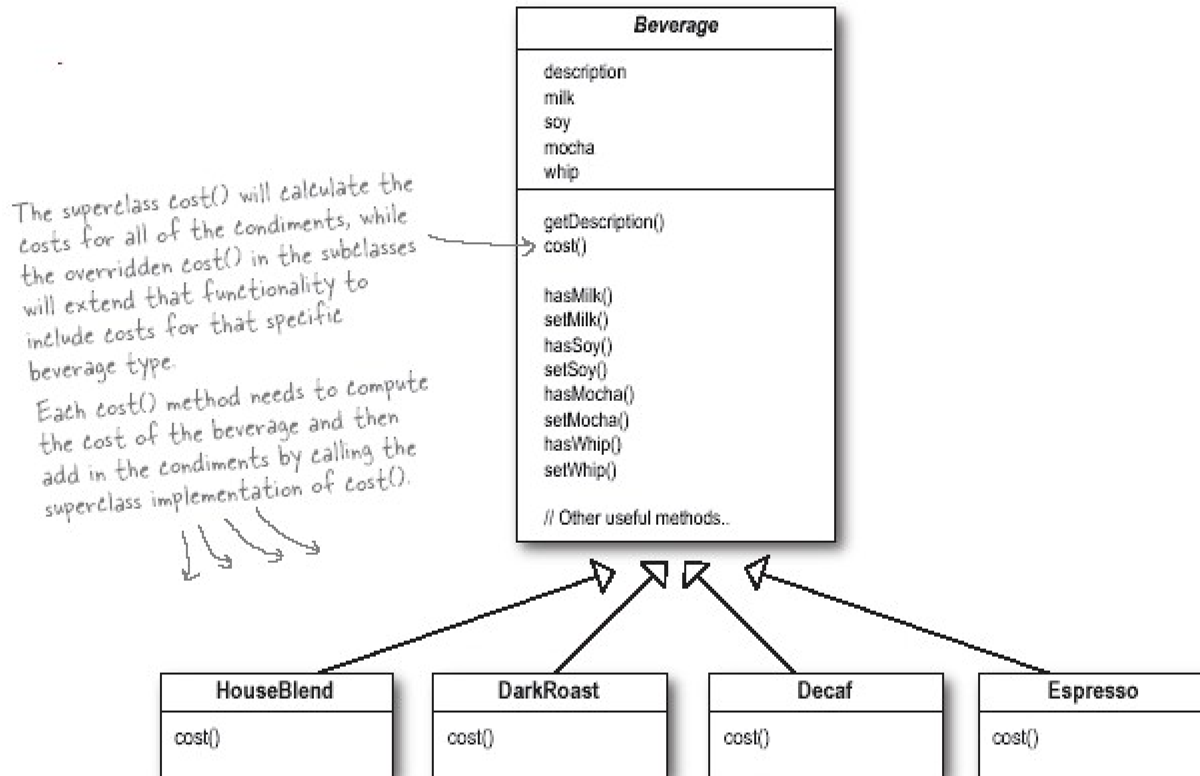
Here's their first attempt:



Each cost method computes the cost of the coffee along with the other condiments in the order.

Decorator

Here we have boolean instance variables with access methods to record the presence of each condiment:



This looks better. Is it good enough?

Decorator

Using inheritance this way certainly reduces the number of classes.But...

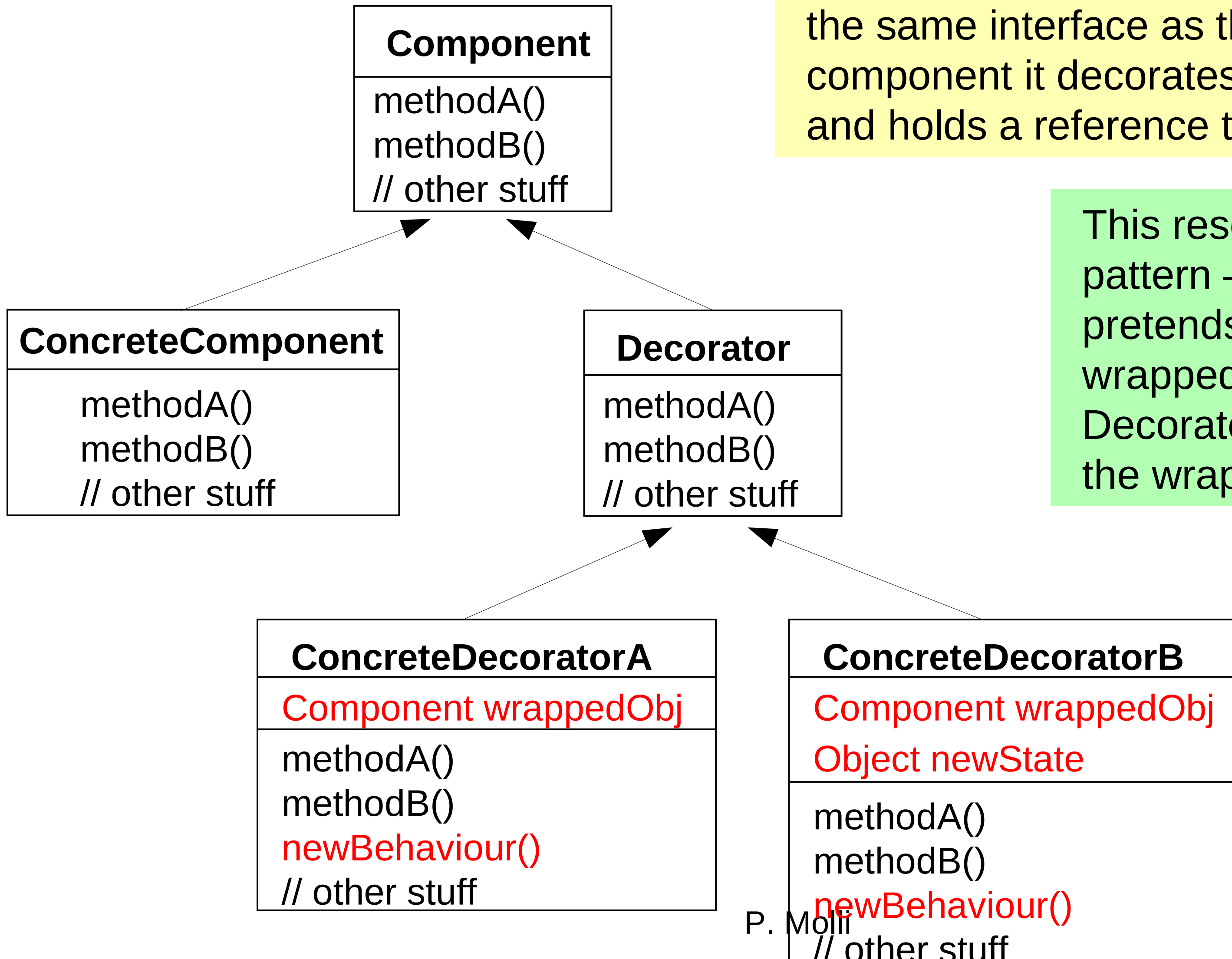
- What happens when the price of a condiment changes?
- How do we add a new condiment?
- How do we add a new beverage?
- What if a given condiment isn't appropriate for a new beverage? It would still be inherited.
- What if a customer wants a double mocha?

Oops. Looks like it's time to introduce the Decorator pattern. :-)

Decorator Structure

A Decorator implements the same interface as the component it decorates, and holds a reference to it.

This resembles the Proxy pattern — but Proxy pretends to **be** the wrapped object, while Decorator **supplements** the wrapped object.



Decorator Notes

- Decorators share the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Because the decorator shares the supertype (*i.e.* implements the same interface), you can pass a decorator anywhere the wrapped object is expected.
- *The decorator adds its own behaviour before and/or after invoking the same method in the wrapped object.*
- Objects can be decorated at run-time, so the choice of decorators for a given object can change during execution of the program.

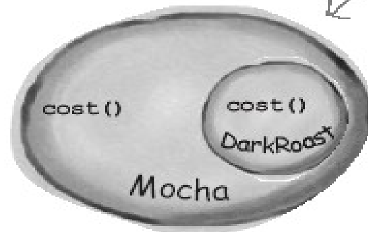
Decorator in Action

Constructing a drink order with Decorators

1 We start with our DarkRoast object.

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

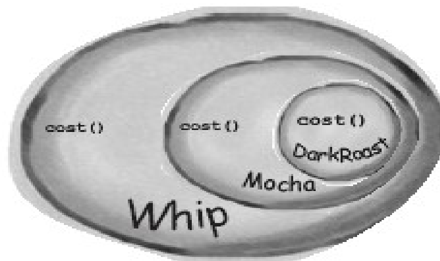
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

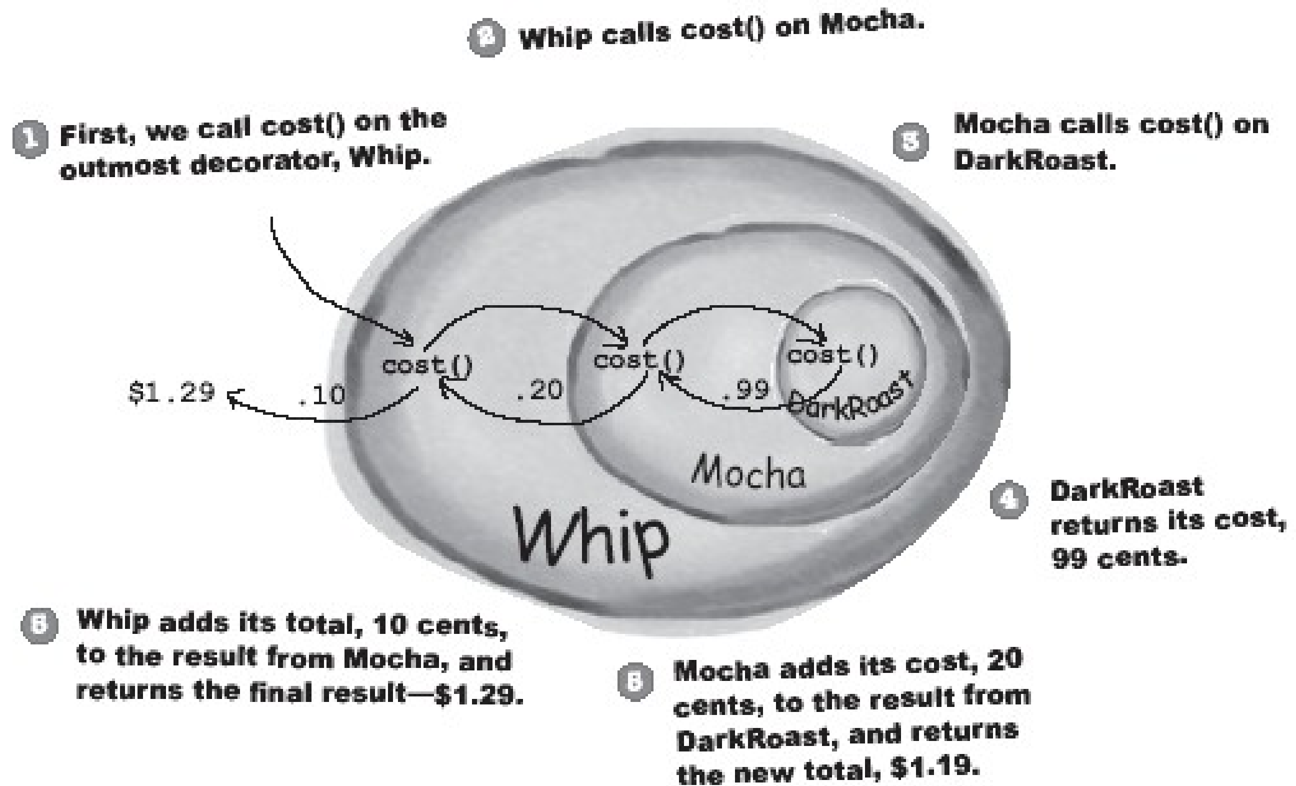


Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

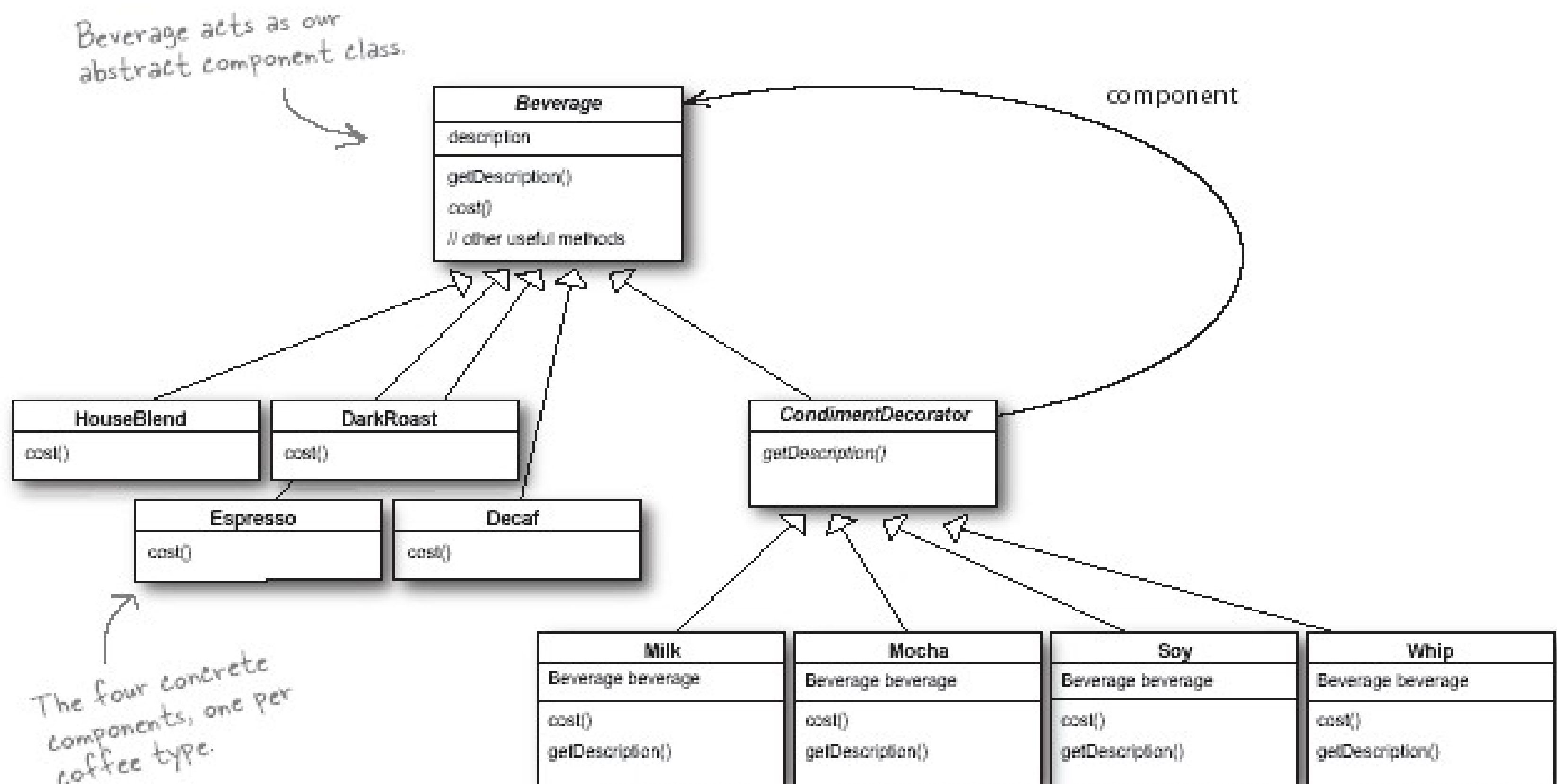
Decorator in Action

❹ **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



Decorated Drinks

Here are the Starbuzz beverage classes reworked using Decorator:



Decorator Consequences

More flexible than static inheritance. Responsibilities can be added or removed at run-time, and the same property can even be added more than once (*e.g.* double mocha, or a double window border).

Avoids feature-laden classes high up in the hierarchy. This helps to avoid circle-ellipse problems; more specifically, it allows the design of simple classes that can be extended flexibly without having to modify the base of a family of classes.

A decorator and its component aren't identical. As we've seen, be careful about type-based assumptions and tests when using Decorator.

Lots of little objects. As with the Java IO classes, Decorator often results in many small classes that look alike; although they differ in how they're connected, they all have the same instance variables and methods. This can be confusing to learn and hard to debug.

Facade

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.

Translation:

Provide the moral equivalent of a batch file or shell script. :-)

Facade Motivation

- Steps involved in compiling a program include preprocessing, syntax analysis, code generation, linking and loading. ...but typical command-line invocation is done in (only) one step: `g++ -o foo foo.cpp`
- Consider the steps involved in getting ready to watch a movie on your home theater system:

(1) Turn on the popcorn popper.

(2) Start the popper popping.

(3) Dim the lights.

(4) Turn on the TV.

(5) Turn on the control unit.

(6) Set the input to DVD.

(7) Set the audio mode to surround sound.

(8) Set the desired volume level.

(9) Turn on the DVD player.

(10) Start the DVD player.

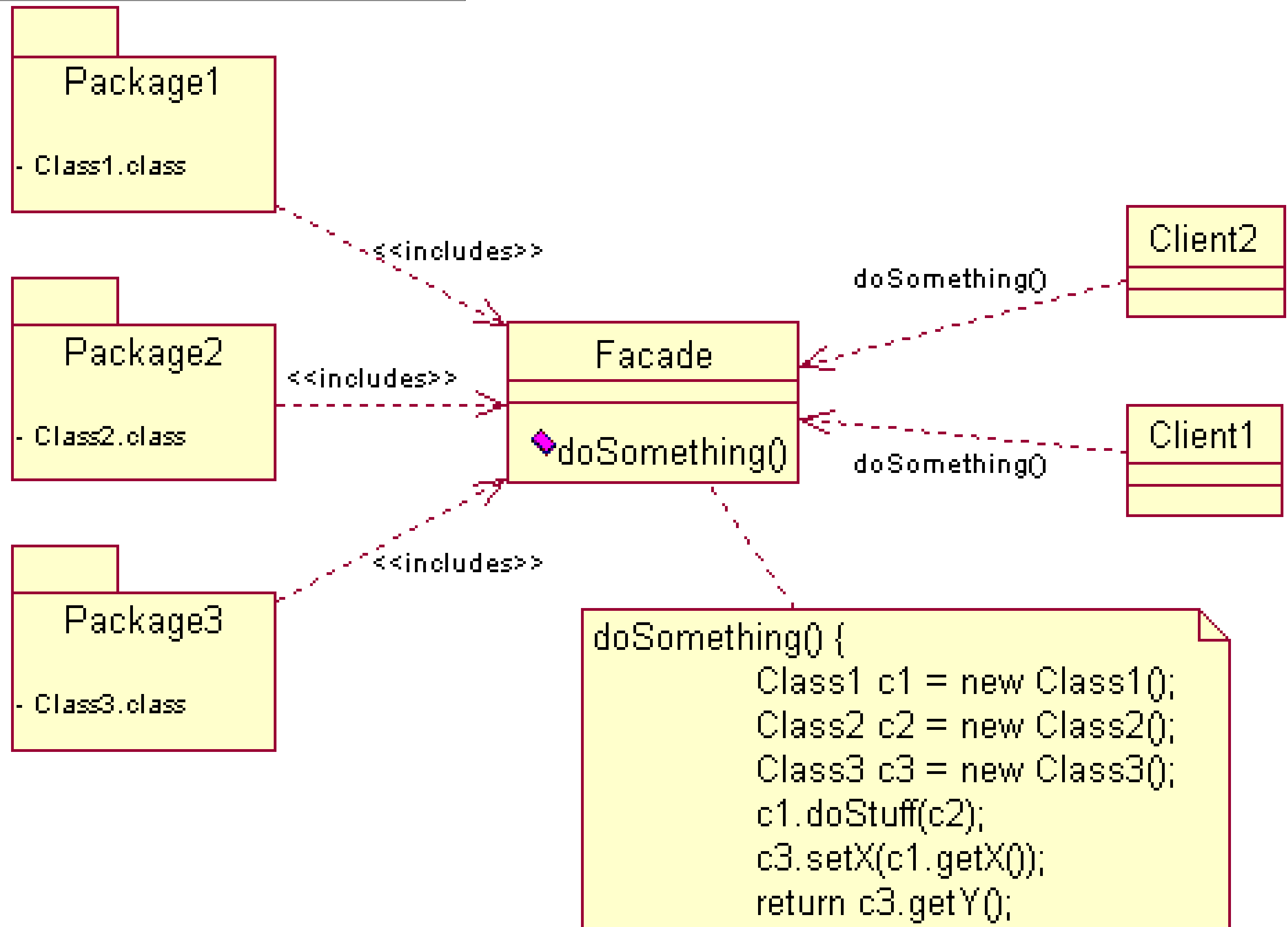
(11) Collect the popcorn.

(12) Sit down and enjoy the show.

Wouldn't it be nice to be able to do all that with one press of a remote control button? :-)

Facade Structure

The (simple) Facade interface sits between the clients and the classes which do the real work.



Facade Consequences

Facade offers the following benefits:

- *It shields clients from subsystem components,* therey reducing the number of objects that clients deal with and making the subsystem easier to use.
- *It promotes weak coupling between the subsystem and its clients.* If the subsystem is modified, the Facade will probably have to be modified also, but the clients probably won't.
- *It doesn't prevent applications from accessing subsystem components directly if they need to.* Sometimes a client will need a service that the Facade doesn't provide, but this isn't a problem because the subsystem's services are still available.

It's All a Matter of Intent

We've now seen three patterns that seem very similar. The major difference between them is in how they're intended to be used:

pattern	purpose
Adapter	Convert one interface to another.
Decorator	Don't alter an interface, but add responsibility.
Facade	Make an interface simpler to use.