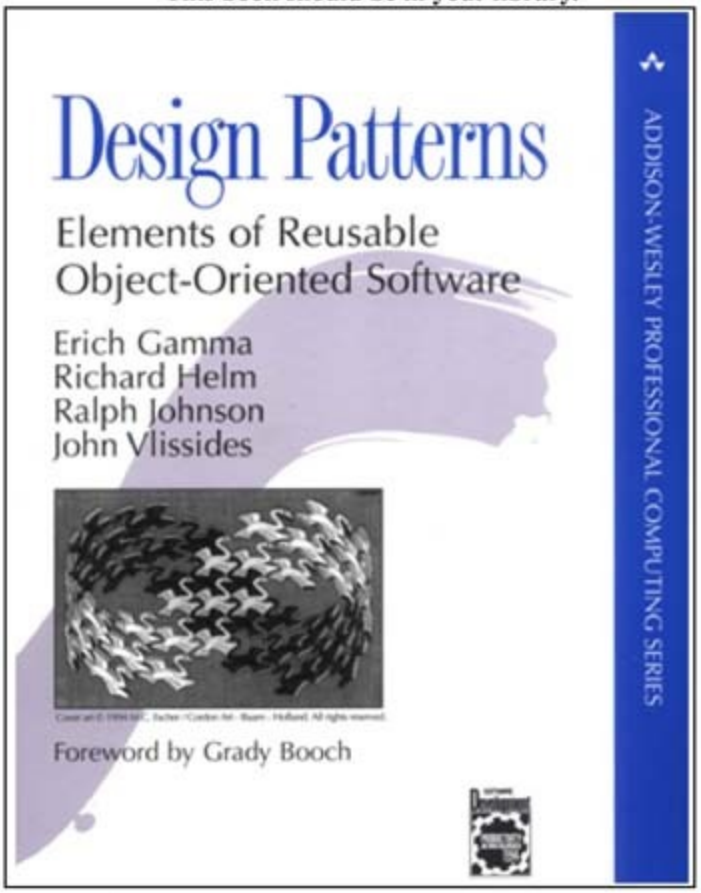
Design Patterns

The definitive work on Design Patterns

This book should be in your library.



Gamma, Helm, Johnson and Vlissides,
also known as the *Gang of Four* or just *GoF*

But if you really want to understand Design Patterns...

This book should be in your brain!

Freeman, Eric and Freeman, Elisabeth
***Head First Design Patterns,* O'Reilly, 2004**

All the Java source code for this book is available from
http://www.headfirstlabs.com/books/hfdp

Design Patterns

Each of these programmers has a copy of the GoF Design Patterns book on their book shelf, ... somewhere.

They are all hoping to get a job developing video games for Flibinite Software, Inc.

Each of these software engineers understands Design Patterns and how to incorporate them into their software designs.

Their company has sent them on an all expences paid vacation to Hawaii because their software designs have more than tripled the company income in the last year.

Get the hint?!!!

Design Patterns

What are they?

- General, reusable solutions to commonly occurring problems in software design.
- A description or template for how to solve a problem that can be used in many different situations.
- Ways of organizing code to increase efficiency and provide greater Object Oriented Design.

What are they not?

- They are not a finished design that can be transformed directly into code.
- Algorithms are not design patterns because they solve computational problems not design problems.
- They are not trivial.
- They are not represented by any features that are built into a programming language, e.g. templates are not design patterns.

Design Patterns

A Brief History

- 1977-1979 - Originated as an architectural (as in building design) concept by Christopher Alexander
- 1987 - Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming. Presented their results at the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Conference that year.
- 1987-1994 - The ideas gained in popularity.
- 1994 - the GoF published the Design Patterns book.
- 1994 - the first Pattern Languages of Programming conference was held.
- 1995 - the Portland Pattern Repository was set up to document design patterns.

Design Patterns

Advantages of Design Patterns

- Speed up the development process
- Helps to prevent subtle issues that can cause major problems
- Improves readability of the code
- Provides patterns for solving problems
- Provides a common language for designers

Disadvantages of Design Patterns

- In order to achieve flexibility design patterns usually add additional levels of indirection which in some cases may complicate the resulting designs and hurt application performance.

Categories of Design Patterns

Creational Patterns

Focuses on how objects are created.
Often involves isolating the details of object creation in such a way that you don't have to make major changes to your code when you have to create a new type of object.

Abstract factory, Builder, Factory method, Prototype, Singleton

Structural Patterns

Focuses on how objects are interconnected.
Attempts to ensure that changes in the system don't require changes in those connections.
These patterns are often dictated by project constraints.

Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

Behavioral Patterns

Focuses on objects that handle particular types of behavior.
Encapsulates processes such as interpreting a language, fulfilling a request, moving through a sequence (as an iterator), or implementing an algorithm.

Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

On Learning Design Patterns

Learn the basic organization/functionality/makeup of each pattern, then look for places in your code where you can apply them.

From the Code Guru

Design Pattern Definitions from the GoF Book

The Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Behavioral Patterns

- The Strategy Pattern
- The Observer Pattern
- The Command Pattern
- The Template Method Pattern
- The Iterator Pattern
- The State Pattern
- The Chain of Responsibility Pattern
- The Interpreter Pattern
- The Mediator Pattern
- The Memento Pattern
- The Visitor Pattern

Structural Patterns

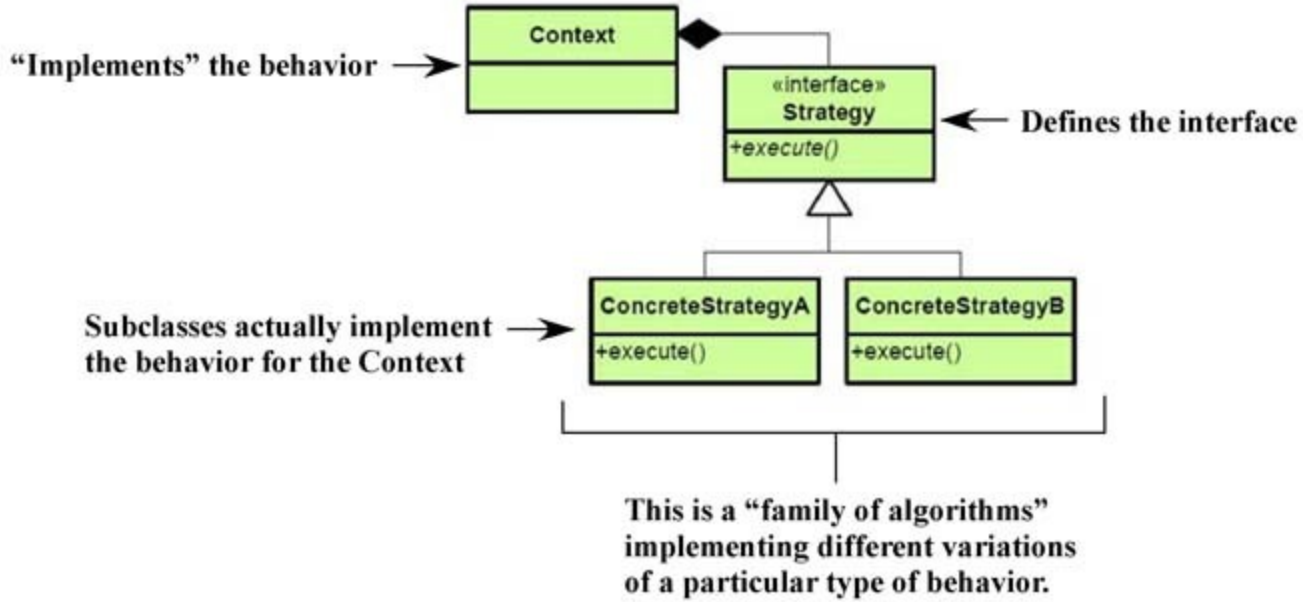
- The Decorator Pattern
- The Adapter Pattern
- The Facade Pattern
- The Composite Pattern
- The Proxy Pattern
- The Bridge Pattern
- The Flyweight Pattern

Creational Patterns

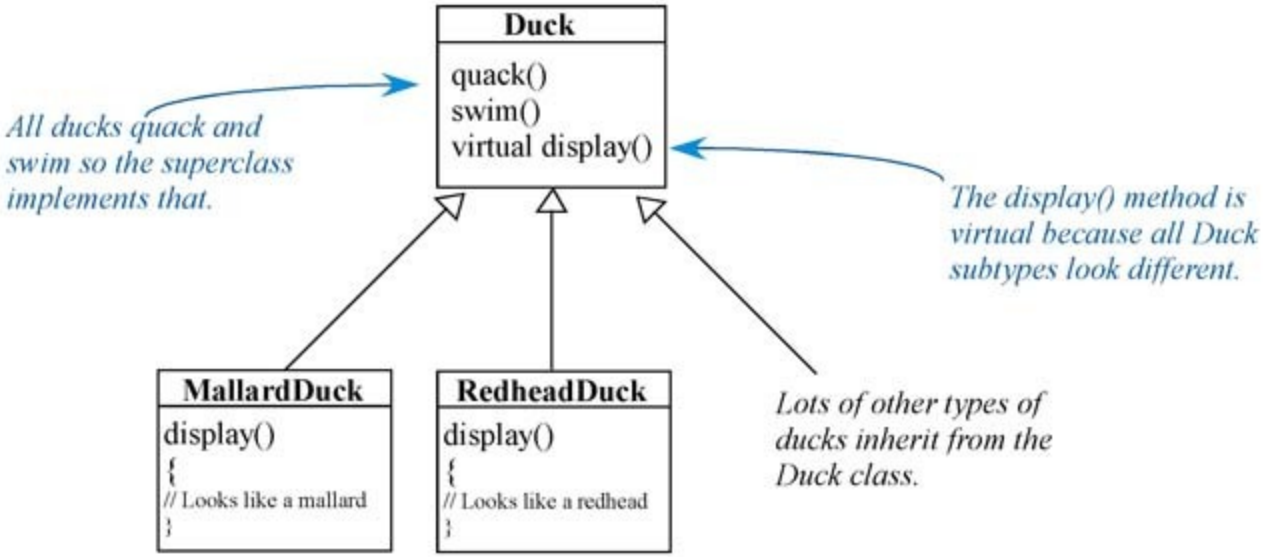
- The Factory Method Pattern
- The Abstract Factory Pattern
- The Singleton Pattern
- The Builder Pattern
- The Prototype Pattern

Design Patterns: Strategy
Quick Overview

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

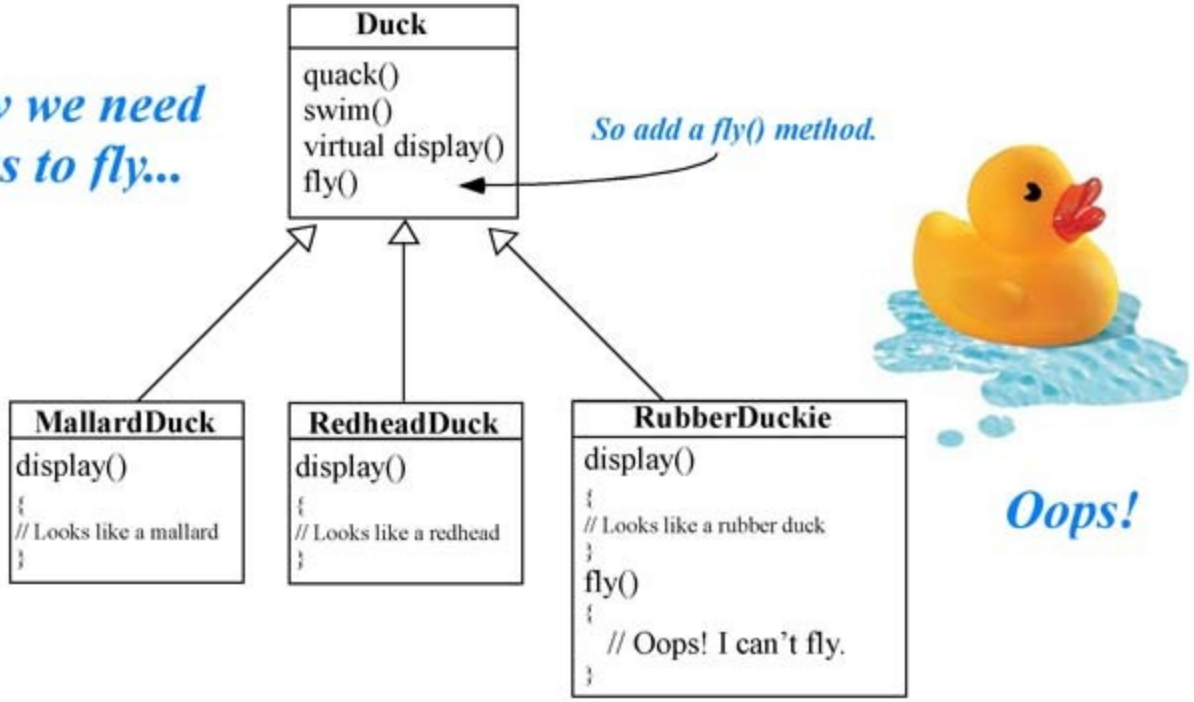


Design Patterns: Strategy



Design Patterns: Strategy

But, now we need the ducks to fly...



Oops!

So what's a programmer to do?

Design Patterns: Strategy

The one constant in software development.

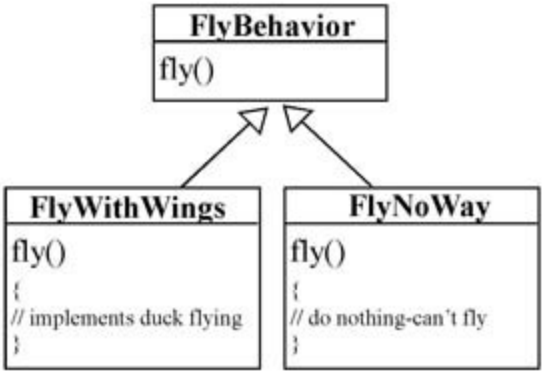
Design Principles

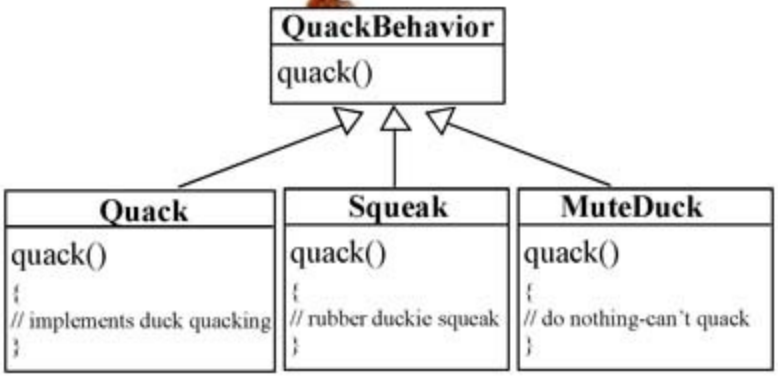
- *Identify the aspects of your application that vary and separate them from what stays the same.*

Remember me?

Design Patterns: Strategy

Encapsulate the part that changes





Design Principles

FlyBehavior and QuackBehavior are the interfaces

- ***Program to an interface, not an implementation.***

FlyWithWings, FlyNoWay Quack, Squeak, MuteDuck are the implementations.

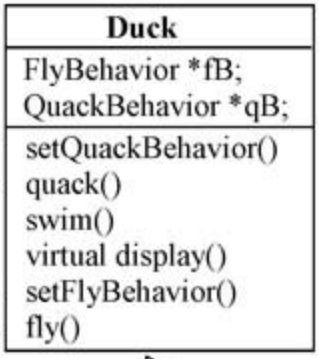
Design Patterns: Strategy

```
// Set Quack behavior for this Duck
void Duck::setQuackBehaivor(
    QuackBehavior Q)
{
    qB = Q;
}
```

```
// Quack Duck!
void Duck::quack()
{
    qB->quack();
}
```

```
// Set Fly behavior for this Duck
void Duck::setFlyBehaivor(
    FlyBehavior F)
{
    fB = F;
}
```

```
// Fly Duck
void Duck::fly()
{
    fB->fly();
}
```



*Duck now **Delagates** its' flying and quacking to instances of FlyBehavior and QuackBehavior so it doesn't care what type of quack or fly behavior it has.*

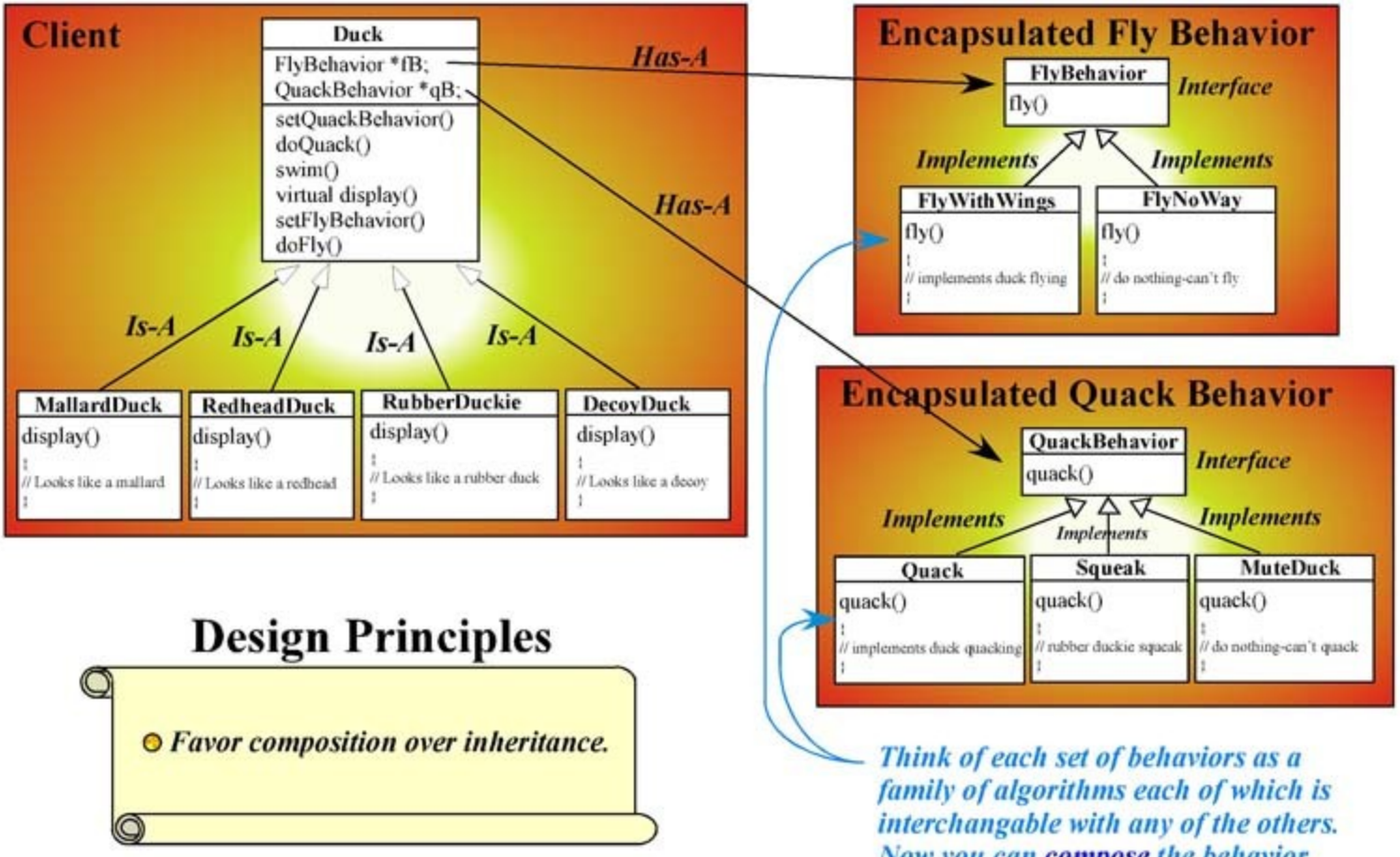
The sub-classes handle that.

The sub-class instantiates its' own default behaviors, but this can be changed through the Duck class setter methods.

class MallardDuck : Duck

```
MallardDuck::MallardDuck()
{
    qB = new Quack();
    fB = new FlyWithWings();
}
```

Design Patterns: Strategy

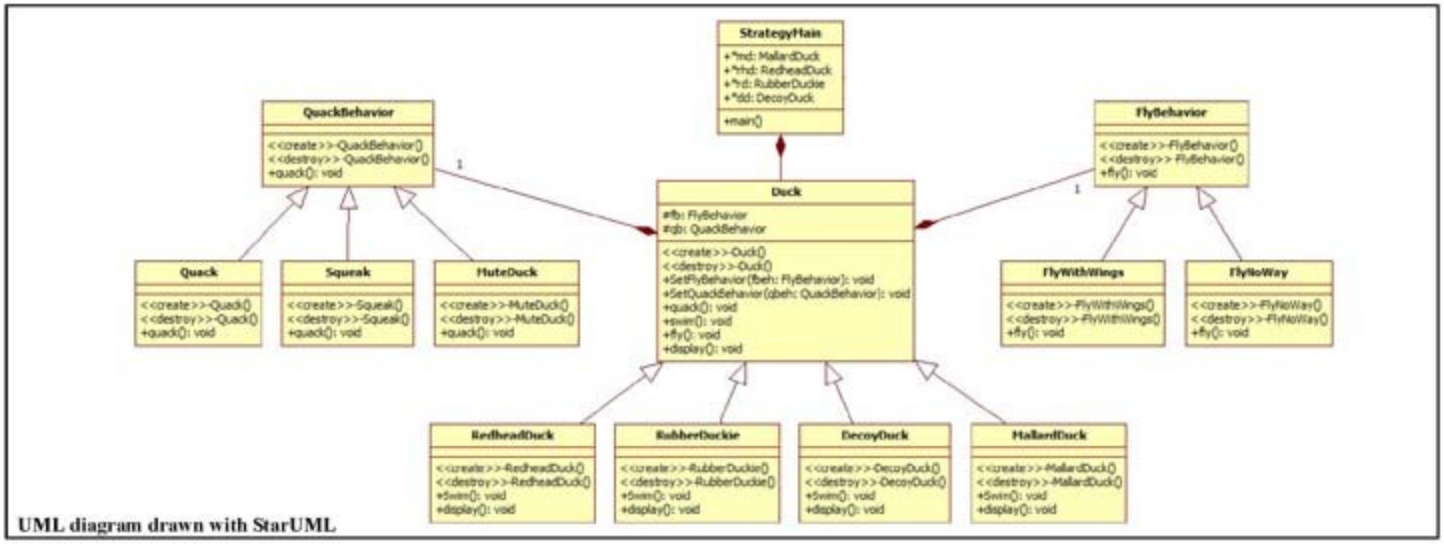


Design Principles

- *Favor composition over inheritance.*

Design Patterns: Strategy

Code Sample



UML diagram drawn with StarUML

StrategyMain

Instantiates instances of:

RedheadDuck

Instantiates instances of Quack and FlyWithWings

RubberDuckie

Instantiates instances of Squeak and FlyNoWay

DecoyDuck

Instantiates instances of MuteDuck and FlyNoWay

MallardDuck

Instantiates instances of Quack and FlyWithWings

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Observer Pattern

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

Behavioral Patterns

- The Strategy Pattern
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- The Observer Pattern
- The Command Pattern
- The Template Method Pattern
- The Iterator Pattern
- The State Pattern
- The Chain of Responsibility Pattern
- The Interpreter Pattern
- The Mediator Pattern
- The Memento Pattern
- The Visitor Pattern

Structural Patterns

- The Decorator Pattern
- The Adapter Pattern
- The Facade Pattern
- The Composite Pattern
- The Proxy Pattern
- The Bridge Pattern
- The Flyweight Pattern

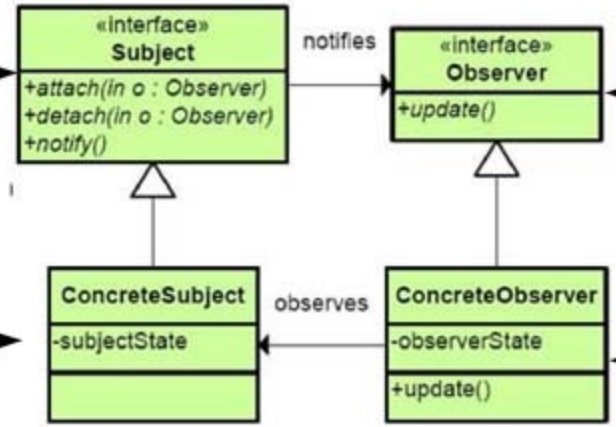
Creational Patterns

- The Factory Method Pattern
- The Abstract Factory Pattern
- The Singleton Pattern
- The Builder Pattern
- The Prototype Pattern

Design Patterns: Observer
Quick Overview

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

Defines the interface the Observers use to access the data provided by a Subject.



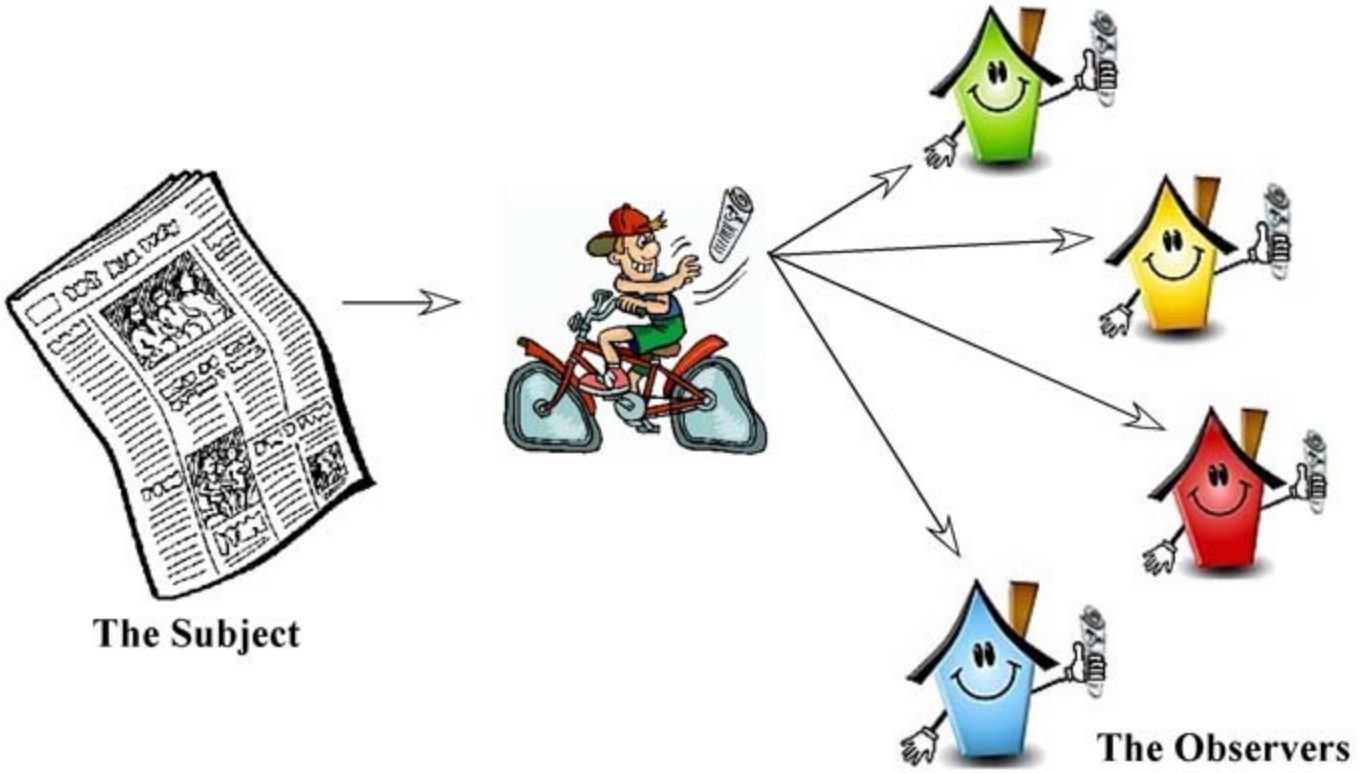
Defines the interface for an Observer that a Subject uses to let it know when data changes.

Actual provider of the data when it's "state" changes.

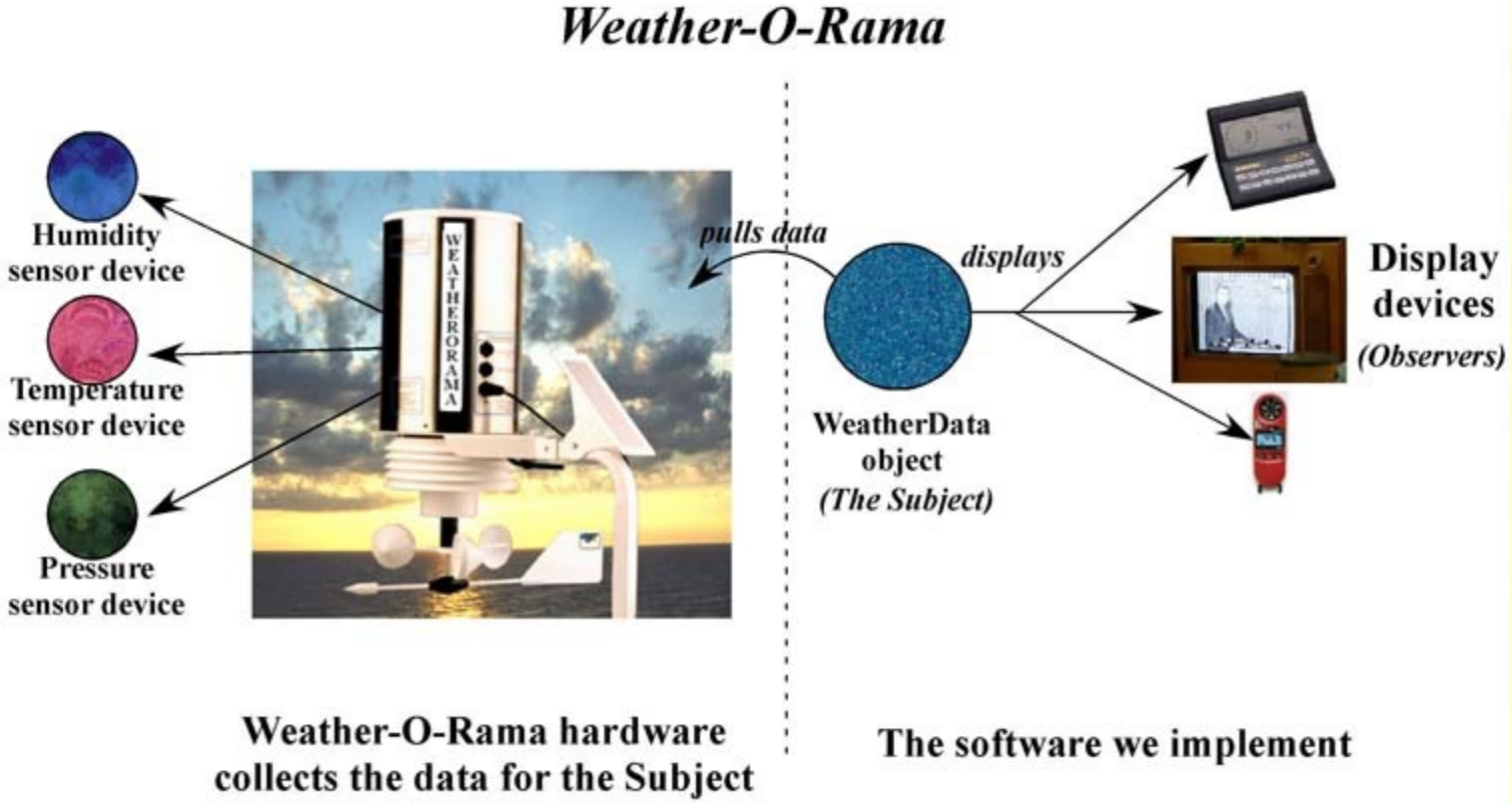
Defines the actual Observer object who wants to know when the data changes.

Design Patterns: Observer

Publisher-Subscriber Relationship

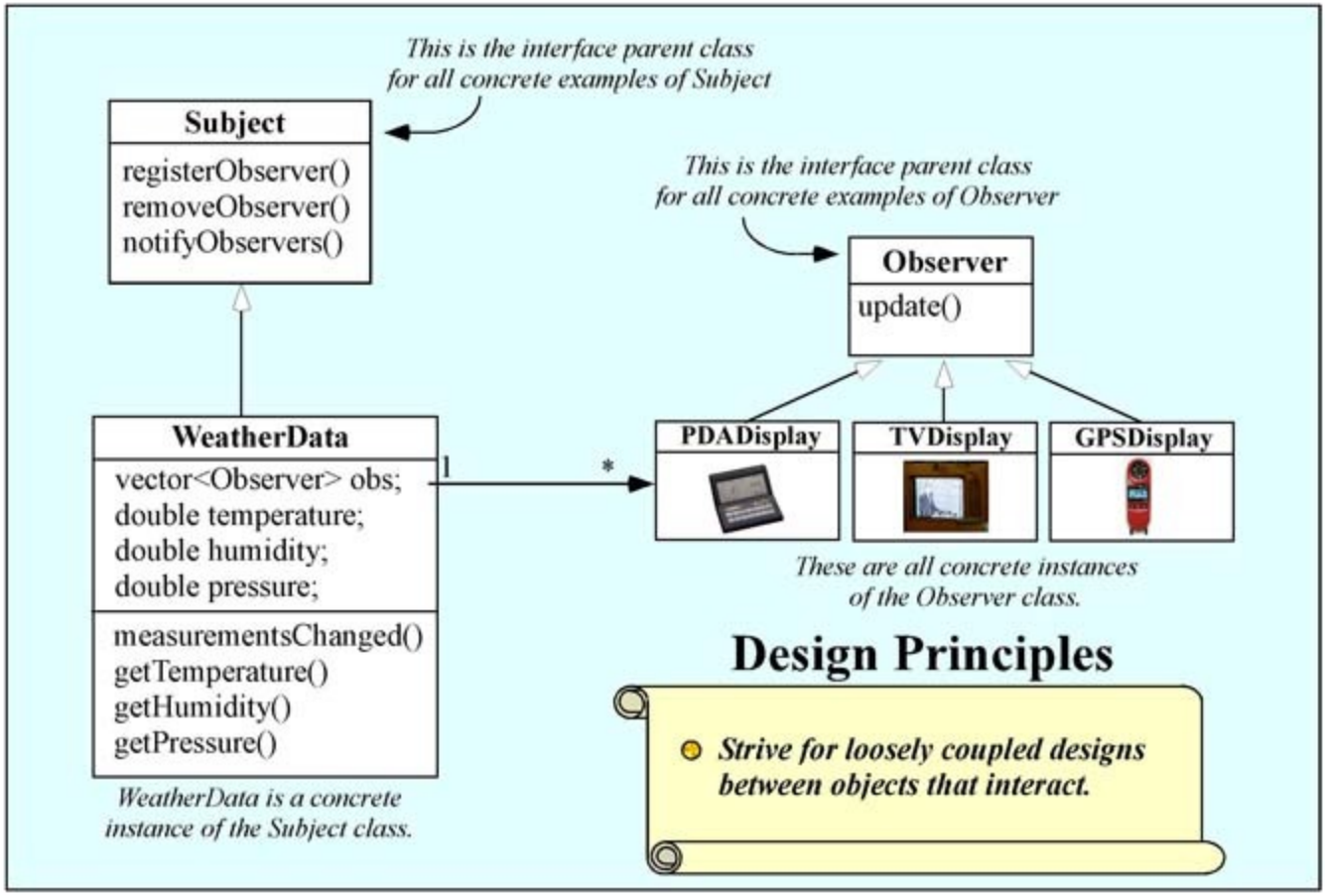


Design Patterns: Observer



Design Patterns: Observer

Class Diagram



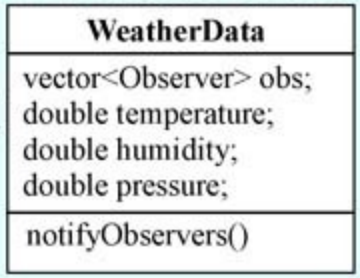
Design Patterns: Observer

A Burning Question

How do the observers actually get the data?

1. Subject PUSHES data to observers

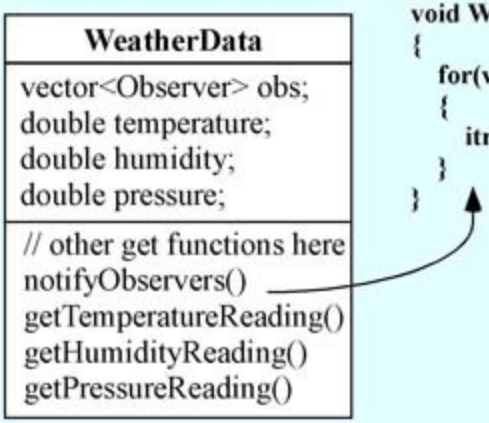
When WeatherData gets new data from the weather station it calls notifyObservers inherited from the Subject class




```
void WeatherData::notifyObservers()
{
   for(vector<Observer>::iterator itr=obs.begin(); itr!=obs.end(); itr++)
   {
      itr->update(temperature, humidity, pressure);
   }
}
```

2. Observers PULL data from the subject

WeatherData calls notifyObservers as above



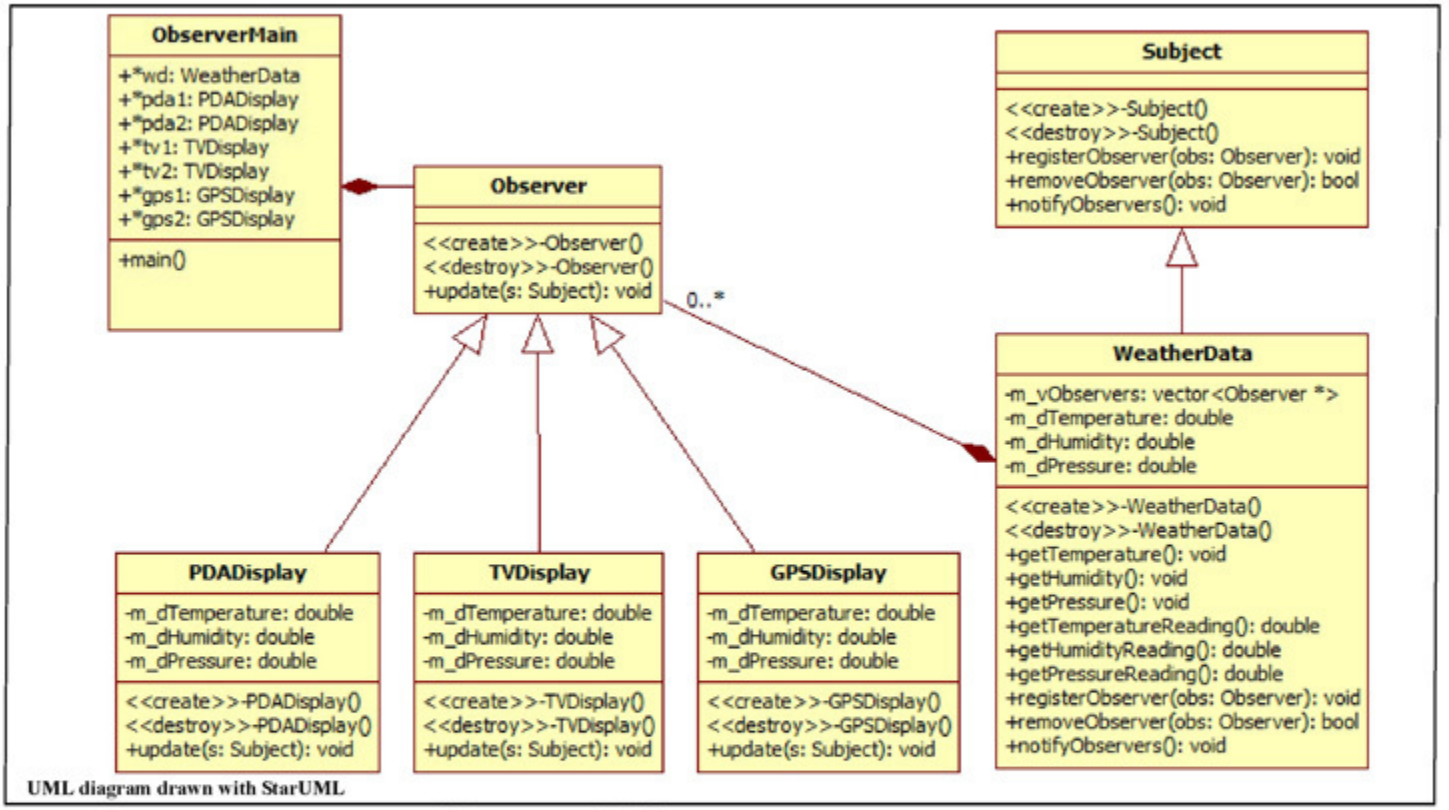

```
void WeatherData::notifyObservers()
{
   for(vector<Observer>::iterator itr=obs.begin(); itr!=obs.end(); itr++)
   {
      itr->update(this);
   }
}
```

```
PDADisplay
void GPSDisplay::update(Subject *s)
{
   this->temp = s->getTemperatureReading()
}
```

Observers can now pull what data they need

Design Patterns: Observer

Code Sample



ObserverMain

- Instantiates Subject as WeatherData
- Instantiates and registers Observers as:
  - PDADisplay, TVDisplay, GPSDisplay
- At one second intervals:
  - Calls WeatherData->notifyObservers
    - Weather Data calls Observer->update(this)
  - Randomly subscribe/unsubscribe observers

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Decorator Pattern

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

Creational Patterns

- **The Factory Method Pattern**
- **The Abstract Factory Pattern**
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

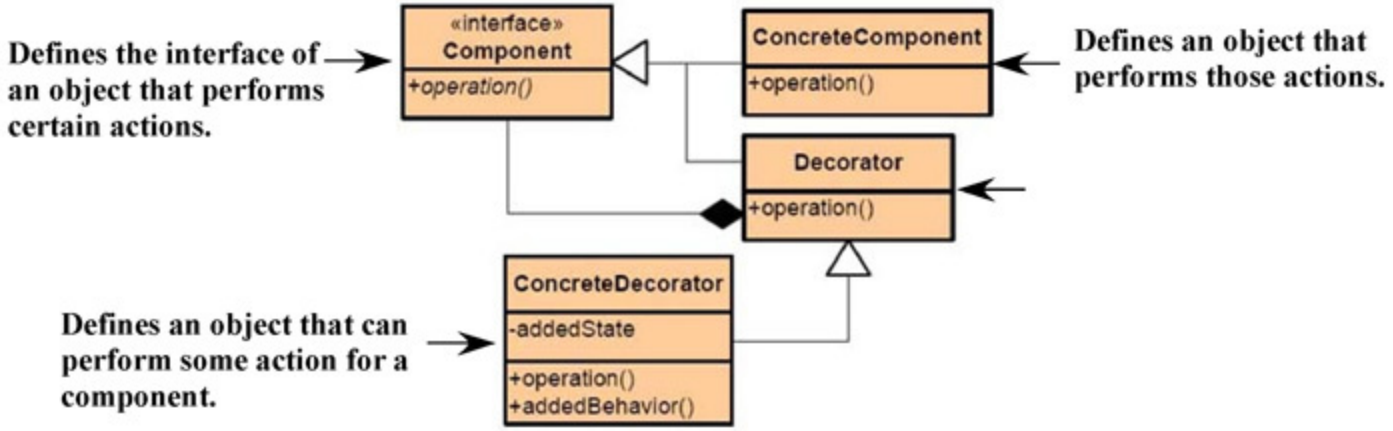
- **The Decorator Pattern**
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

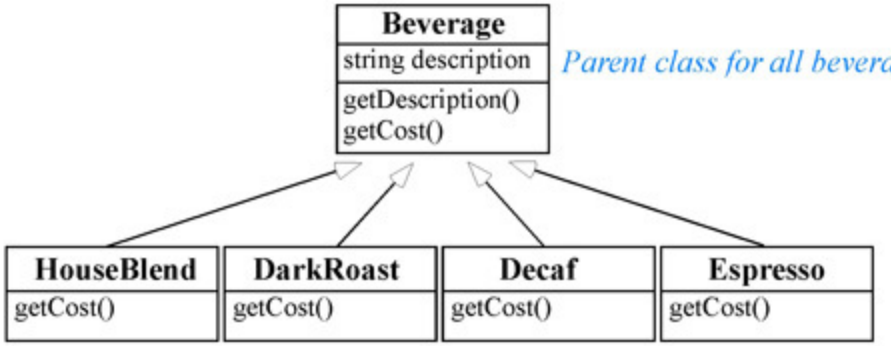
Design Patterns: Decorator
Quick Overview

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



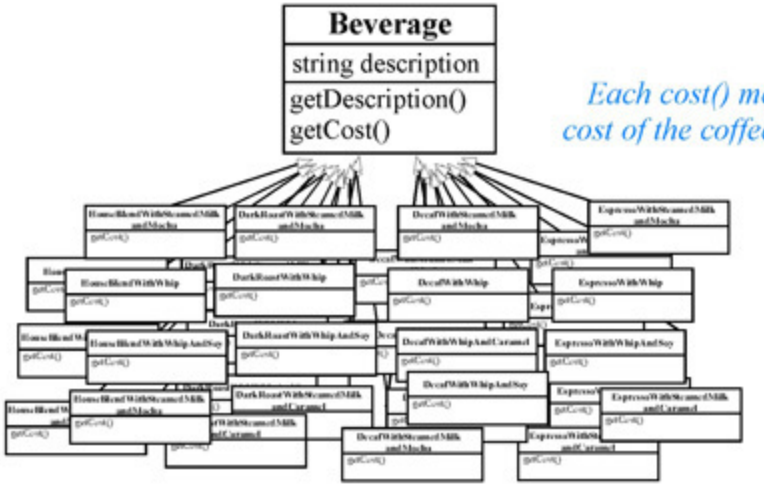
Design Patterns: Decorator

Welcome to Starbuzz Coffee



Parent class for all beverages

In the beginning it was simple. But, then they expanded and things changed...



*Each cost() method calculates the cost of the coffee **and** the condiments.*

Design Patterns: Decorator

What happens when things change?

We add a new beverage, like tea

The price of milk goes up

*The customer wants a **double** mocha*

We add a new condiment

One Possible Solution

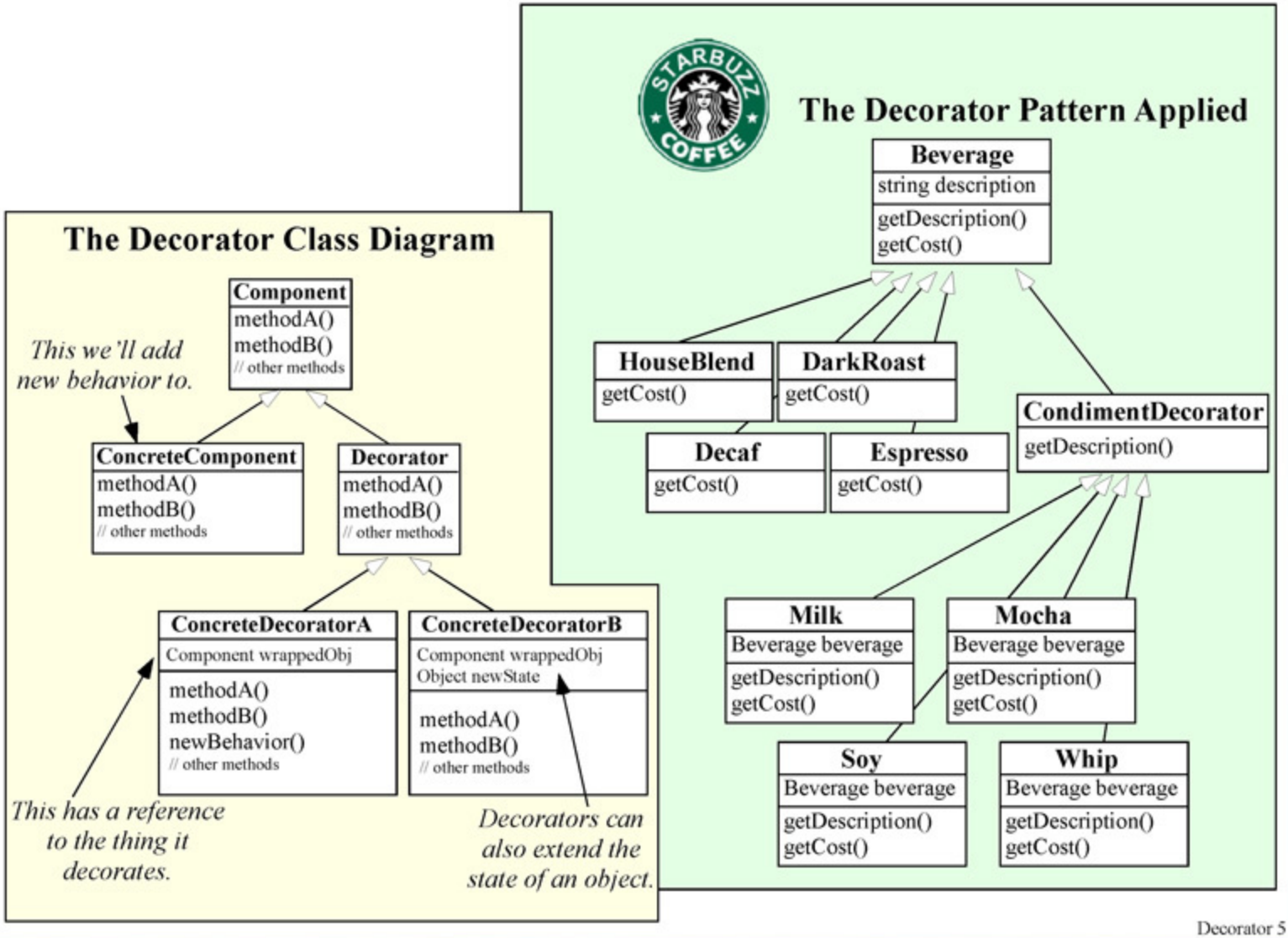
The parent class now handles the cost of the condiments and subclasses take care of their special type.

Beverage
string description bool milk bool soy bool mocha bool whip
getDescription() getCost() hasMilk() setMilk() hasSoy() setSoy() hasMocha() setMocha() hasWhip() setWhip()

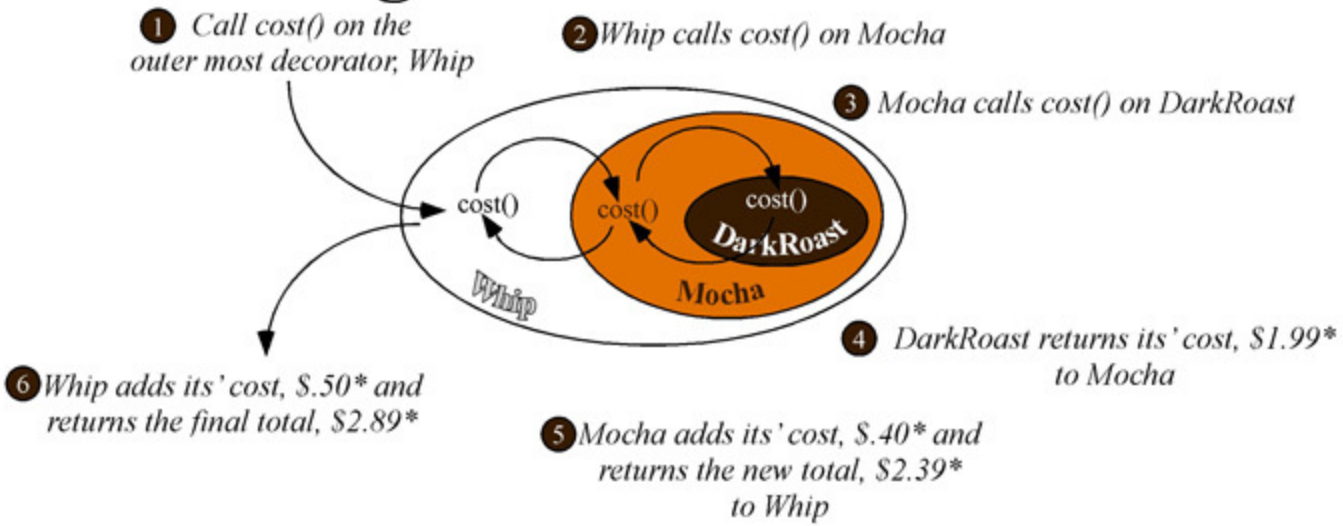
Design Principles

- ***Classes should be open for extension, but closed for modification.***

Design Patterns: Decorator



Design Patterns: Decorator



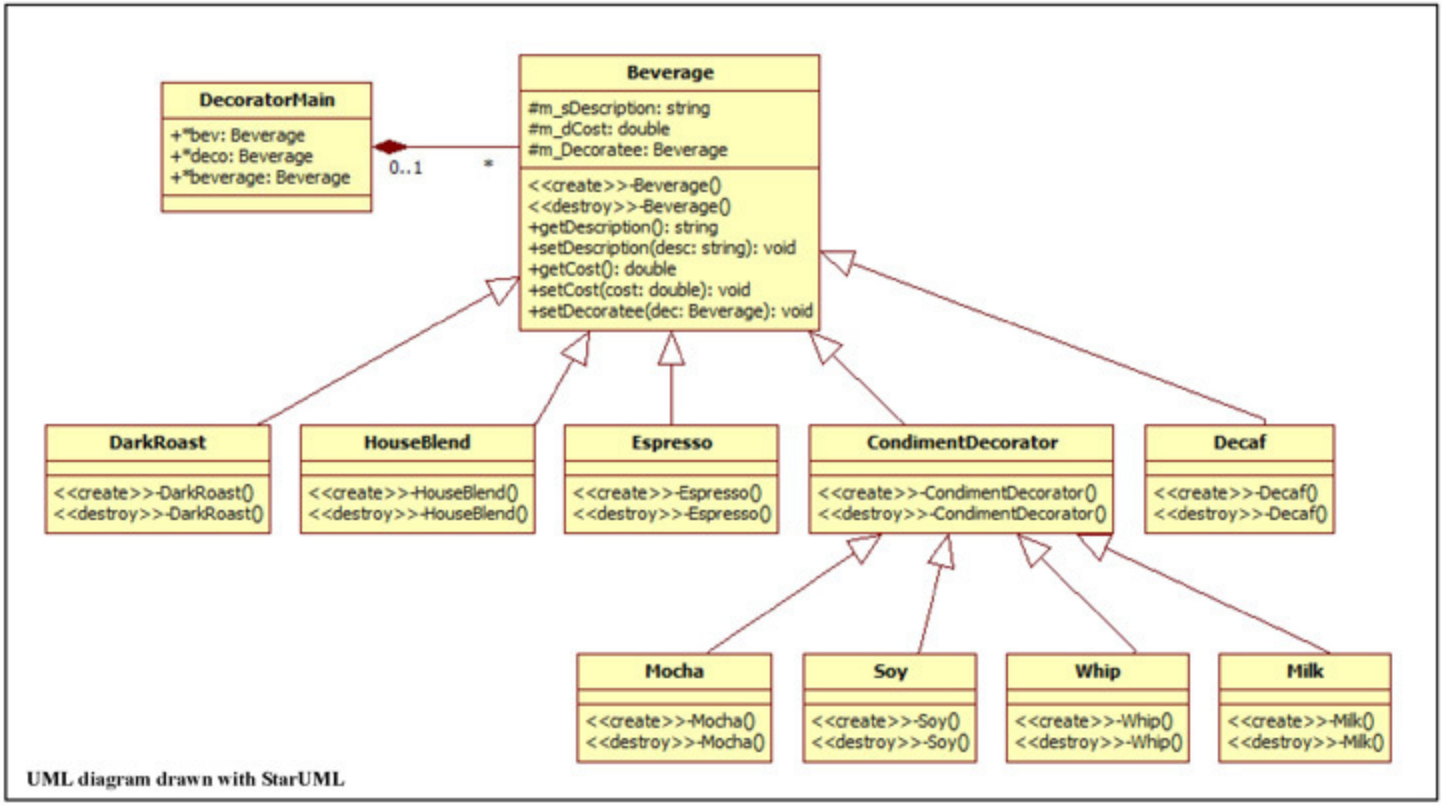
Sure looks a lot like a linked list, or a stack doesn't it?

- **Decorators have the same parent type as the objects they decorate.**
- **One or more decorators can wrap an object.**
- **Because both share the same parent type, a decorator can be passed in place of the object it wraps.**
- **A Decorator adds its' own behavior before and/or after delegating to the object it wraps to do the rest of the job.**
- **Objects can be decorated dynamically at run time.**

** Yeah, I know these prices are probably way too low for reality!*

Design Patterns: Decorator

Code Sample



DecoratorMain

- Instantiates instances of Beverage
- Stacks each with a variety of CondimentDecorators
- Calls getCost on the outermost decorator

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Factory Method Pattern

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

Creational Patterns

- **The Factory Method Pattern**
- **The Abstract Factory Pattern**
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

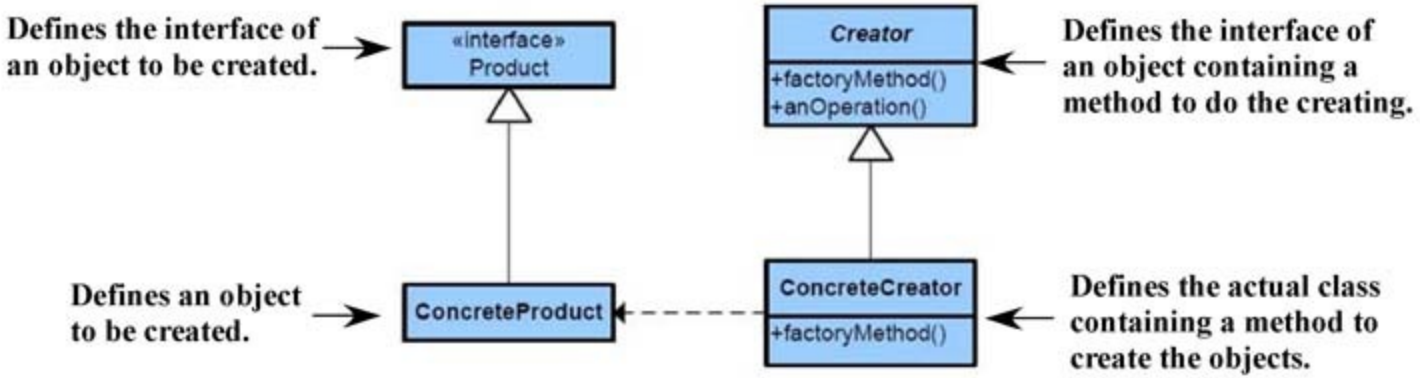
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Factory Method
Quick Overview

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



Design Patterns: The Factory Method

Suppose you have several places in your code where you create ducks. Maybe the code looks like this...

```
Duck duck;

if(picnic)
        duck = new MallardDuck();
else if(hunting)
        duck = new DecoyDuck();
else if(inBathTub)
        duck = new RubberDuck();
```

So what's wrong with using *new* here?

Design Patterns: The Factory Method


```
Pizza *PizzaShop::orderPizza()
{
   Pizza *pizza = new Pizza();

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

A Typical Pizza Shop method

What if I want a special type of pizza?

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;

   if(type == "cheese")
      pizza = new CheesePizza();
   else if(type == "greek")
      pizza = new GreekPizza();
   else if(type == "pepperoni")
      pizza = new PepperoniPizza()

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

Cheese, Greek, and Pepperoni are subclasses of Pizza.

That's better, but...

Remember that old ***Demon Change****?*

Design Patterns: The Factory Method

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;

   if(type == "cheese")
      pizza = new CheesePizza();
   else if(type == "greek")
      pizza = new GreekPizza();
   else if(type == "pepperoni")
      pizza = new PepperoniPizza()

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

Not good! This has to change every time we add a new type of pizza and everywhere in the code where pizzas are created.

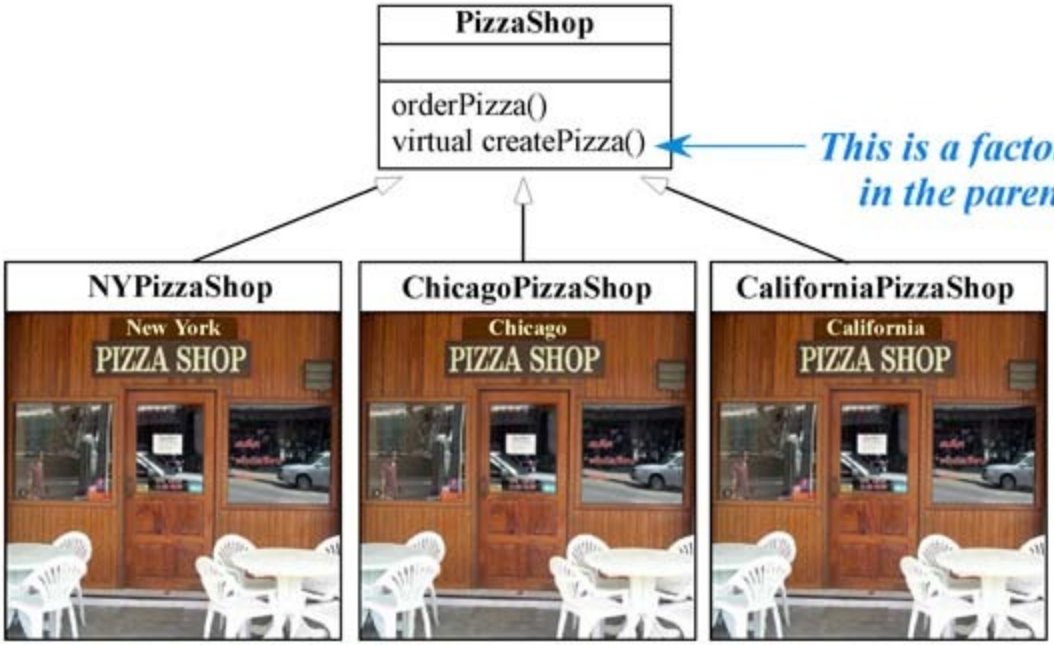
```
Pizza *PizzaShop::createPizza(string type)
{
   Pizza *pizza = NULL;

   if(type == "cheese")
      pizza = new CheesePizza();
   else if(type == "greek")
      pizza = new GreekPizza();
   else if(type == "pepperoni")
      pizza = new PepperoniPizza();
   else if (type == "clam")
      pizza = new ClamPizza();
   else if (type == "veggie")
      pizza = new VeggiePizza();

   return pizza;
}
```

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;
   pizza = this->createPizza(type);
   // Same prepare, bake, cut, box goes here
   return pizza;
}
```

Now we're cooking! ... Well, almost.

Not all pizzas are the same style, even if they are the same type.

Design Patterns: The Factory Method

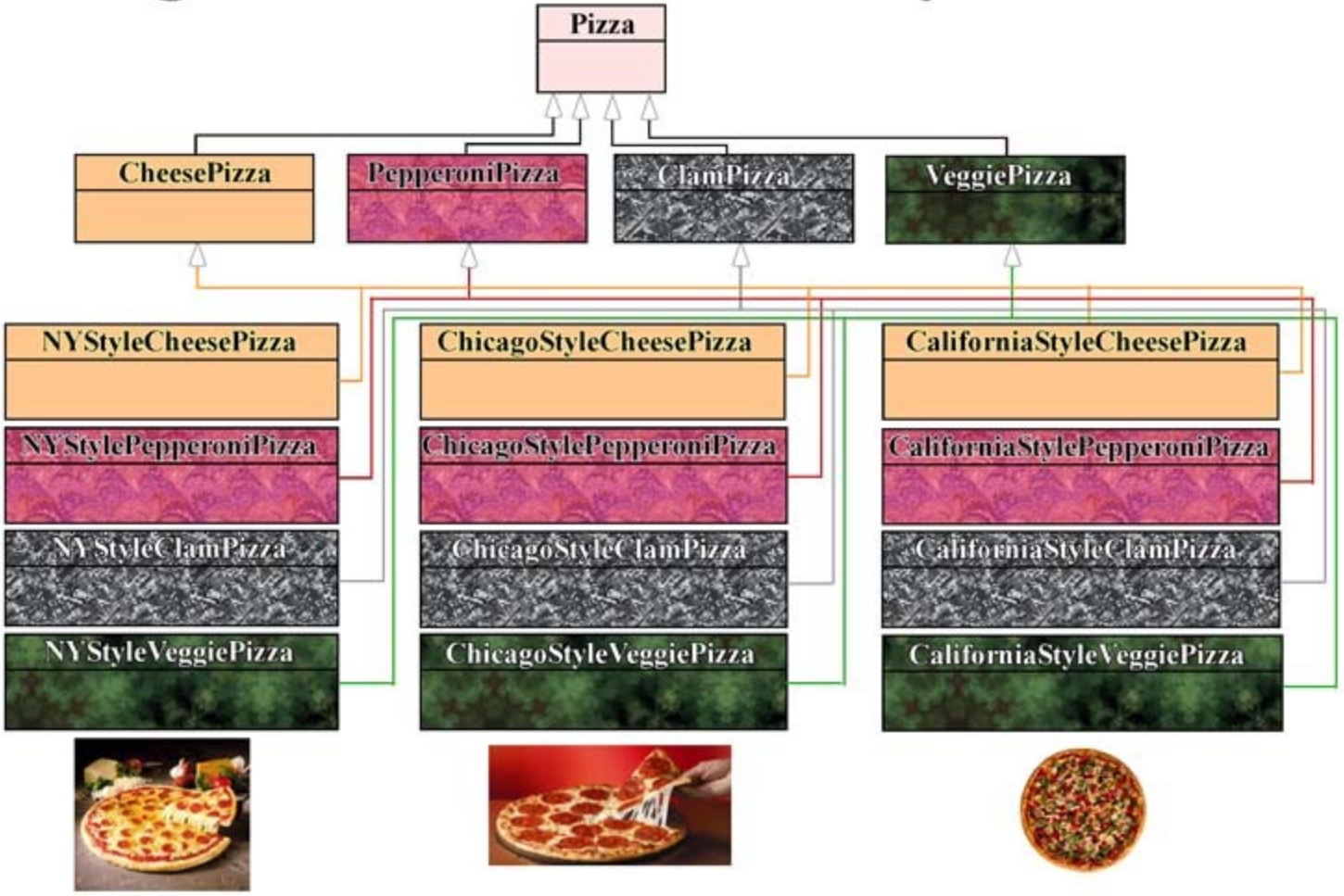



```
Pizza *NYPizzaShop::createPizza(string type)
{
    Pizza *pizza = NULL;

    if(type == "cheese")
      pizza = new NYStyleCheesePizza();
    else if(type == "pepperoni")
      pizza = new NYStylePepperoniPizza();
    else if (type == "clam")
      pizza = new NYStyleClamPizza();
    else if (type == "veggie")
      pizza = new NYStyleVeggiePizza();

    return pizza;
}
```

Each sub-class now must implement this method to create its' particular regional style of pizza.

And we can take this another step...

Design Patterns: The Factory Method



Each PizzaShop sub-class encapsulates the knowledge to create its' particular regional style of pizza.

Design Patterns: The Factory Method

The VeryDependentPizzaShop

```
Pizza *VeryDependentPizzaShop::createPizza(string style, string type)
{
   Pizza *pizza;

   if(style == "NY")
   {
      if(type == "cheese")
         pizza = new NYCheesePizza();
      else if(type == "veggie")
         pizza = new NYVeggiePizza();
      else if(type == "pepperoni";
         pizza = new NYPepperoniPizza()
      else if(type == "clam";
         pizza = new NYClamPizza()
   }
   else if(style == "Chicago")
   {
      if(type == "cheese")
         pizza = new ChicagoCheesePizza();
      else if(type == "veggie")
         pizza = new ChicagoVeggiePizza();
      else if(type == "pepperoni";
         pizza = new ChicagoPepperoniPizza()
      else if(type == "clam";
         pizza = new ChicagoClamPizza()
   }
   // etc. for all the other "Styles"
   else
   {
      cout << "Error: Invalid type of pizza.";
      return NULL;
   }
   if(pizza != NULL)
   {
      pizza->prepare();
      pizza->bake();
      pizza->cut();
      pizza->box();
   }
return pizza;
}
```

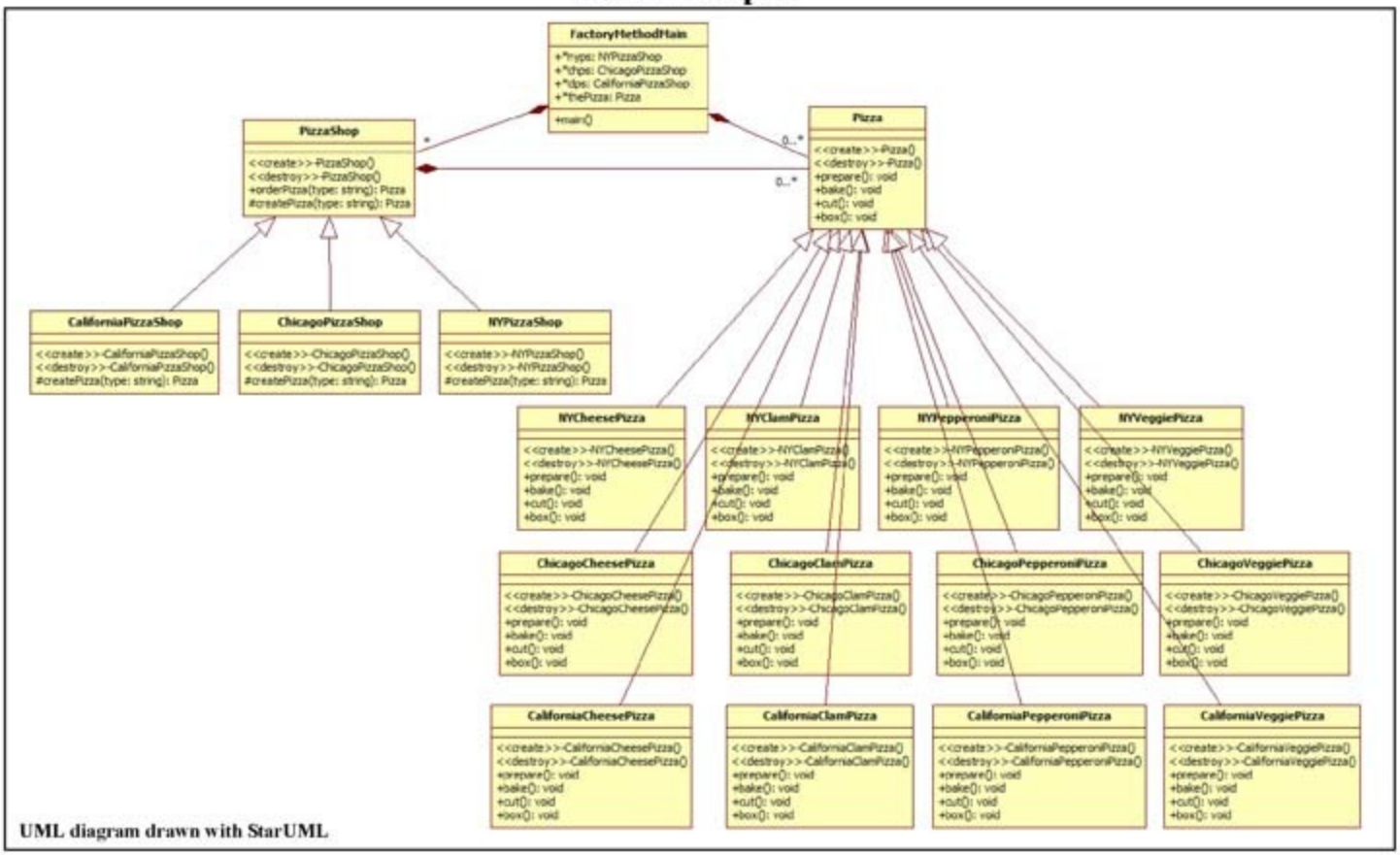
Unfortunately, this is an all too common example of how code can just keep growing and growing and growing as changes occur.

Design Principles

- *Depend on abstractions. Do not depend on concrete classes.*

Design Patterns: The Factory Method

Code Sample



FactoryMethodMain

Instantiates sub-classes of PizzaShop using pointers to Pizzashop
Calls createPizza in each PizzaShop ordering a type of pizza
Using its Factory Method each pizza shop creates its specialty pizza.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Abstract Factory Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

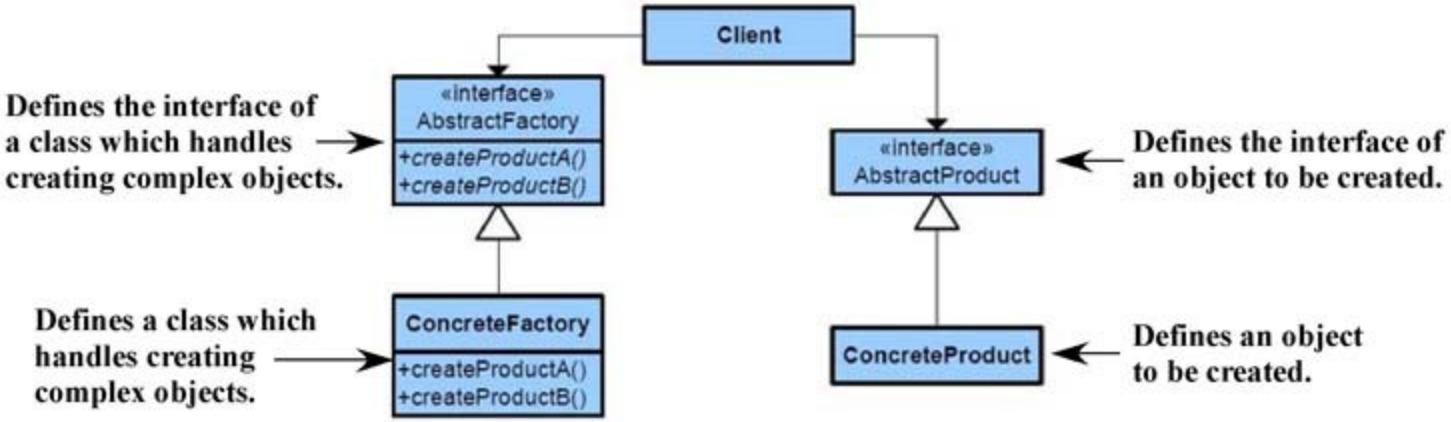
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

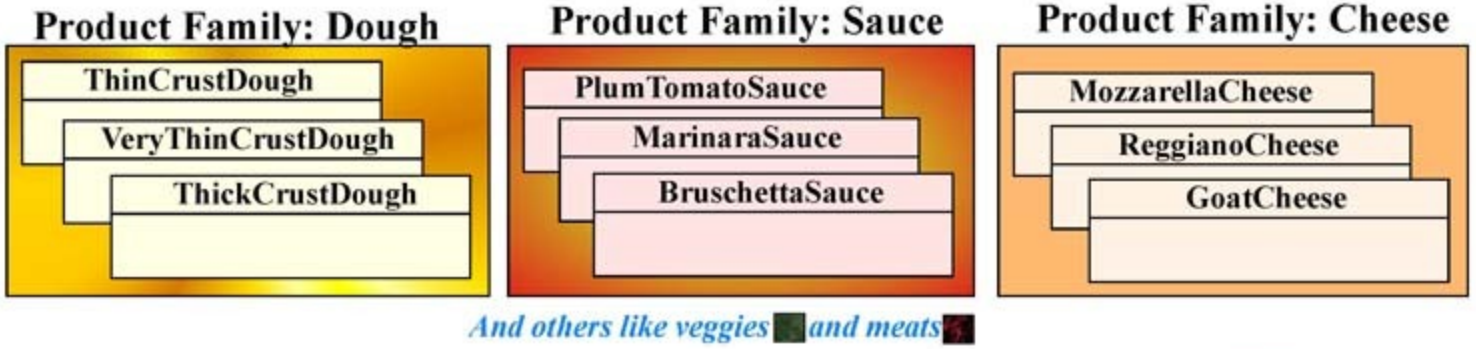
Design Patterns: The Abstract Factory
Quick Overview

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

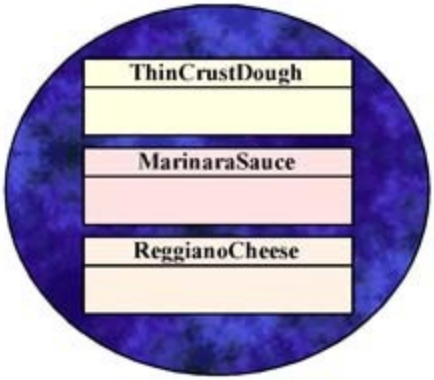


Design Patterns: The Abstract Factory

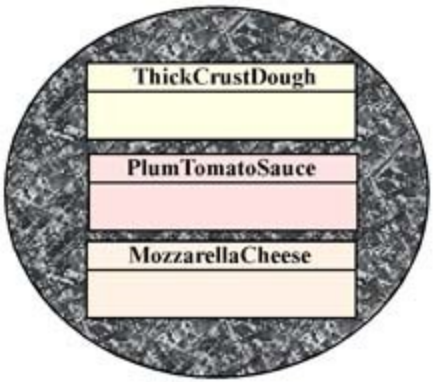
We have the same product families for all of our pizzas, but different implementations based on the region.



And others like veggies and meats



New York



Chicago

California

Design Patterns: The Abstract Factory

PizzaIngredientFactory

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

This becomes our abstract interface for building the various regional factories...

...like this NYPizzaIngredientFactory

```
Dough NYPizzaIngredientFactory::createDough()
{
    return new ThinCrustDough();
}

Sauce NYPizzaIngredientFactory::createSauce()
{
    return new MarinaraSauce();
}

Cheese NYPizzaIngredientFactory::createCheese()
{
    return new ReggianoCheese();
}
```

```
Pepperoni NYPizzaIngredientFactory::createPepperoni()
{
    return new SlicedPepperoni();
}

Veggies NYPizzaIngredientFactory::createVeggiesi()
{
    Veggies veggies[] = {new Garlic(), new Onion(),
          new Mushroom(), new RedPepper()};
    return veggies;
}

Clams NYPizzaIngredientFactory::createClams()
{
    return new FreshClams();
}
```

When they are created each PizzaShop sub-class instantiates the appropriate PizzaIngredientFactory

Design Patterns: The Abstract Factory

Now let's rework the Pizza class.

Pizza

```
class Pizza
{
    public:
        Dough dough;
        Sauce sauce;
        Cheese cheese;
        Veggies veggies[];
        Pepperoni pepperoni;
        Clams clams;

        virtual void prepare();
        void bake();
        void cut();
        void box();
        void setName(string name);
        string getName();
}
```

And each of its sub-classes like this CheesePizza.

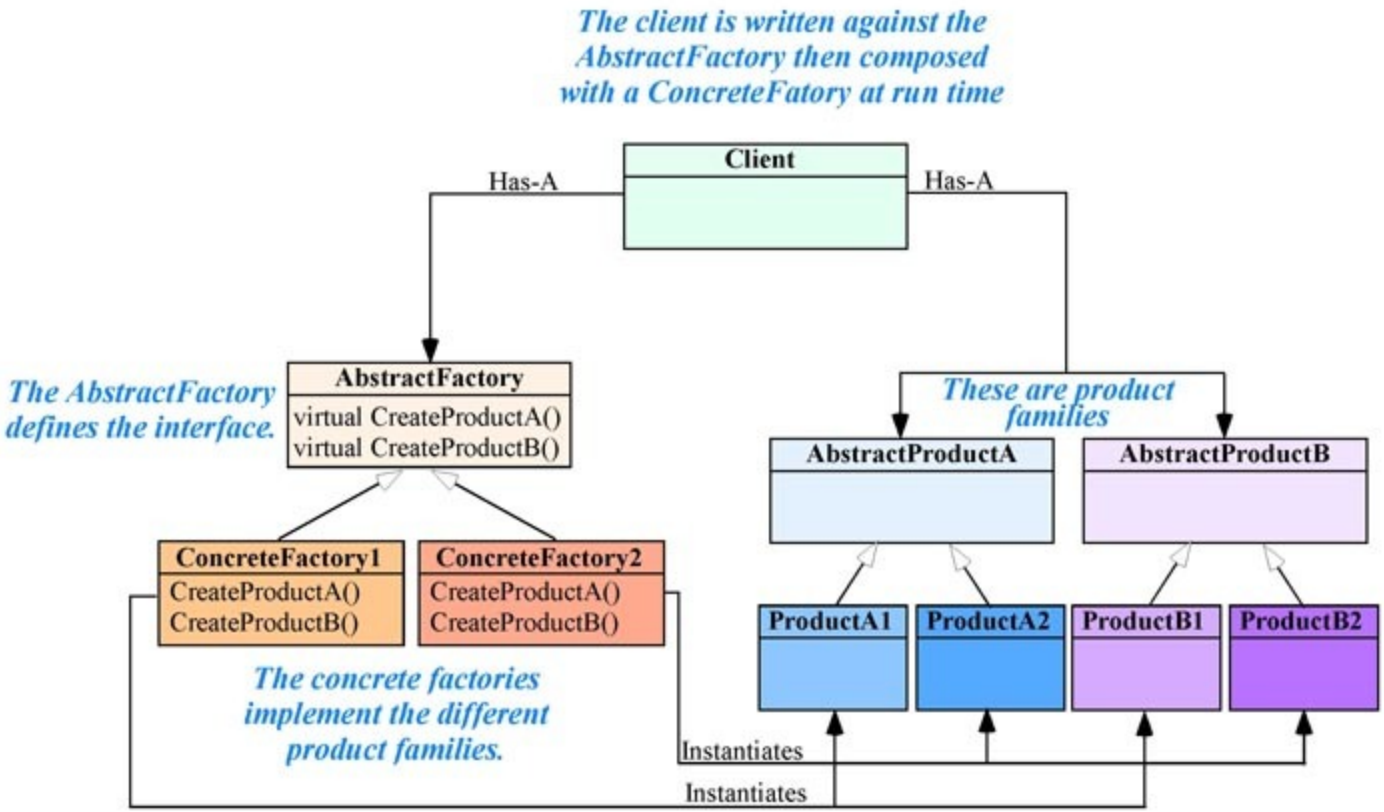
CheesePizza

```
class CheesePizza::Pizza
{
   private:
      PizzaIngredientFactory ingredientFactory;

    public:
        CheesePizza(PizzaIngredientFactory
                                    ingredientFactory)
        {
            this->ingredientFactory = ingredientFactory;
        }

        void prepare()
        {
            cout << "Preparing " << name;
            dough=ingredientFactory.createDough();
            sauce=ingredientFactory.createSauce();
            cheese=ingredientFactory.createCheese();
        }
}
```

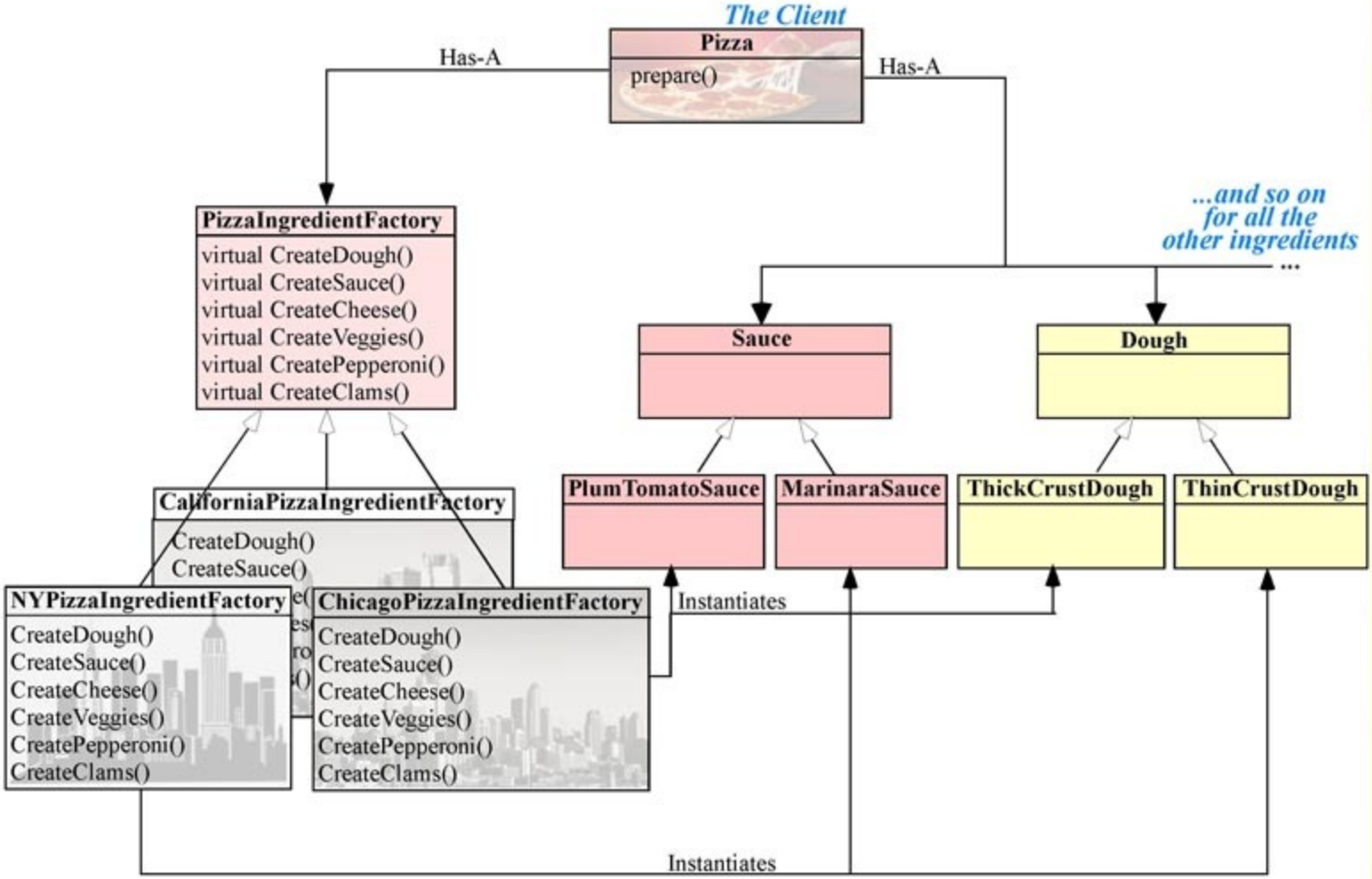
In the constructor we pass in the appropriate PizzaIngredientFactory for the region.

The Pizza doesn't care which PizzaIngredientFactory it uses.

Design Patterns: The Abstract Factory

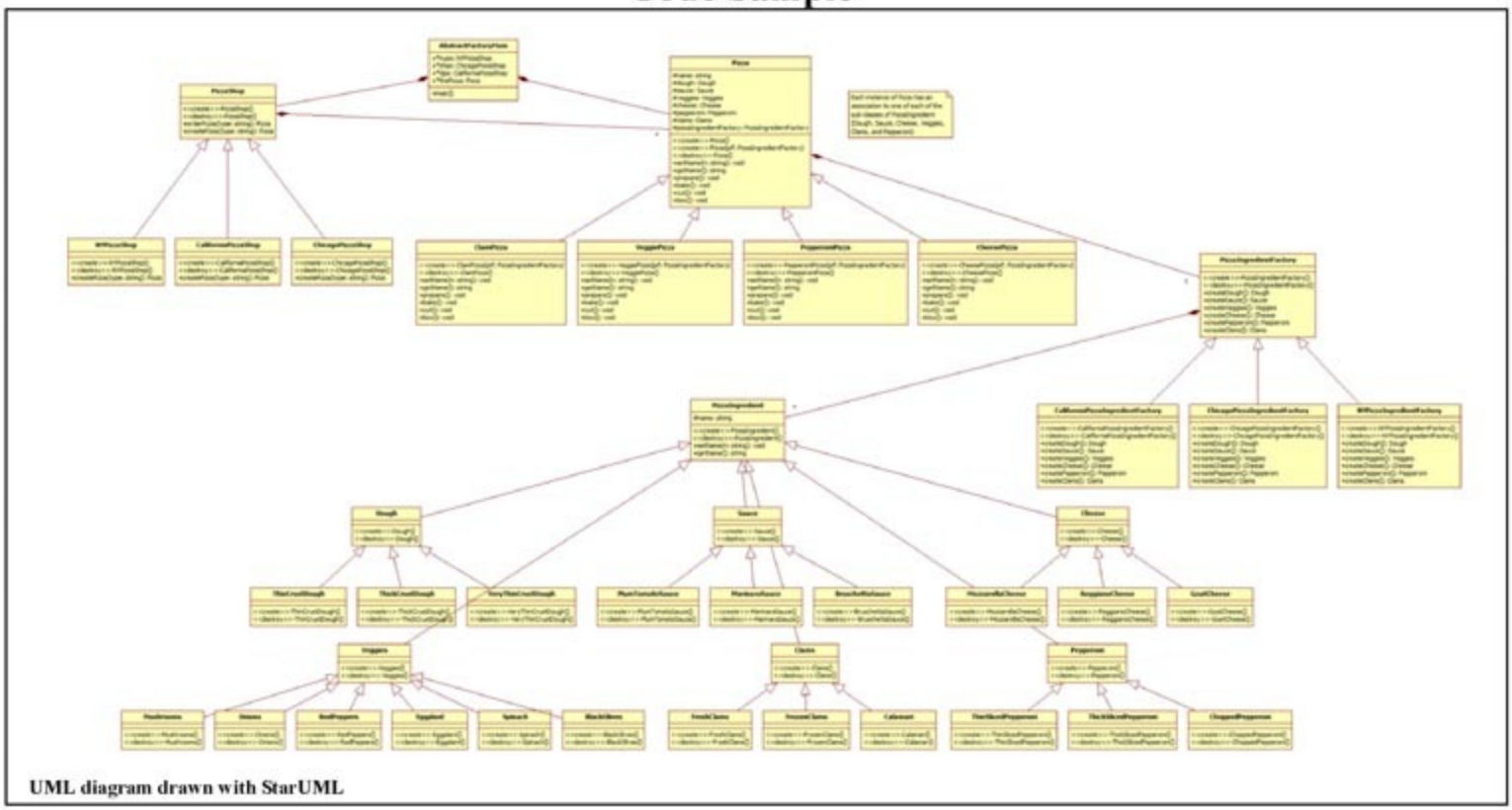


Design Patterns: The Abstract Factory



Design Patterns: Abstract Factory

Code Sample



UML diagram drawn with StarUML

AbstractFactorMain

"Customer" enters a PizzaShop (NY, Chi., or Cal.)

"Customer" orders a type of pizza (Cheese, Clam, Pepperoni, Veggie)

Calls PizzaShop->OrderPizza(type) which calls CreatePizza(type)

Creates the Pizza type passing it the appropriate ingredient factory

Pizza uses the ingredient factory to create its regional style

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Singleton Pattern

Ensures a class has only one instance, and provides a global point of access to it.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

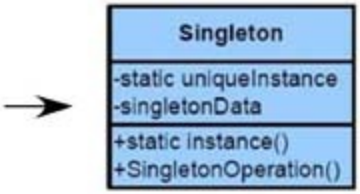
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Singleton

Quick Overview

Ensures a class has only one instance, and provides a global point of access to it.

Defines the class as a Singleton and ensures only one instance can be created.



Design Patterns: The Singleton

1 and only 1

Design Patterns: The Singleton

Examples

- A factory class which instantiates instances of other classes, all of which must follow a standard.

- A print spooler which must have one and only one instance sending print jobs to a printer.

- A window manager for a GUI application.

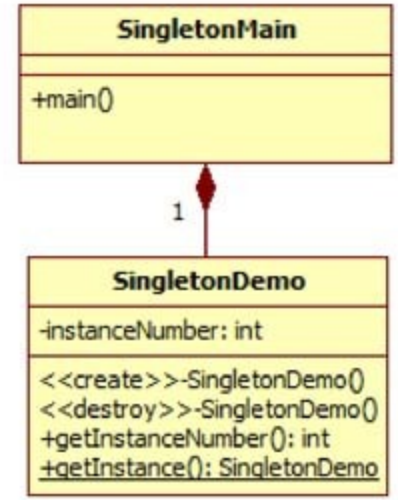
Design Patterns: The Singleton

```
//=========================================
// SingletonDemo.h
//=========================================
#ifndef SINGLETONDEMO_H
#define SINGLETONDEMO_H
class SingletonDemo
{
        private:
                int instanceNumber;
                SingletonDemo();
        public:
                ~SingletonDemo();
                int getInstanceNumber();
                static SingletonDemo *getInstance();
};
#endif
```

```
//=========================================
// SingletonDemo.cpp
//=========================================
#include "SingletonDemo.h"
SingletonDemo::SingletonDemo() {}
SingletonDemo::~SingletonDemo() {}
int SingletonDemo::getInstanceNumber()
{
   return this->instanceNumber;
}
//=========================================
// Return the singleton instance
//=========================================
SingletonDemo *SingletonDemo::getInstance()
{
   static SingletonDemo *theInstance = NULL;
   static int counter = 1;
   if(theInstance == NULL)
   {
      theInstance = new SingletonDemo();
      theInstance->instanceNumber = counter;
      counter++;
   }
   return theInstance;
}
```

Design Patterns: Singleton

Code Sample



UML diagram drawn with StarUML

SingletonMain

- Creates two pointers to SingletonDemo
- Calls getInstance() for each pointer
- Calls getInstanceNumber to show both point to the same instance.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Command Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

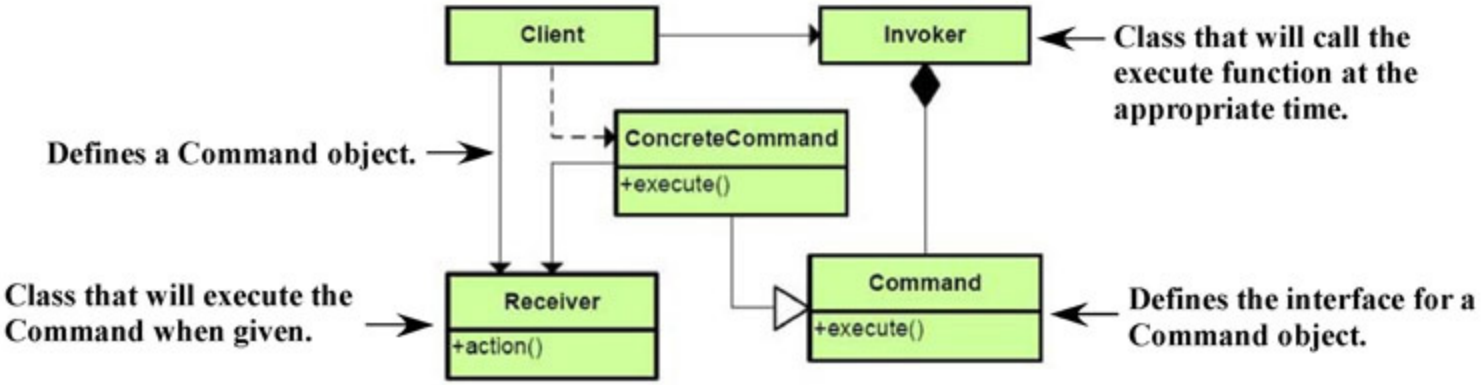
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

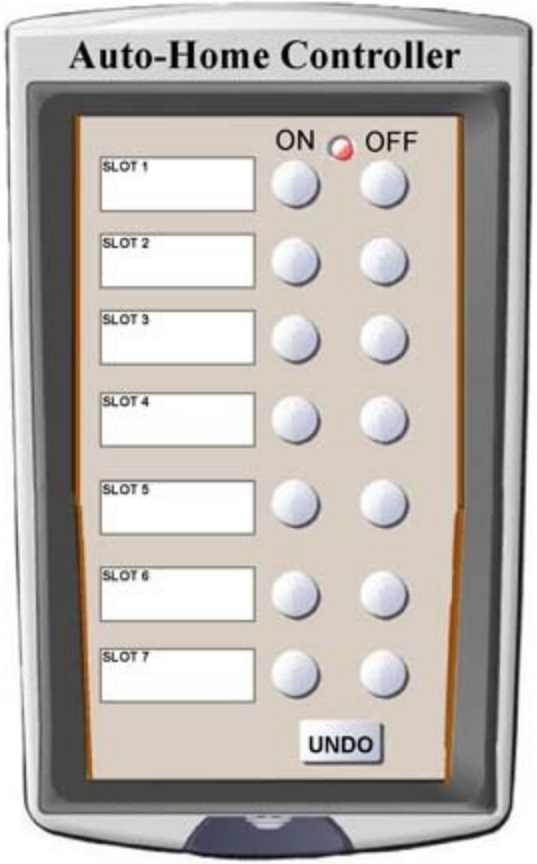
Design Patterns: The Command
Quick Overview

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

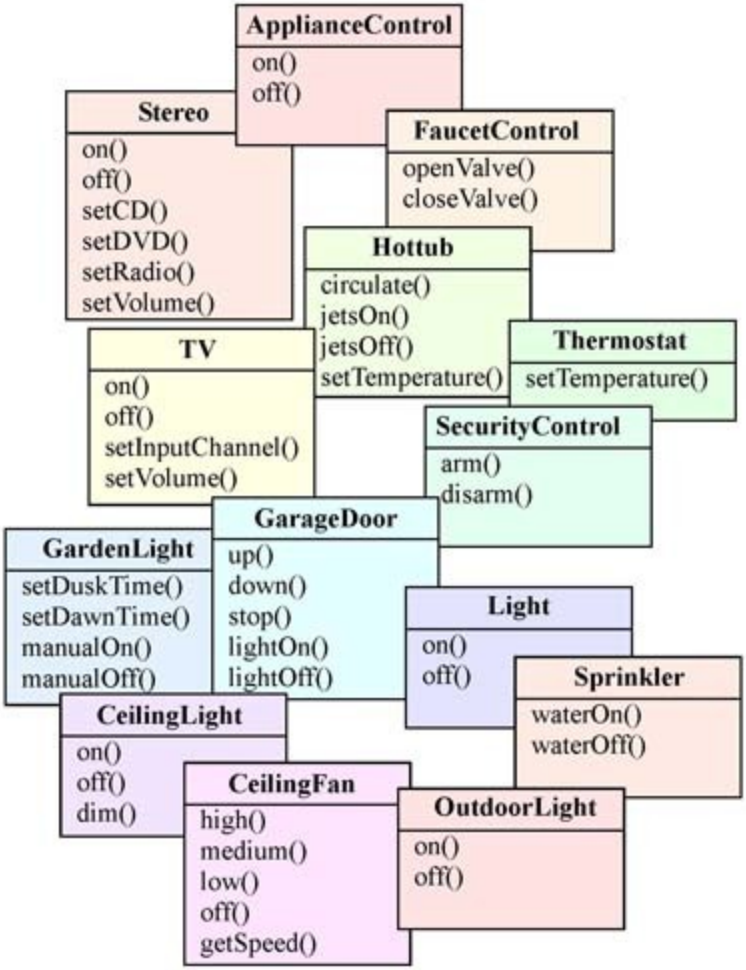


Design Patterns: The Command

We need to program this...



...to interface with and control any of these.



Design Patterns: The Command

How would you design it?

Some starting points to consider

- *None of the different devices have a common interface.*
- *There will probably be many more such devices added in the future, i.e. we can expect the system to CHANGE.*
- *The remote control should know how to interpret button presses and make requests, but it shouldn't know any of the details about how a device works.*
- *We don't want the remote to consist of a long series of* **if** *statements, like*

```
if slot1==Light then light.on()
else if slot1==Hottub then hottub.jetsOn()
```

Just a hint...

The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action.

Now think about that for a minute...

Design Patterns: The Command

You go into the diner and your order is taken by Flo.

Who writes down your order, and takes it to the order counter.

Earl, the short-order cook picks up the order and prepares your meal then calls "Order up!"

Design Patterns: The Command

From the Diner to the Command Pattern

Invoker

Receiver

Command

Client

The client creates a command object (your order) and sets its Receiver (Earl)

createCommandObject(receiver)

The client gives the command to the Invoker (Flo)

setCommand(command)

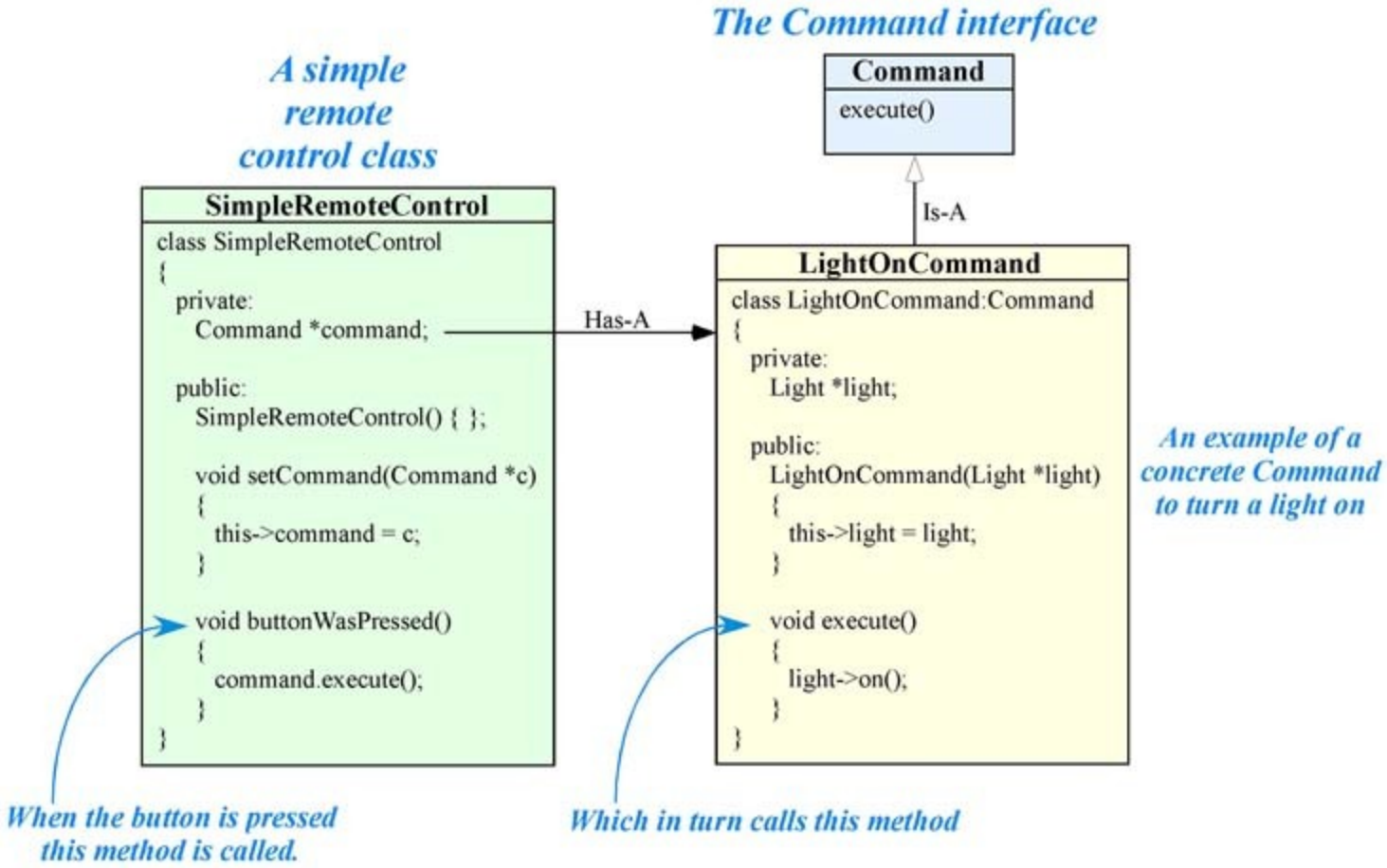
The Invoker (Flo) calls the execute command at the appropriate time.

command.execute()

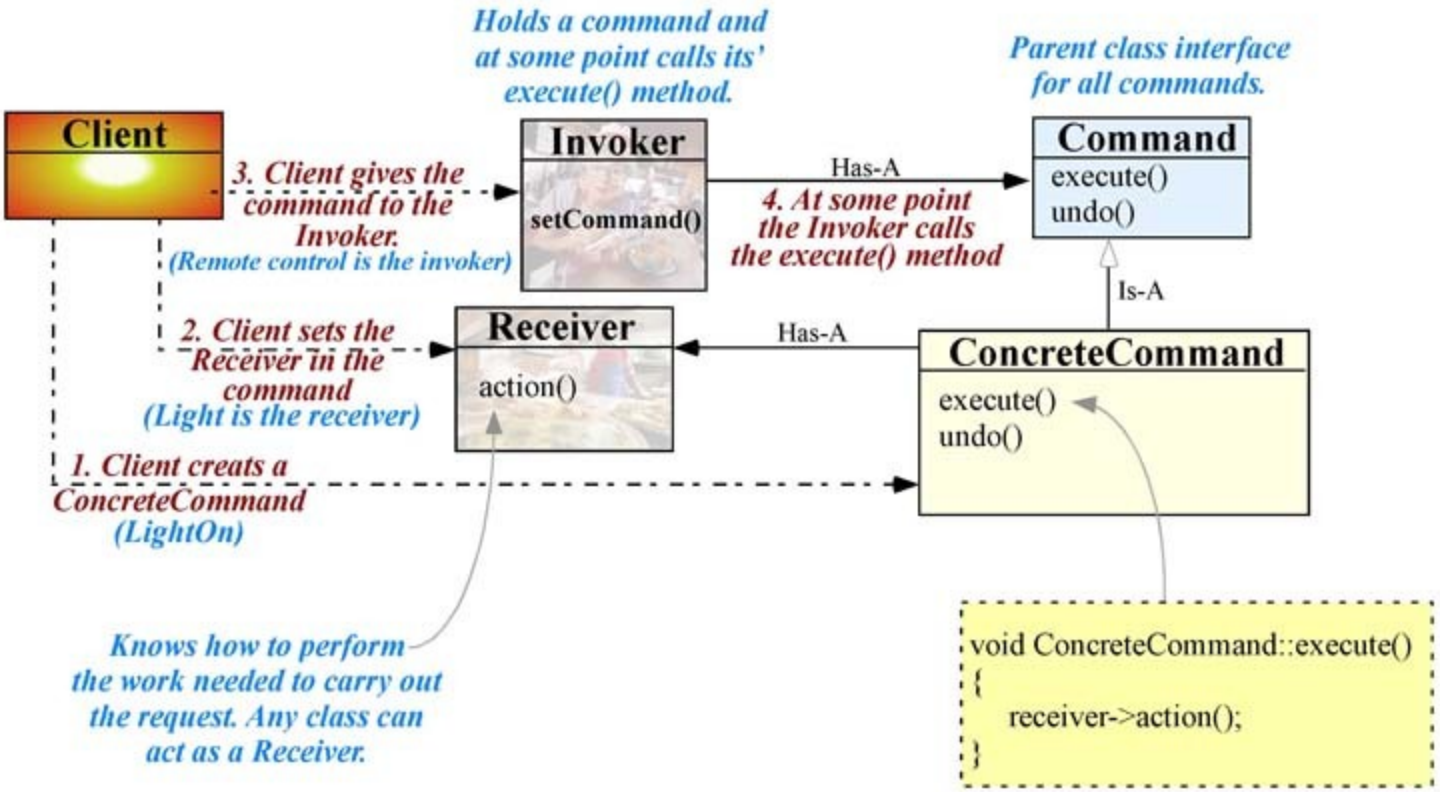
The Receiver (Earl) executes the command.

action1()
action2()

Design Patterns: The Command



Design Patterns: The Command



Design Patterns: The Command

Implementing the Remote Control

```
class RemoteControl
{
    private:
        Command *onCommands[7];
        Command *offCommands[7];
        Command *undoCommand;
    public:
        RemoteControl();
        void setCommand(int slot,
                Command *onCommand,
                Command *offCommand);
        void onButtonPushed(int slot);
        void offButtonPushed(int slot);
}
```

?1

?1 *You'll see what this command is shortly.*

?2 *So we don't have to keep checking to see if there really is a command here.*

NoCommand

execute() {}; // Do nothing
undo(){}; // Do nothing

```
RemoteControl::RemoteControl()
{
    for(int i=0; i<7; i++)
    {
        onCommands[i] = new NoCommand();
        offCommands[i] = new NoCommand();
    }
    undoCommand = new NoCommand();
}
void RemoteControl::setCommand(int slot,
                Command *onCommand,
                Command *offCommand)
{
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
void RemoteControl::onButtonPushed(int slot)
{
    onCommands[slot]->execute();
    undoCommand = onCommands[slot];
}
 void RemoteControl::offButtonPushed(int slot)
{
    offCommands[slot]->execute();
    undoCommand = offCommands[slot];
}
```

?2

Starting to get an idea?

Design Patterns: The Command

Implementing the Commands and Undo

LightOnCommand

```
class LightOnCommand:Command
{
  private:
    Light *light;
  public:
    LightOnCommand(Light *light)
    {
      this->light = light;
    }
    void execute()
    {
      light->on();
    }
    void undo()
    {
      light->off();
    }
}
```

LightOffCommand

```
class LightOffCommand:Command
{
  private:
    Light *light;
  public:
    LightOffCommand(Light *light)
    {
      this->light = light;
    }
    void execute()
    {
      light->off();
    }
    void undo()
    {
      light->on();
    }
}
```

StereoOnWithCDCommand

```
class StereoOnCommand:Command
{
  private:
    Stereo *stereo;
  public:
    StereoOnCommand(Stereo *stereo)
    {
      this->stereo = stereo;
    }
    void execute()
    {
      stereo->on();
      stereo->setCD();
      stereo->setVolume(11);
    }
    void undo()
    {
      stereo->off();
    }
}
```

...and so on.

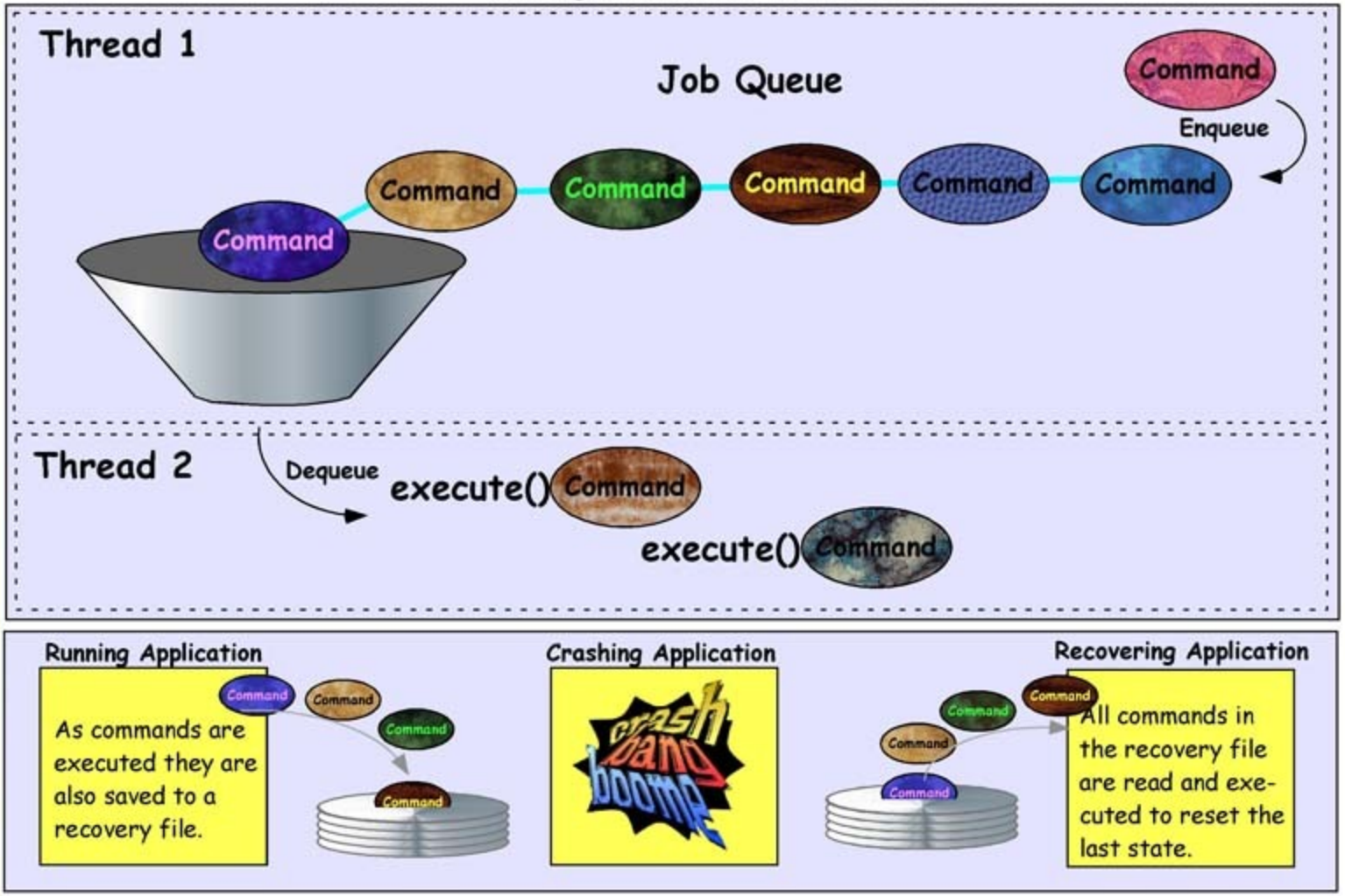
Notice we have now added an undo() function.

...and we add this function to the RemoteControl to handle the undo action.

```
void RemoteControl::undoButtonPushed()
{
    undoCommand->undo;
}
```

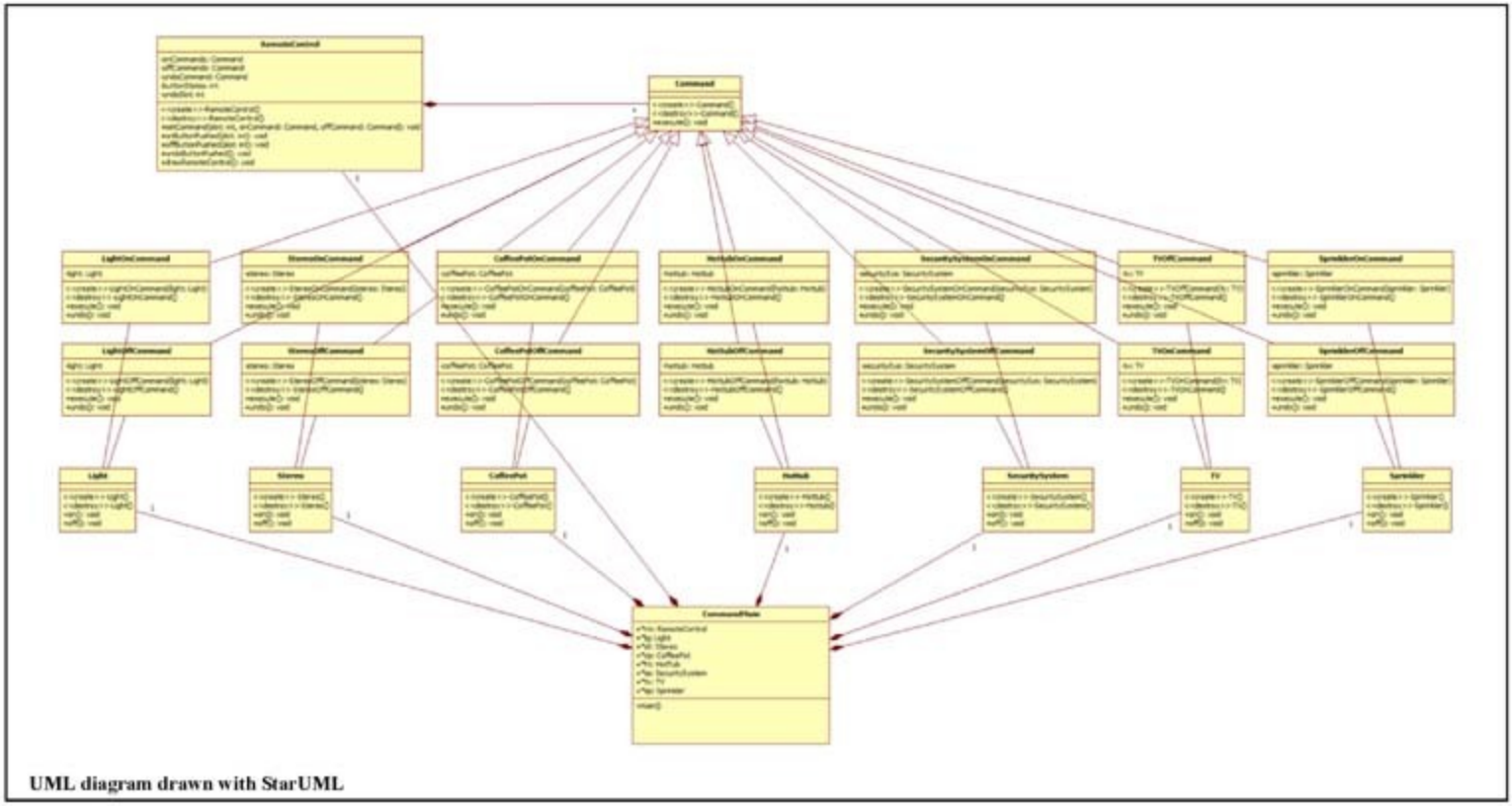
Design Patterns: The Command

Doing More with Commands



Design Patterns: The Command

Code Sample



UML diagram drawn with StarUML

CommandMain

Creates a number of instances of Command and registers with the Invoker class RemoteControl

For each button input from user

Calls execute so Receiver can handle the command.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Adapter Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

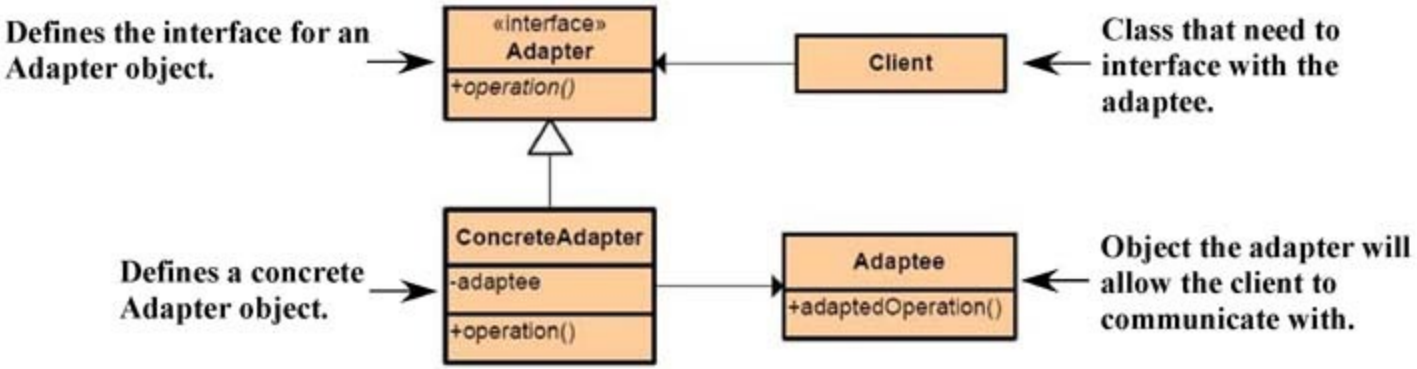
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

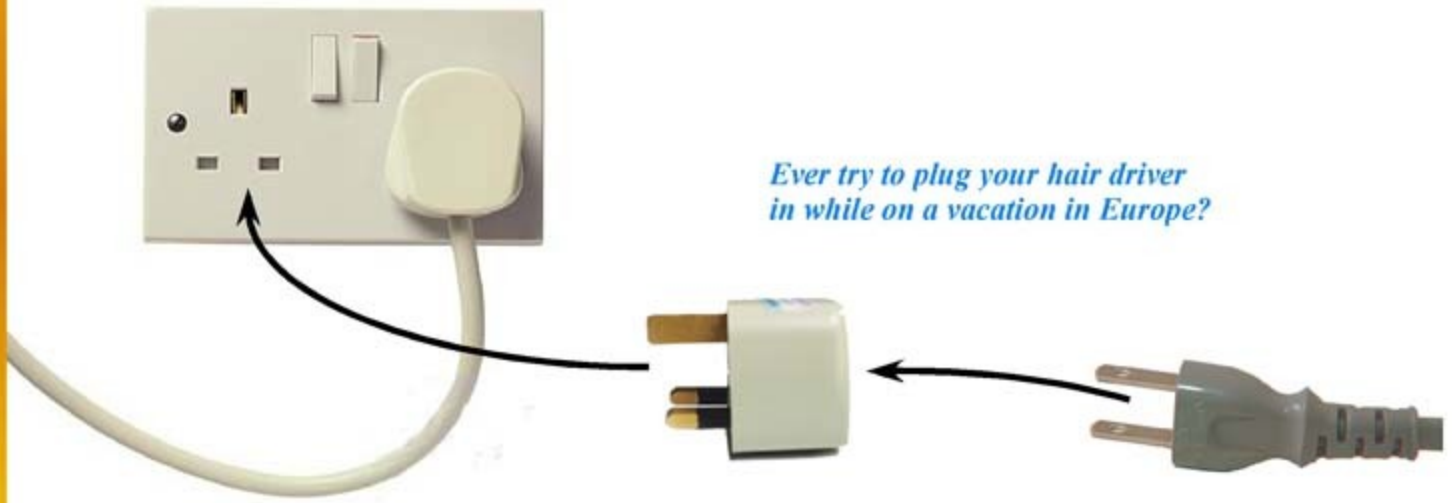
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Adapter
Quick Overview

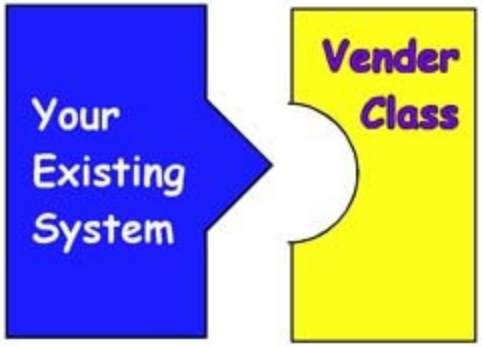
Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



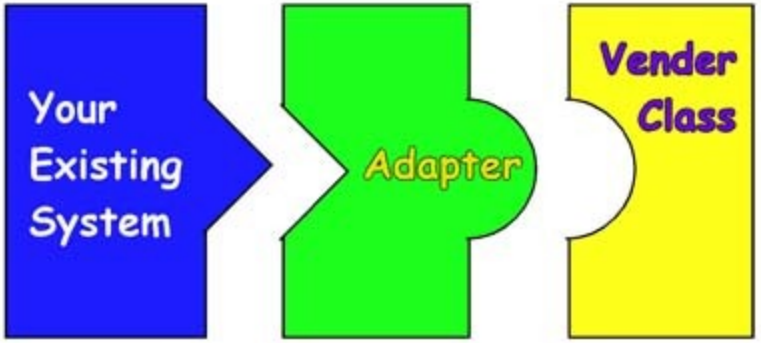
Design Patterns: The Adapter



Ever try to plug your hair driver in while on a vacation in Europe?



Without Adapter



With Adapter

Design Patterns: The Adapter

Remember the Duck interface?

Meet the newest fowl on the block, the Turkey interface.

Duck

void quack()
void fly()

Turkey

void gobble()
void fly()

Turkey's don't quack, they gobble.

Turkey's can fly, but only short distances.

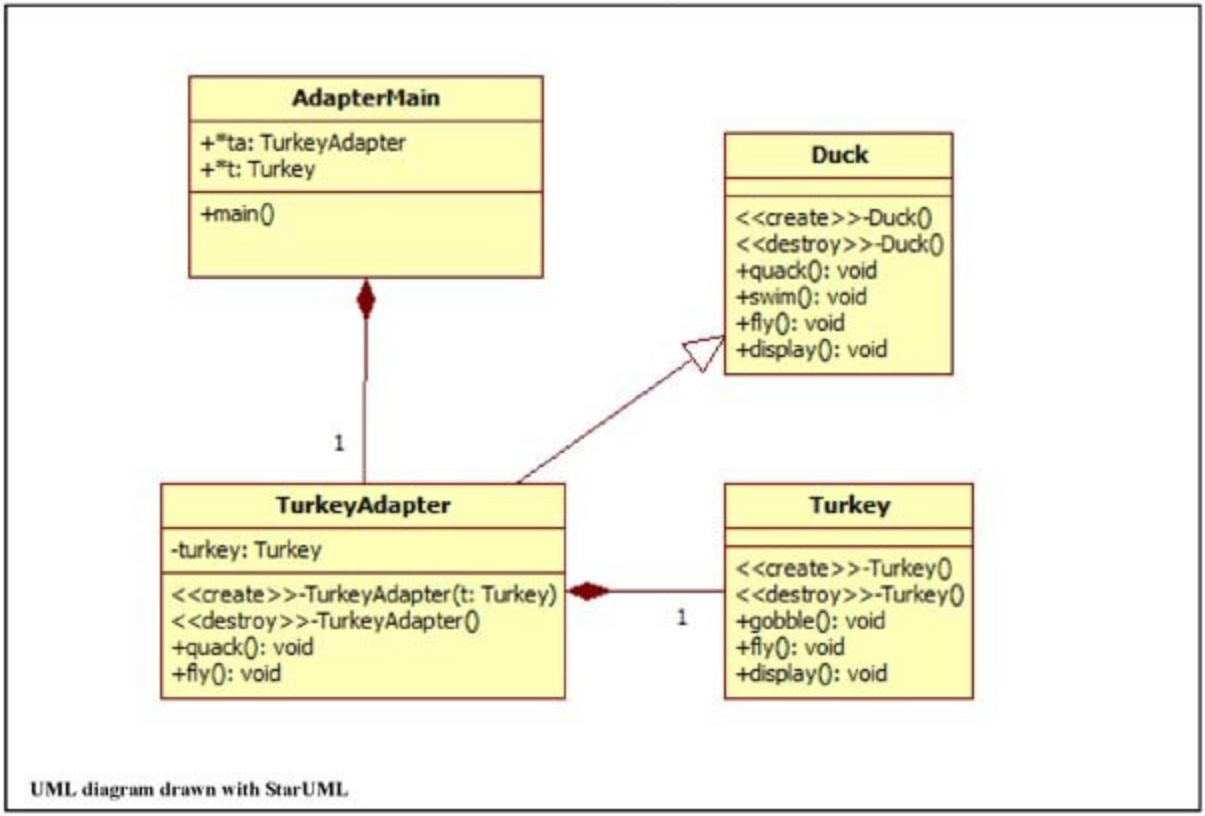
TurkeyAdapter

```
class TurkeyAdapter:Duck
{
   private:
        Turkey *turkey;
    public:
        TurkeyAdapter(
                Turkey *turkey)
        {
           this->turkey = turkey;
        }
        void quack()
        {
           turkey->gobble();
        }
        void fly()
        {
           for(int i=0; i<5; i++)
              turkey->fly();
        }
}
```

Turkey's can fly, but only in short spurts. They can't fly long distances like ducks.

Design Patterns: The Adapter

Code Sample



AdapterMain

Creates an instance of TurkeyAdapter and connects it to a Turkey

Makes Duck interface calls which the TurkeyAdapter translates for the Turkey.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Facade Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

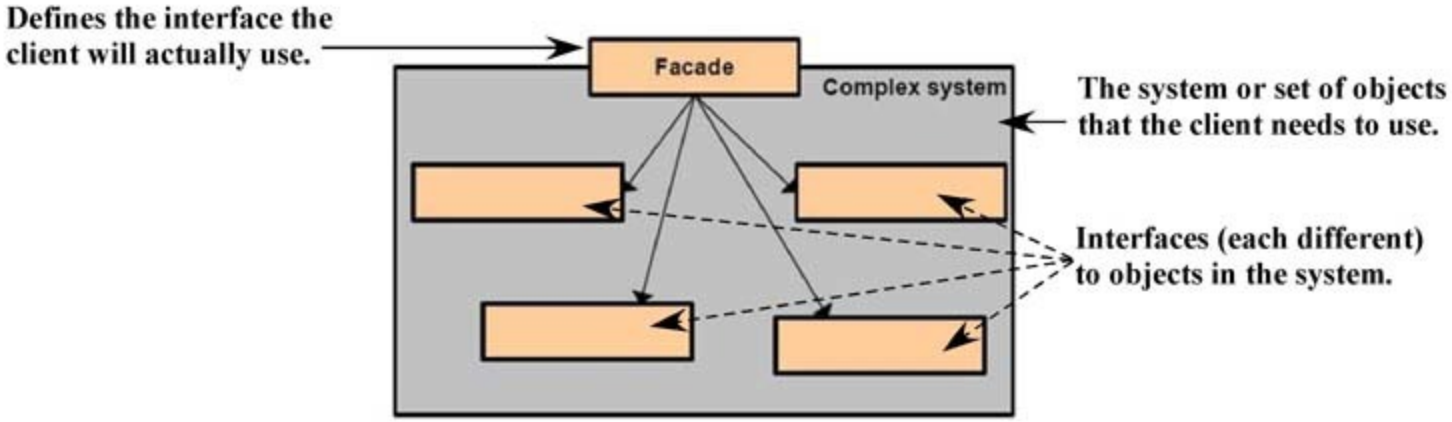
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

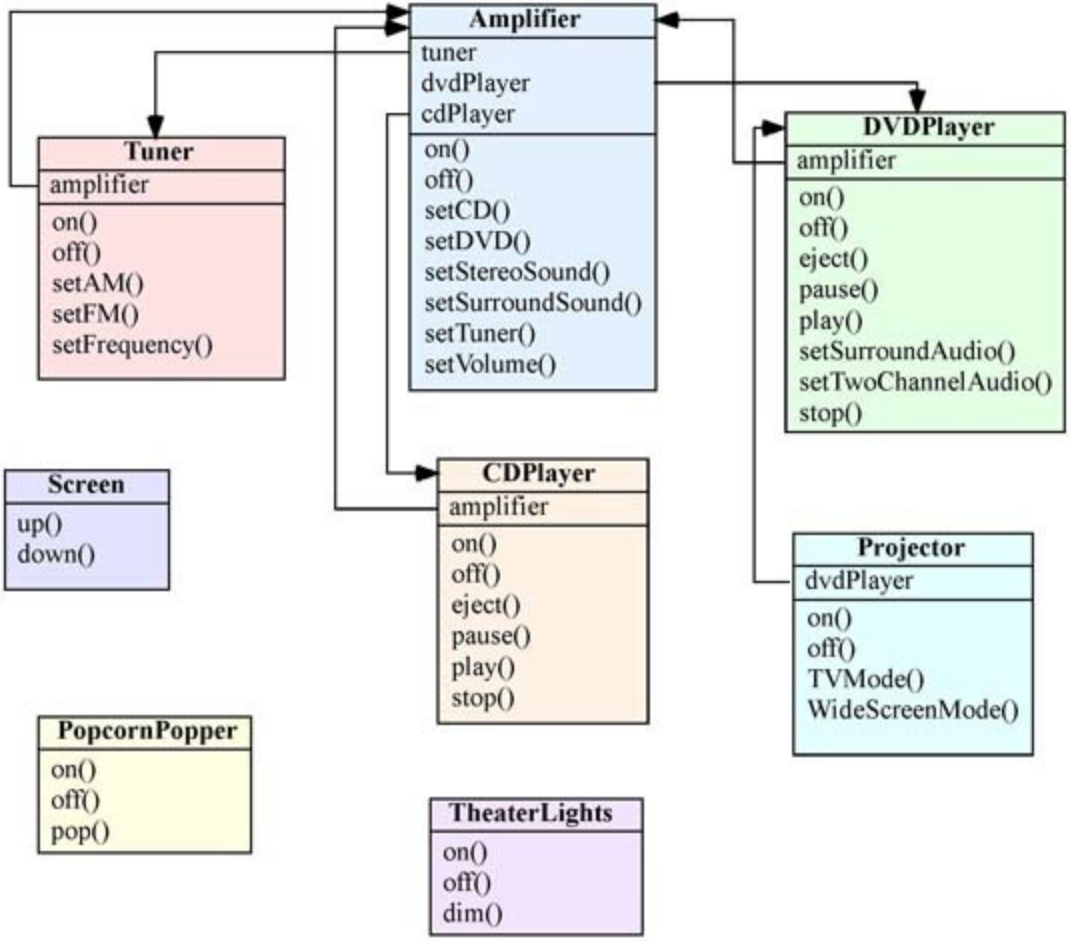
Design Patterns: The Facade
Quick Overview

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



Design Patterns: The Facade

The Home Theater System



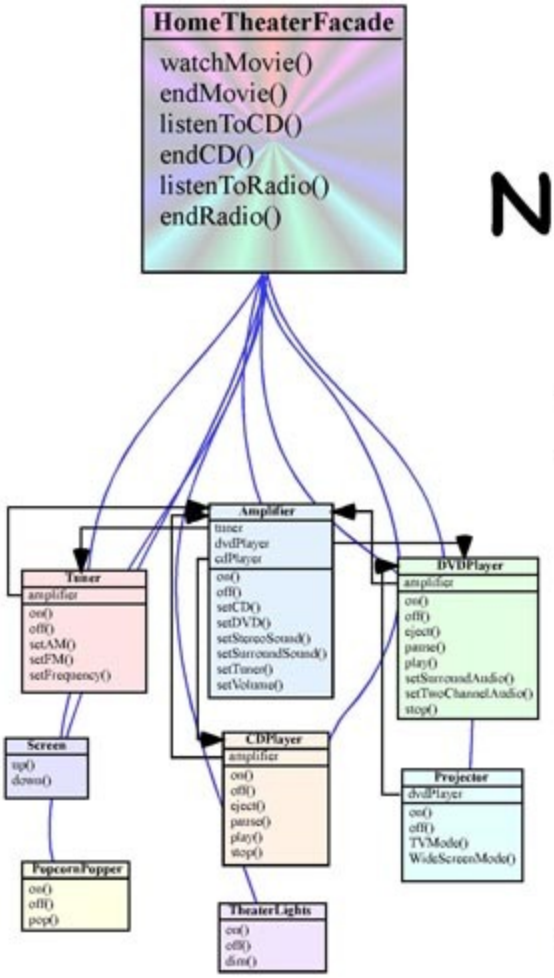
Design Patterns: The Facade

"I want to play a movie."

1. Turn on the popcorn popper.
2. Start the popper popping.
3. Dim the lights.
4. Put the screen down.
5. Turn the projector on.
6. Set the projector input to DVD.
7. Put the projector on wide-screen mode.
8. Turn the sound amplifier on.
9. Set the amplifier to DVD input.
10. Set the amplifier to surround sound.
11. Set the amplifier volume to medium (5).
12. Turn the DVD Player on.
13. Start the DVD Player playing.

And after the movie finishes playing you have to repeat everything, but in reverse order to turn the entire system off.

Design Patterns: The Facade



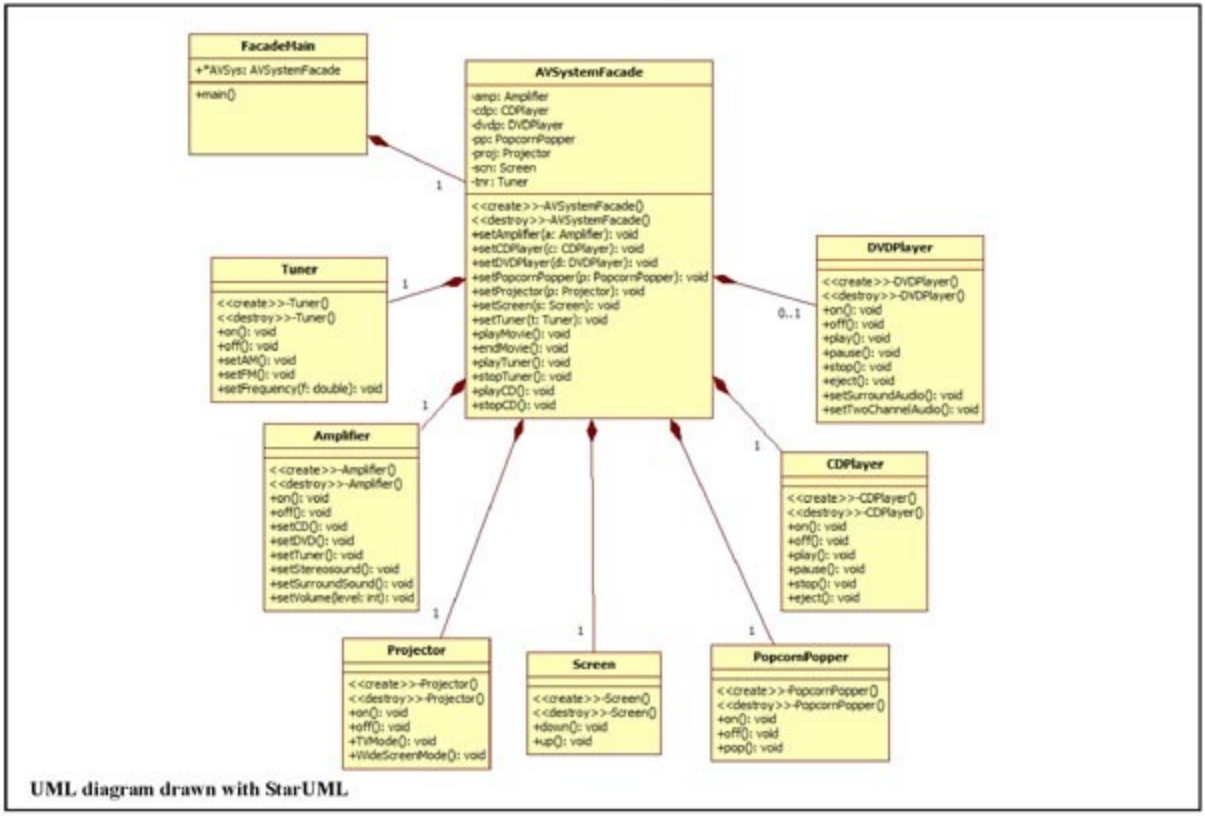
Now isn't that a whole lot easier?

Design Principles

- ***Principle of Least Knowledge - talk only to your immediate friends.***

Design Patterns: The Facade

Code Sample



UML diagram drawn with StarUML

FacadeMain

Demonstrates how to play a movie without the Facade.
Creates the home theater system.
Demonstrates how to play a movie with the Facade.
Demonstrates how to switch after the movie to play a CD.
Demonstrates how to play a CD with the Facade.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Template Method Pattern

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

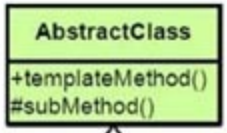
Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

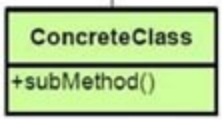
Design Patterns: The Template Method
Quick Overview

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.

The class that defines the interface for the algorithm possibly along with the common functionality.



The class that defines the specific steps for the algorithm inheriting the common functionality.



Design Patterns: The Template Method

Starbuzz Coffee Recipe

```
class Coffee
{
    public:
        void prepareRecipe()
        {
            boilWater();
            brewCoffeeGrinds();
            pourInCup();
            addSugarAndMilk();
        }
        void boilWater();
        void brewCoffeeGrinds();
        void pourInCup();
        void addSugarAndMilk();
}
```

Starbuzz Tea Recipe

```
class Tea
{
    public:
        void prepareRecipe()
        {
            boilWater();
            steepTeaBag();
            pourInCup();
            addSugarAndLemon();
        }
        void boilWater();
        void steepTeaBag();
        void pourInCup();
        void addSugarAndLemon();
}
```

Do you notice the similarities in these two Beverage "Algorithms"?

Design Patterns: The Template Method

Starbuzz Beverage Recipe

```
class CaffeineBeverage
{
    public:
        void prepareRecipe()
        {
            boilWater();
            brew();
            pourInCup();
            addCondiments();
        }
        void boilWater();
        virtual void brew();
        void pourInCup();
        virtual void addCondiments();
}
```

Now we have an Algorithm that is the same for both beveages.

Starbuzz Coffee Recipe

```
class Coffee:CaffeineBeverage
{
    public:
        void brew()
        {
            cout << "Dripping coffee through filter";
        }
        void addCondiments();
        {
            cout << "Adding sugar and milk";
        }
}
```

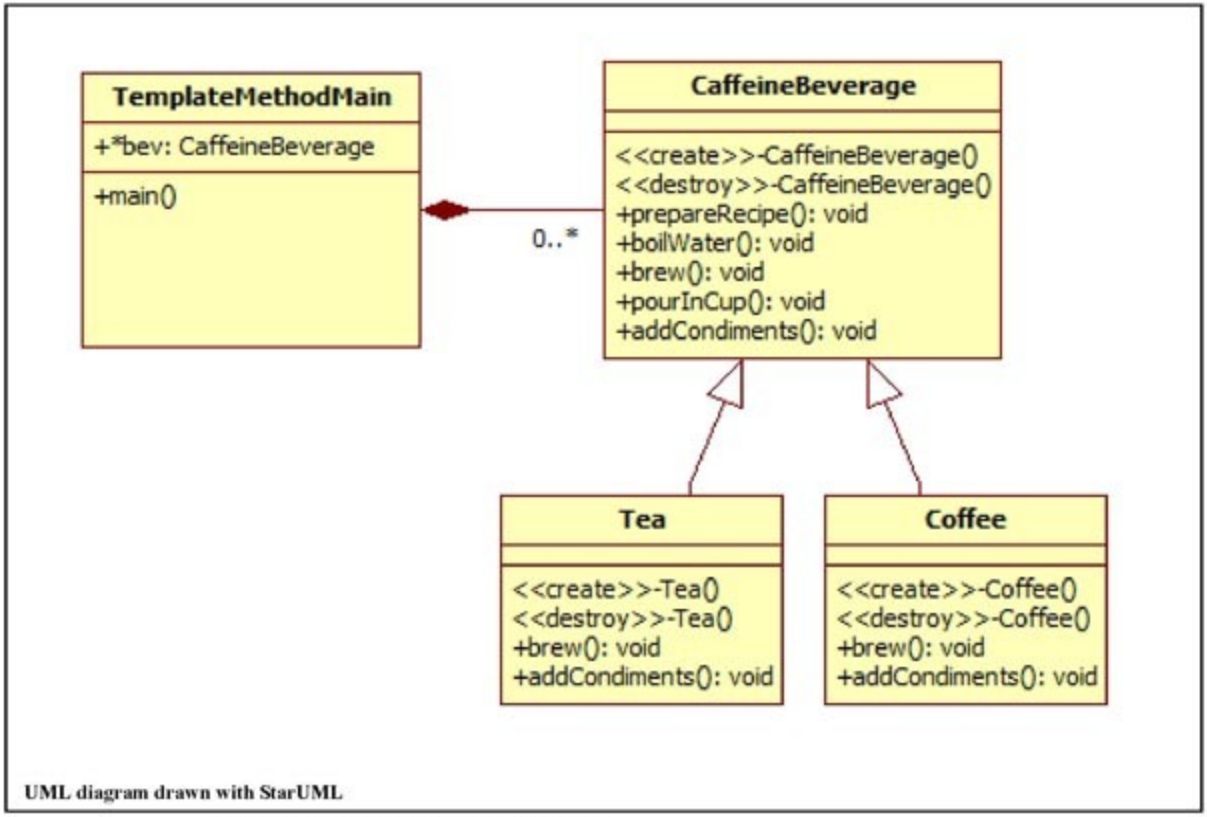
Starbuzz tea Recipe

```
class Tea:CaffeineBeverage
{
    public:
        void brew()
        {
            cout << "Steeping the tea";
        }
        void addCondiments();
        {
            cout << "Adding sugar and lemon";
        }
}
```

But we have deferred certain steps of the algorithm to sub-classes.

Design Patterns: The Template Method

Code Sample



TemplateMethodMain

- Creates an instance of Tea and Coffee using a pointer to a CaffeineBeverage
- Calls each of the preparation algorithm functions in each.
- Parent class handles all common steps.
- Sub-classes handle all specific steps.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Iterator Pattern

Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

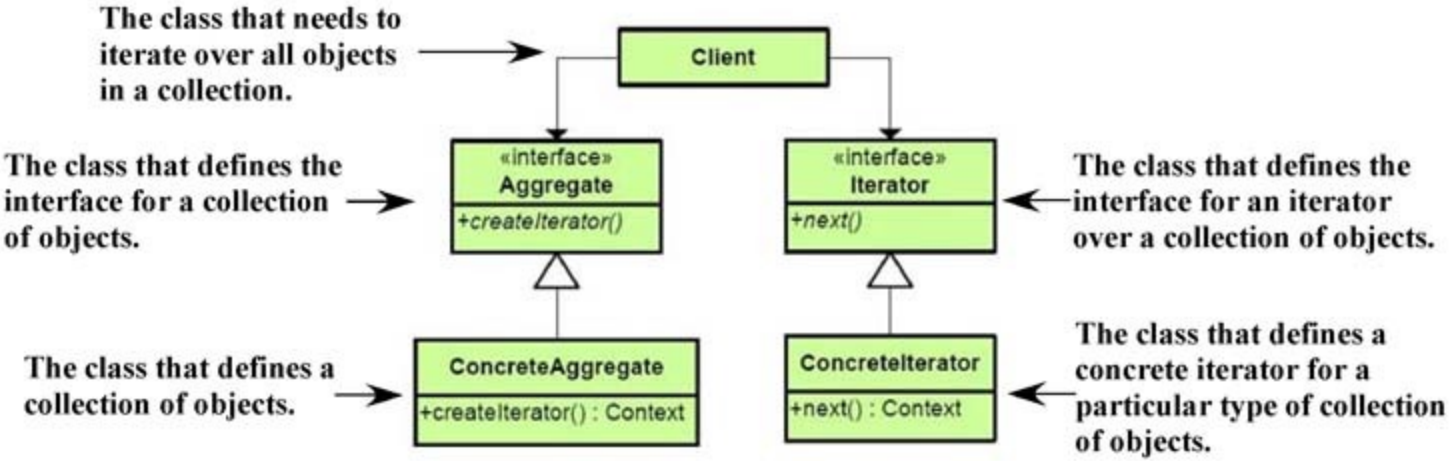
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Iterator Pattern
Quick Overview

Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.



Design Patterns: The Iterator Pattern

PancakeHouseMenu

ArrayList

MenuItem

string name
string description
bool vegetarian
double price

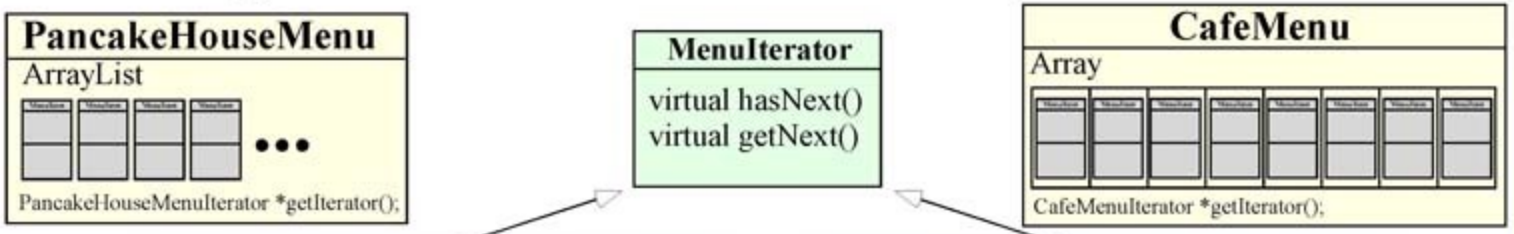
getName()
getDescription()
getPrice()
isVegetarian()

CafeMenu

Array

Hey Flo, whats on the menu?

Design Patterns: The Iterator Pattern



PancakeHouseMenuIterator

```
class PancakeHouseMenuIterator:MenuIterator
{
    private:
        ArrayList *items;
        int position = 0;
    public:
        PancakeHouseMenuIterator(
            ArrayList *items)
        {
            this->items=items;
        }
        MenuItem *getNext()
        {
            MenuItem *itm = items.item[position];
            position = position++;
            return itm;
        }
        bool hasNext()
        {
            return(position<(items.length()-1));
        }
}
```

CafeMenuIterator

```
class CafeMenuIterator:MenuIterator
{
    private:
        MenuItem *items;
        int count = 0;
        int position = 0;
    public:
        CafeMenuIterator(MenuItems *mArray
                int ct)
        {
            items = mArray;
            count = ct;
        {
        MenuItem *getNext()
        {
            MenuItem *itm = items[position];
            position = position++;
            return itm;
        {
        bool hasNext()
        {
            return(position<(count-1));
        {
{
```

Design Patterns: The Iterator Pattern

Let's have Flo give us a list of all items on both menus.

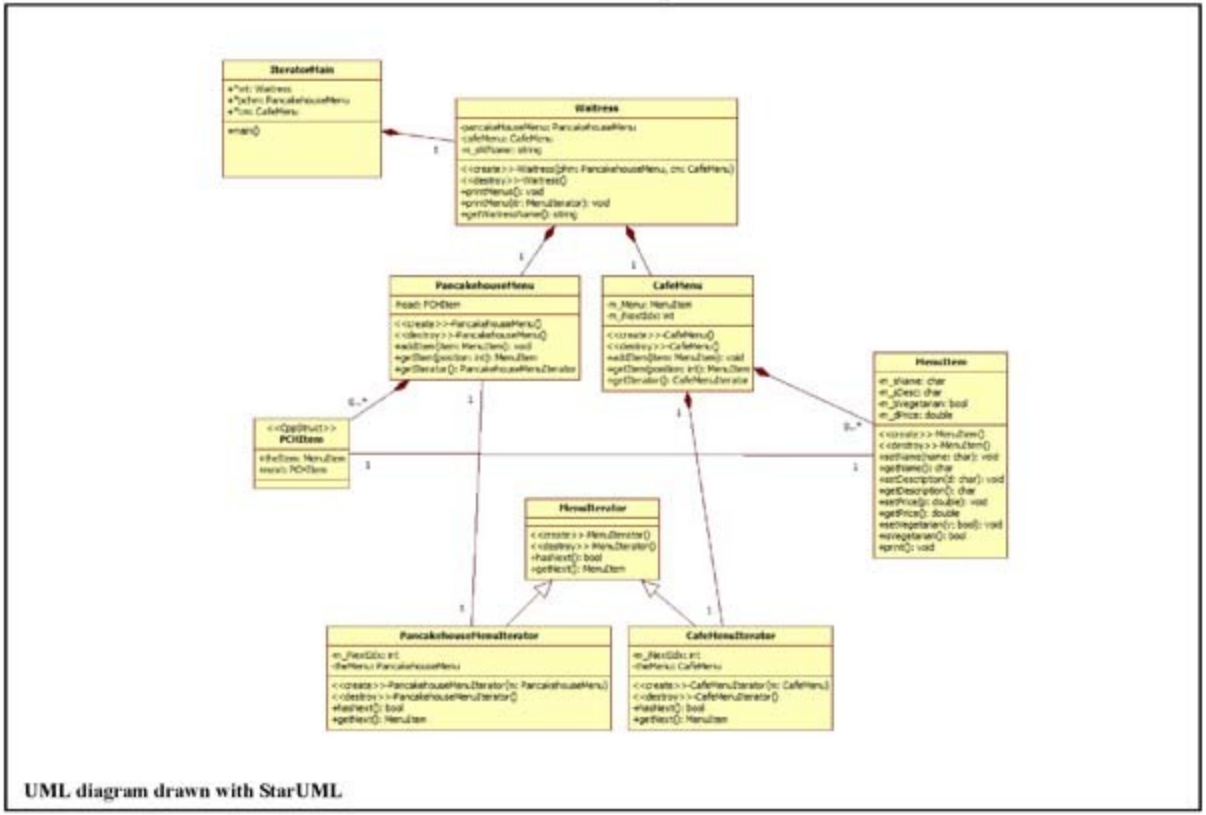
```
class Waitress
{
    private:
        PancakeHouseMenu *pancakeHouseMenu;
        CafeMenu *cafeMenu;
    public:
        Waitress(PancakeHouseMenu *phm, CafeMenu *cm)
        {
            this->pancakeHouseMenu=phm;
            this->cafeMenu = cm;
        }
        void printMenus()
        {
            MenuIterator *phIterator = pancakeHouseMenu->getIterator();
            MenuIterator *cIterator = cafeMenu->getIterator();
            cout << "MENU\n----\nBREAKFAST";
            printMenu(phIterator);
            cout << "MENU\n----\nLunch";
            printMenu(cIterator);
        }
        void printMenu(Iterator *itr)
        {
            while(itr->hasNext())
            {
                MenuItem *menuItem = itr->getNext();
                menuItem->print();
            }
        }
}
```

Design Principles

- ***A class should have only one reason to change.***

Design Patterns: The Iterator Pattern

Code Sample



UML diagram drawn with StarUML

IteratorMain

Creates an instance of PancakeHouseMenu and CafeMenu and gives them to Waitress

Calls Waitress.printMenus().

Waitress:

Get an iterator for each Menu and uses it to access all instances of MenuItem

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Composite Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

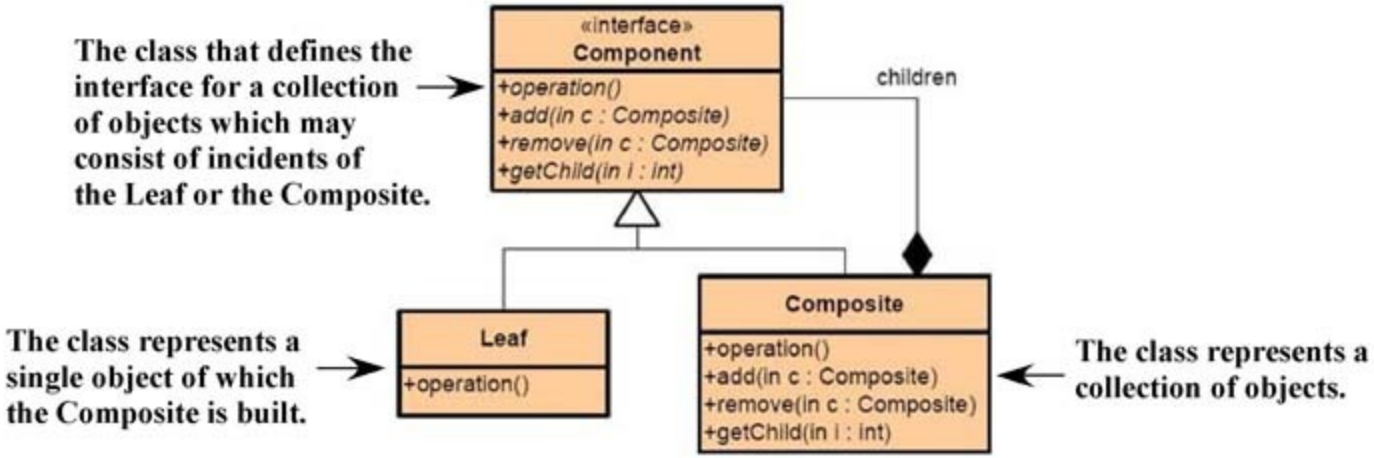
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Composite Pattern
Quick Overview

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



Design Patterns: The Composite Pattern

Here we go again!

PancakeHouseMenu

ArrayList

PancakeHouseMenuIterator *getIterator();

CafeMenu

Array

CafeMenuIterator *getIterator();

DinerMenu

HashTable

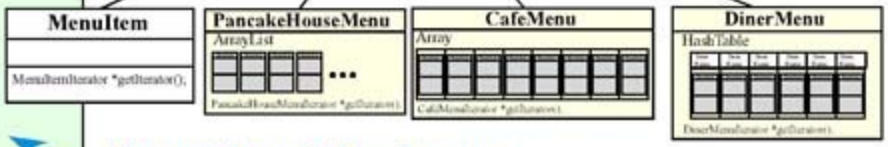
DinerMenuIterator *getIterator();

How many times am I going to have to change the way I do things?

Design Patterns: The Composite Pattern

MenuItemIterator

```
class MenuItemIterator:MenuIterator
{
    private:
        bool notAccessed=true;
            MenuItem *theItem;
    public:
        MenuItemIterator(MenuItem *m)
        {
                theItem = m;
            }
        MenuItem *next()
        {
            notAccessed = false;
            return theItem;
        }
        bool hasNext()
        {
            return(notAccessed);
        }
}
```



This class now lets us treat different implementations of the MenuItem containers the same as the primitive.

New subclass of MenuIterator created for compatibility.

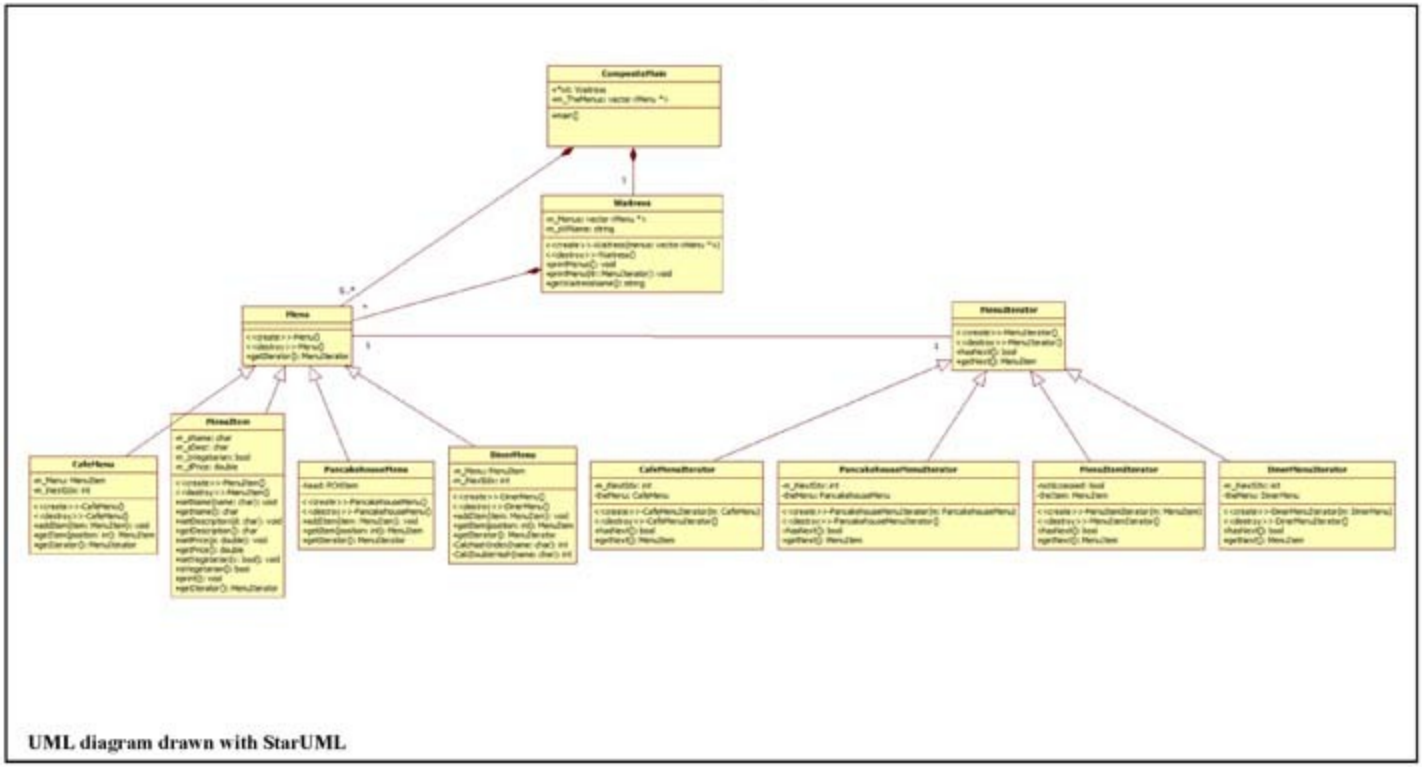
Change your menus all you want to now, Baby. I can handle it.

```
class Waitress
{
    private:
        vector <Menu>menus;
    public:
        Waitress(vector<Menu> menus)
        {
            this->menus = menus;
        }
        void printMenus()
        {
            for(vector<Menu>::iterator itr = menus.begin();
                    itr != menus.end(); itr++)
            {
                MenuIterator *mItr = itr->getIterator();
                printMenu(mItr);
            }
        }
```

```
        void printMenu(MenuIterator *itr)
        {
            while(itr->hasNext())
            {
                MenuItem *menuItem = itr->next();
                menuItem->print();
            }
        }
}
```

Design Patterns: The Composite Pattern

Code Sample



UML diagram drawn with StarUML

CompositeMain

Creates instances of CafeMenu, PancakeHouseMenu, DinerMenu, and single MenuItem

Calls Waitress.printMenus().

Waitress:

Get an iterator for each Menu or MenuItem and uses it to access all instances of MenuItem to call each print function.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The State Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

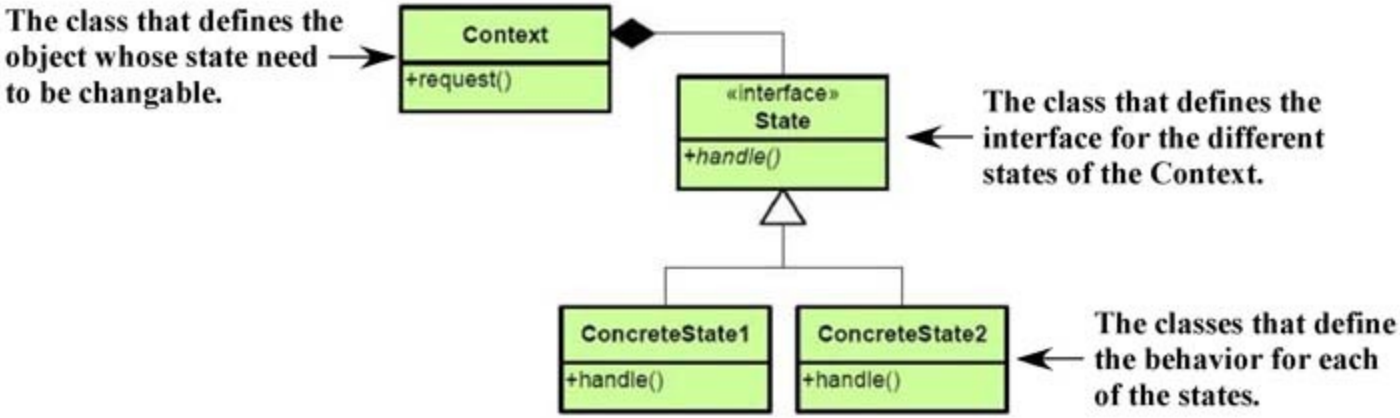
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
  Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

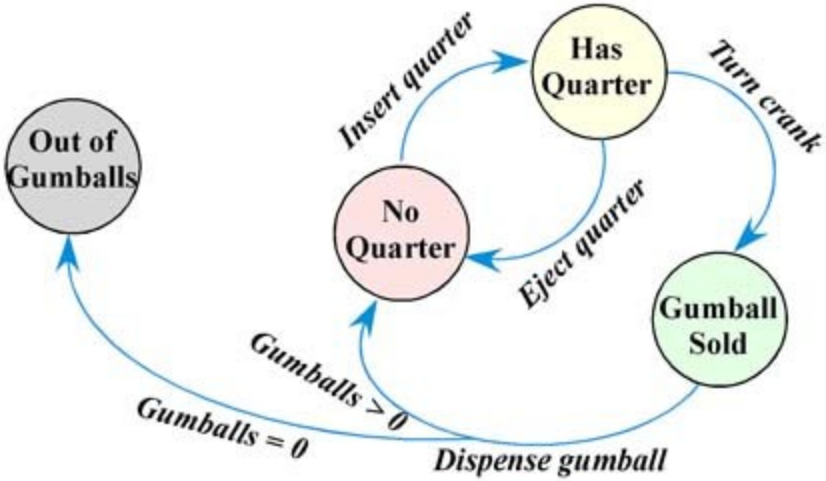
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The State Pattern
Quick Overview

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



Design Patterns: The State Pattern*



It's a state machine...

*State and Strategy were twins separated at birth.

Design Patterns: The State Pattern

Possible states

Actions to change states

Define an instance variable to hold the state

```
#define SOLD_OUT     0
#define NO_QUARTER   1
#define HAS_QUARTER  2
#define SOLD         3

int state = SOLD_OUT;
```

Create a class to act as a state machine

```
class GumballMachine
{
  private:
    int state = SOLD_OUT;
    int count = 0;
  public:
    GumballMachine::GumballMachine(int count)
    {
      this->count = count;
      if(count > 0) state = NO_QUARTER;
    }
    void insertQuarter()
    {
      if(state == HAS_QUARTER)
        cout << "You can't insert another quarter.\n";
      else if(state == NO_QUARTER)
      {
        state = HAS_QUARTER;
        cout << "You inserted a quarter.\n";
      }
      else if(state == SOLD_OUT)
        cout << "You can't insert a quarter. Machine is sold out.\n";
      else if(state == SOLD)
        cout << "Please wait. We are giving you a gumball.\n";
    }
    void ejectQuarter()
    {
      if(state == HAS_QUARTER)
      {
        cout << "Quarter returned.\n";
        state = NO_QUARTER;
      }
      else if(state == NO_QUARTER)
         cout << "You haven't inserted a quarter.\n";
      else if(state == SOLD_OUT)
        cout << "You can't eject. You haven't inserted a quarter yet.\n";
      else if(state == SOLD)
        cout << "Sorry, you already turned the crank.\n";
    }
```

...and so on for turnCrank(), and dispense()

Design Patterns: The State Pattern

"But, I know engineers. They love to change things."

Let's make it a game so that 10% of the time the customer gets two gumballs instead of one.


```
#define SOLD_OUT     0
#define NO_QUARTER   1
#define HAS_QUARTER  2
#define SOLD         4

class GumballMachine
{
  private:
    int state = SOLD_OUT;
    int count = 0;
  public:
    GumballMachine(int count)
    {
      this->count = count;
      if(count > 0) state = NO_QUARTER;
    }
    void insertQuarter()
    {
      // insert quarter code here
    }
    void ejectQuarter()
    {
      // eject quarter code here
    }
    void turnCrank()
    {
      // turn crank code here
    }
    void dispense()
    {
      // dispense code here
    }
```

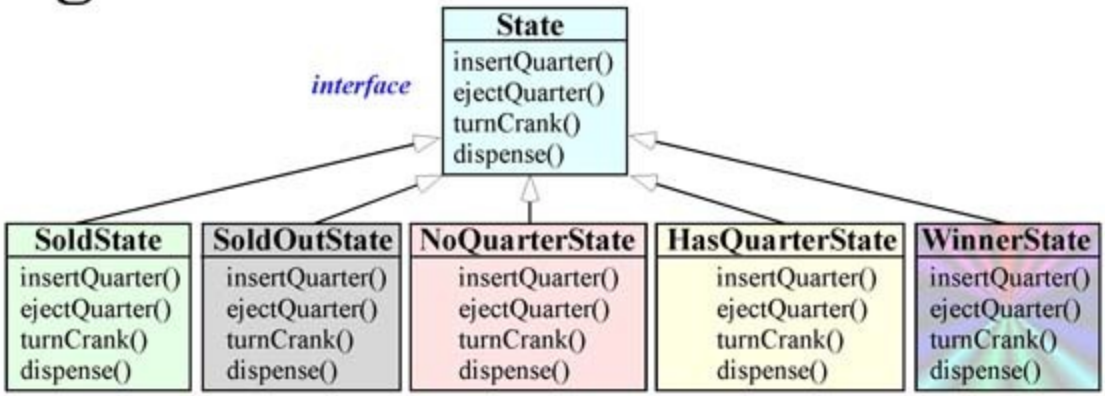
First you have to add a new WINNER state.

Then you have to add a new conditional in every method to handle the WINNER state.

This one gets especially messy because you have to add code to check for a winner then switch to either the WINNER state or the SOLD state.

"This is not pretty!"

Design Patterns: The State Pattern



Here's an example of how one state is implemented.

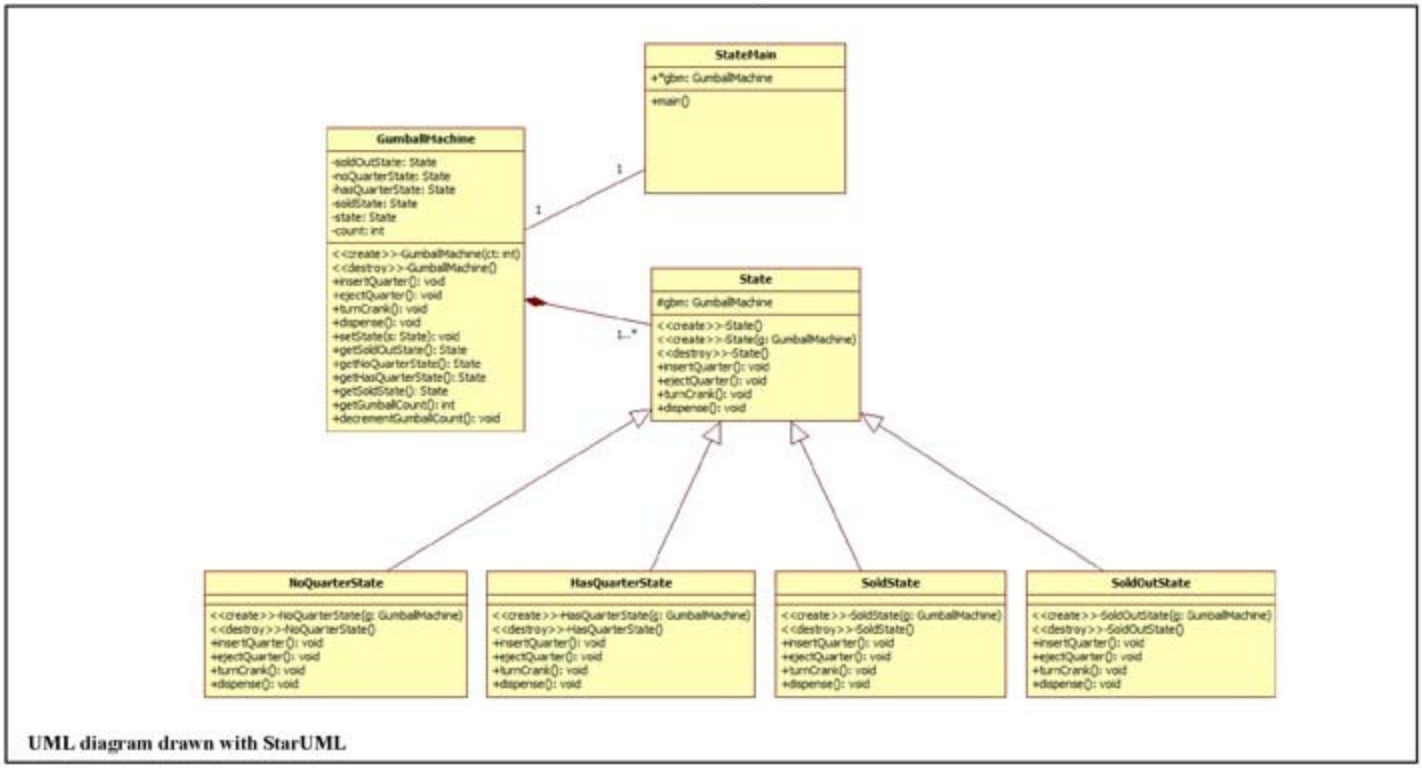
```
class NoQuarterState:State
{
  private:
    GumballMachine *gbm;
  public:
    NoQuarterState(GumballMachine *gbm)
    {
      this->gbm = gbm;
    }
    void insertQuarter()
    {
      gbm->setState(gbm->getHasQuarterState());
      cout << "You inserted a quarter.\n";
    }
    void ejectQuarter()
    {
      cout << "You haven't inserted a quarter.\n";
    }
    void turnCrank()
    {
      cout << "You turned but there's no quarter.\n";
    }
    void dispense()
    {
      cout << "You need to pay first.\n";
    }
};
```

Here's our revised GumballMachine

```
class GumballMachine
{
  private:
    State *soldOutState;
    State *noQuarterState;
    State *hasQuarterState;
    State *soldState;
    State *state;
    int count = 0;
  public:
    GumballMachine(int count)
    {
      soldOutState = new SoldOutState();
      noQuarterState = new NoQuarterState();
      hasQuarterState = new HasQuarterState();
      soldState = new SoldState;
      this->count = count;
      if(count > 0) state = noQuarterState;
      else state = soldOutState;
    }
    void insertQuarter() {state.insertQuarter();}
    void ejectQuarter() {state.ejectQuarter();}
    void turnCrank() {state.turnCrank();}
    void dispense() {state.dispense();}
    void setState(State *s) {state = s};
    State *getHasQuarterState(){
      return hasQuarterState;}
    // get() functions for all other states go here
}
```

Design Patterns: The State Pattern

Code Sample



UML diagram drawn with StarUML

StateMain

- Creates an instance of GumballMachine

GumballMachine

- Creates instances of each of its states

xxxState

- Executes appropriate behavior for each function
- Changes the State objet in GumballMachine when the appropriate event occurs.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Proxy Pattern

Provides a surrogate or place holder for another object to control access to it.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

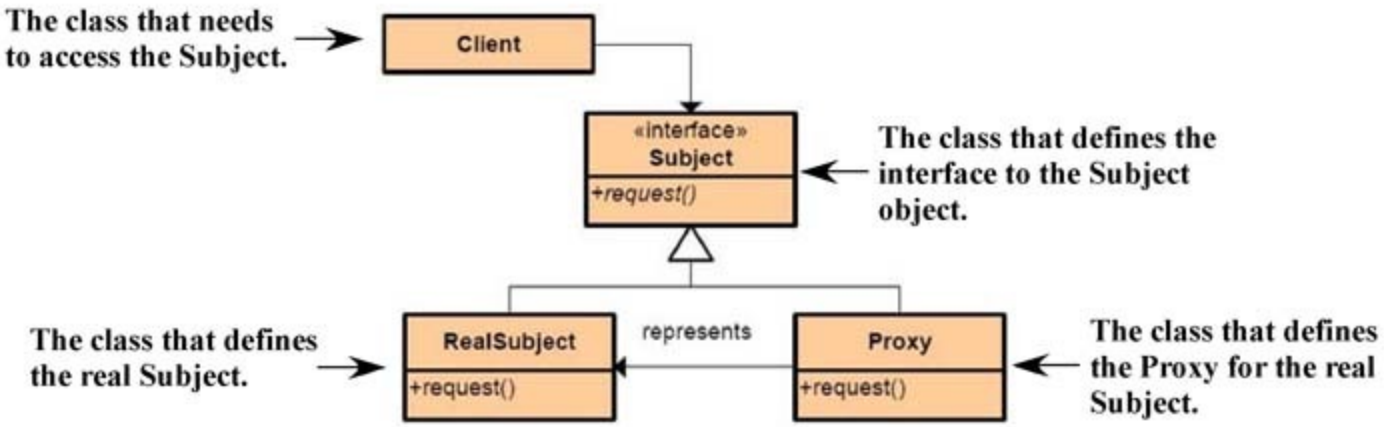
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
  Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

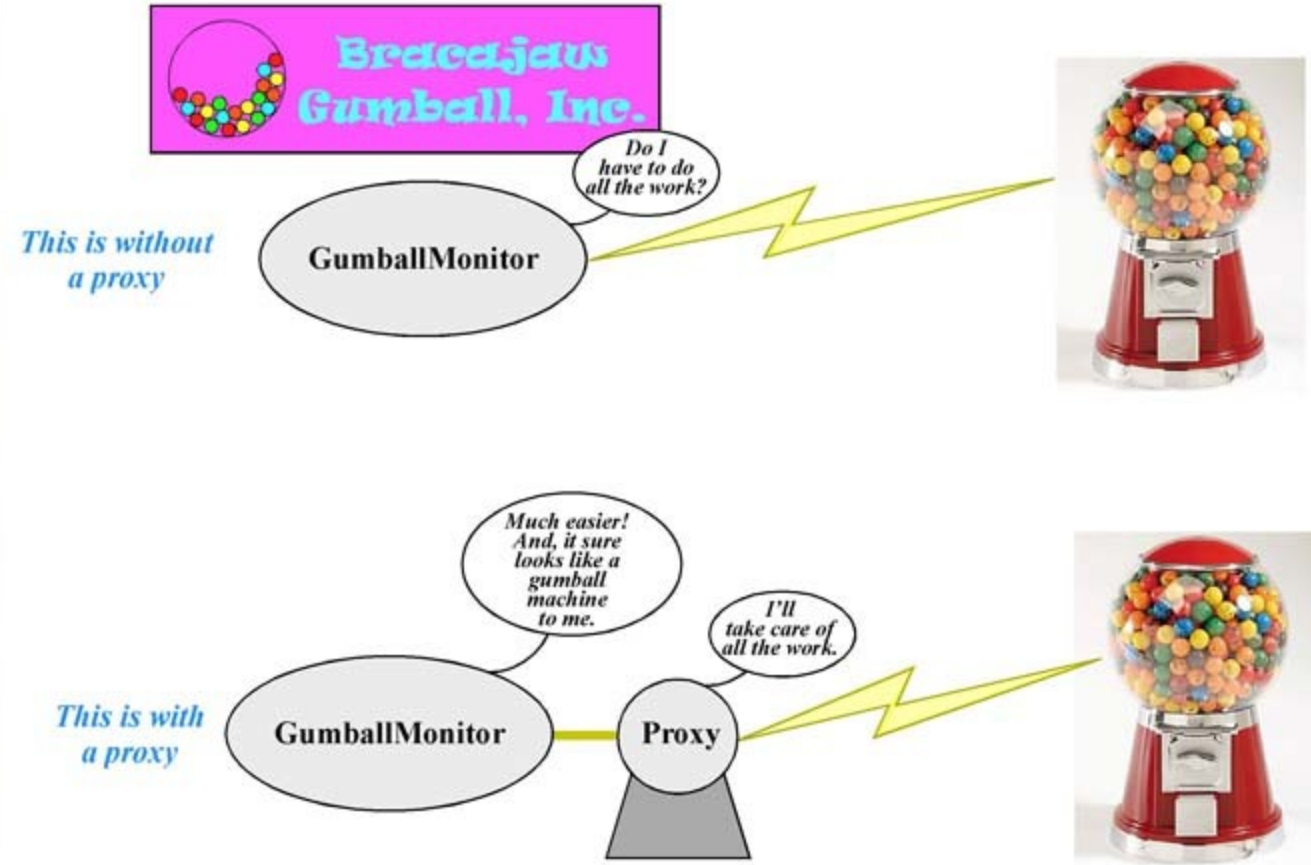
Design Patterns: The Proxy Pattern
Quick Overview

Provides a surrogate or place holder for another object to control access to it.

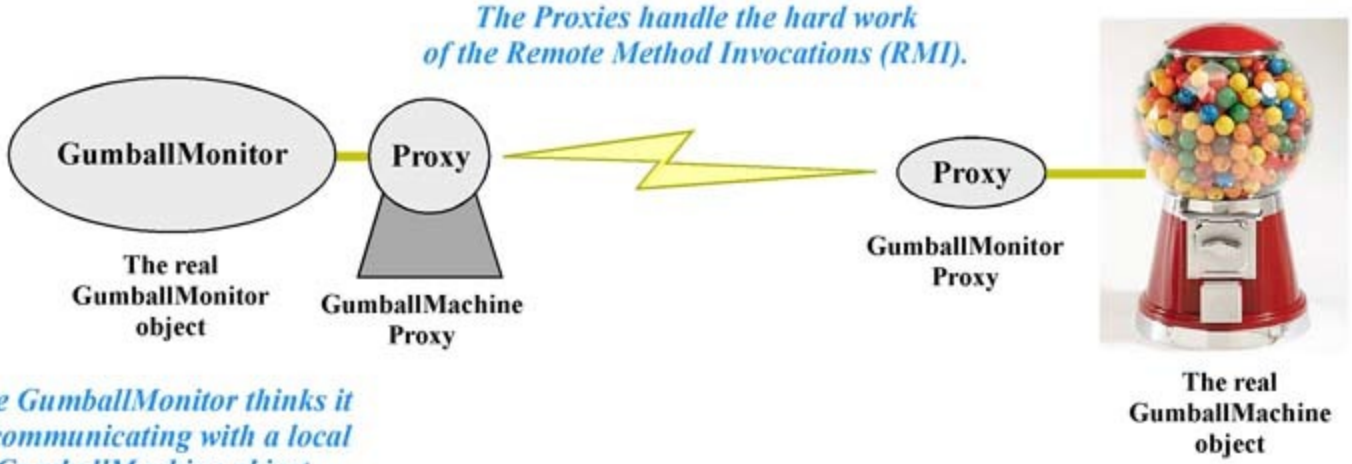


Design Patterns: The Proxy Pattern

A proxy is a stand in for another object



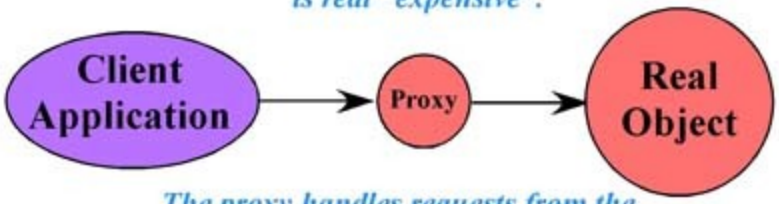
Design Patterns: The Proxy Pattern



Design Patterns: The Proxy Pattern

The Virtual Proxy

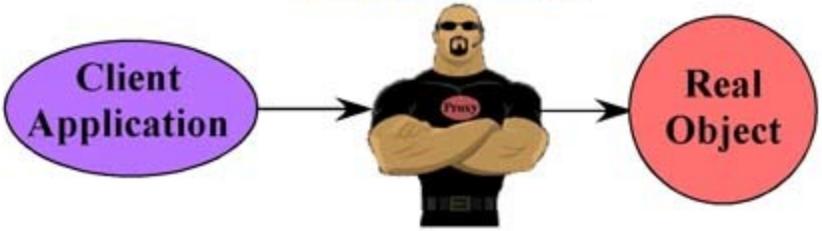
For when the real thing is real "expensive".



The proxy handles requests from the client until the real object is created. It may then disappear or handle requests by passing them on to the real object.

The Protection Proxy

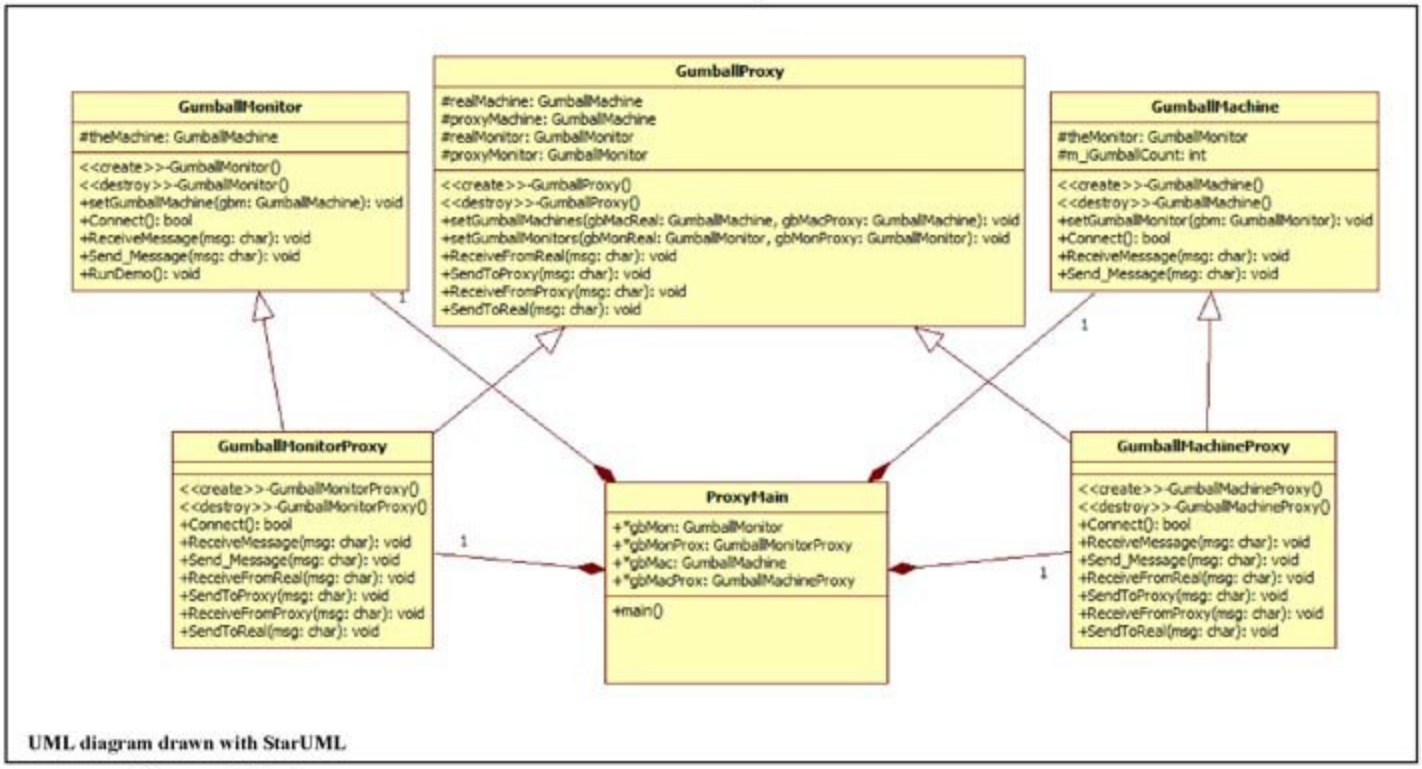
For when the real thing needs greater security.



The proxy handles requests from the client and allows access to the real object based on the client's access rights.

Design Patterns: The Proxy Pattern

Code Sample



UML diagram drawn with StarUML

ProxyMain

- Creates a GumballMachine connected to GumballMachineProxy connected to GumballMonitorProxy connected to GumballMonitor

GumballMonitor

- Passes messsages to GumballMachineProxy which relays messages to GumballMonitorProxy which relays messsages to GumballMachine

GumballMachine

- Passes messsages to GumballMonitorProxy which relays messages to GumballMachineProxy which relays messsages to GumballMonitor

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Bridge Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

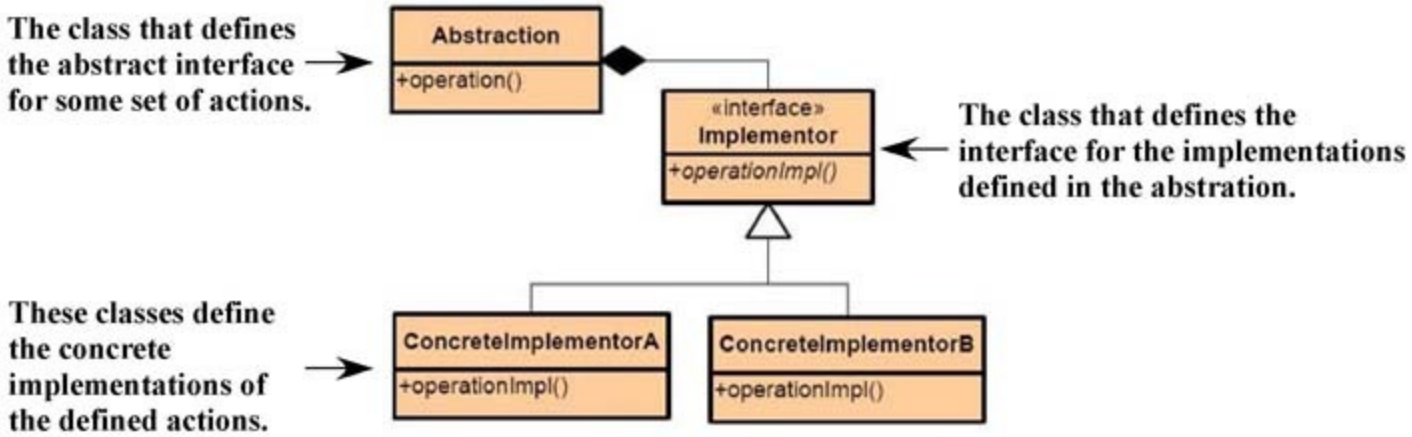
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
  Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **The Proxy Pattern**
  Provides a surrogate or place holder for another object to control access to it.
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Bridge Pattern
Quick Overview

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

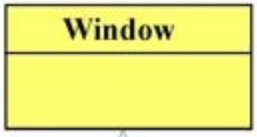


Design Patterns: The Bridge Pattern

Not the Bridge Pattern

Suppose you are writing an application that uses a GUI.
And, you want it to be portable to different platforms.

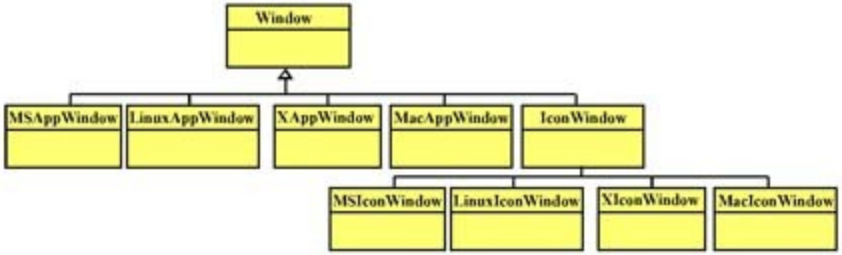
This defines the abstract interface.



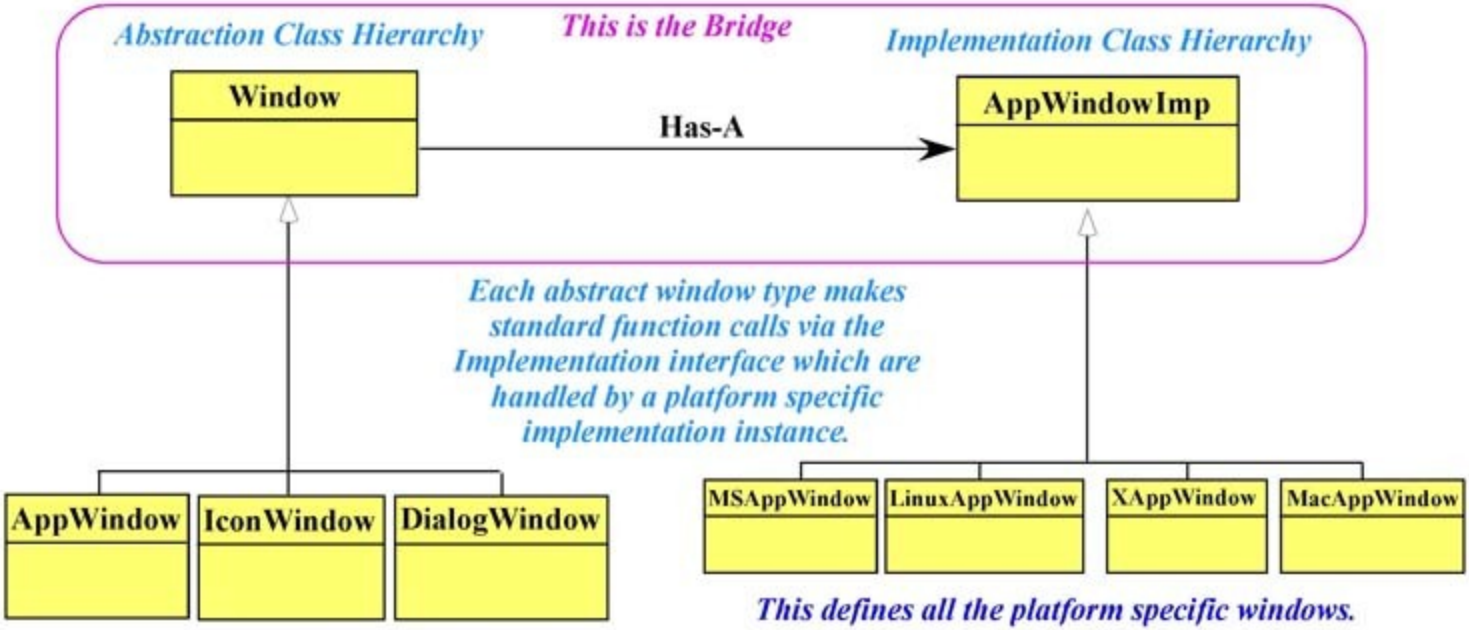
These define the platform specific instances

Problem: What happens when we have to add other window types (icon window, dialog box window, etc.) and other platforms (Sun, SGI, etc.)

This gets messy in a hurry...



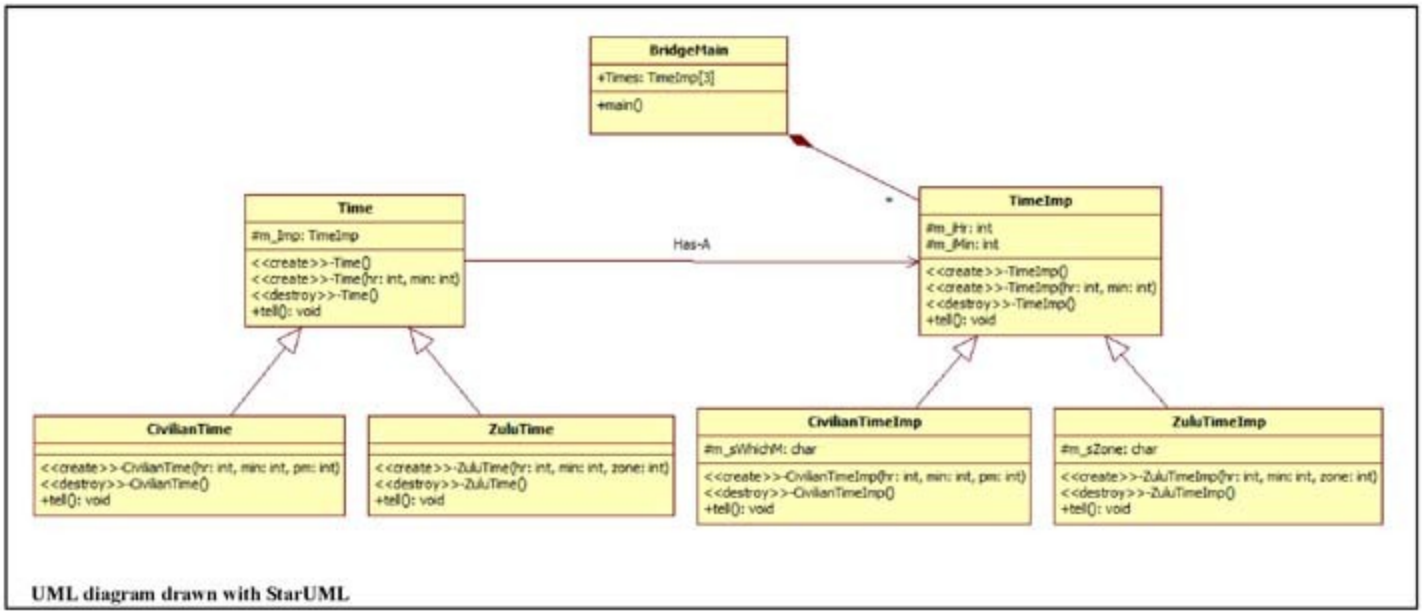
Design Patterns: The Bridge Pattern



Java uses this pattern to allow you to create and control any type of window (application, icon, dialog, etc.) you want in an application, but separates that from the actual platform implementation. Selecting the concrete implementation instance is postponed until run time.

Design Patterns: The Bridge Pattern

Code Sample



UML diagram drawn with StarUML

BridgeMain

Creates references to Time objects: CivilianTime and ZuluTime interfaces

CivilianTime has a reference to a TimeImp set to a CivilianTimeImp

ZuluTime has a reference to a TimeImp set to a ZuluTimeImp

Calls tell() on each of its Time objects

Each Time subclass calls tell() on its TimeImp object.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Builder Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

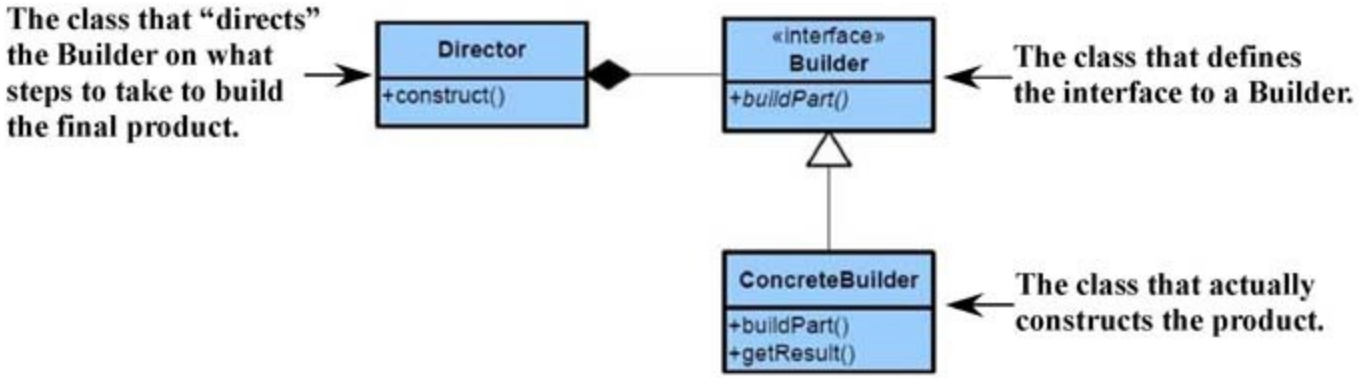
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
  Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **The Proxy Pattern**
  Provides a surrogate or place holder for another object to control access to it.
- **The Bridge Pattern**
  Decouples an abstraction from its implementation so that the two can vary independently
- **The Flyweight Pattern**

Behavioral Patterns

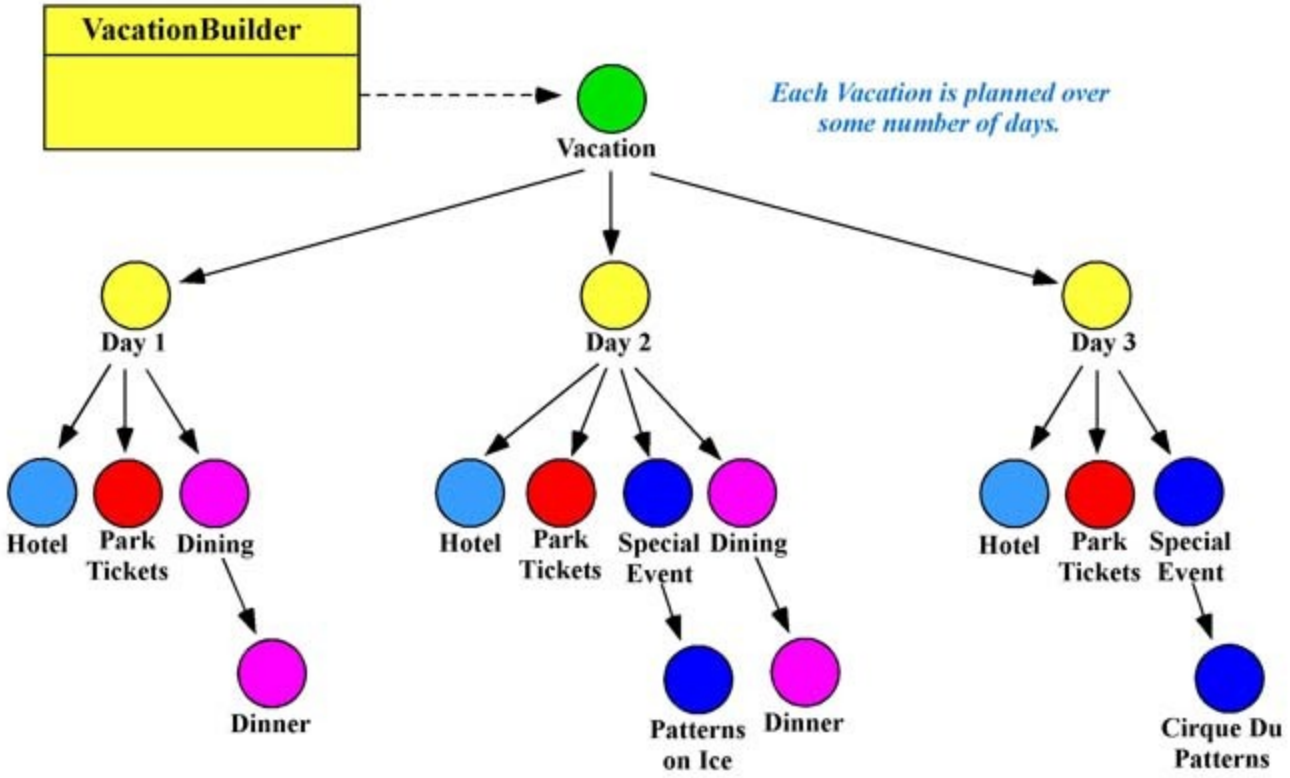
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Builder Pattern
Quick Overview

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



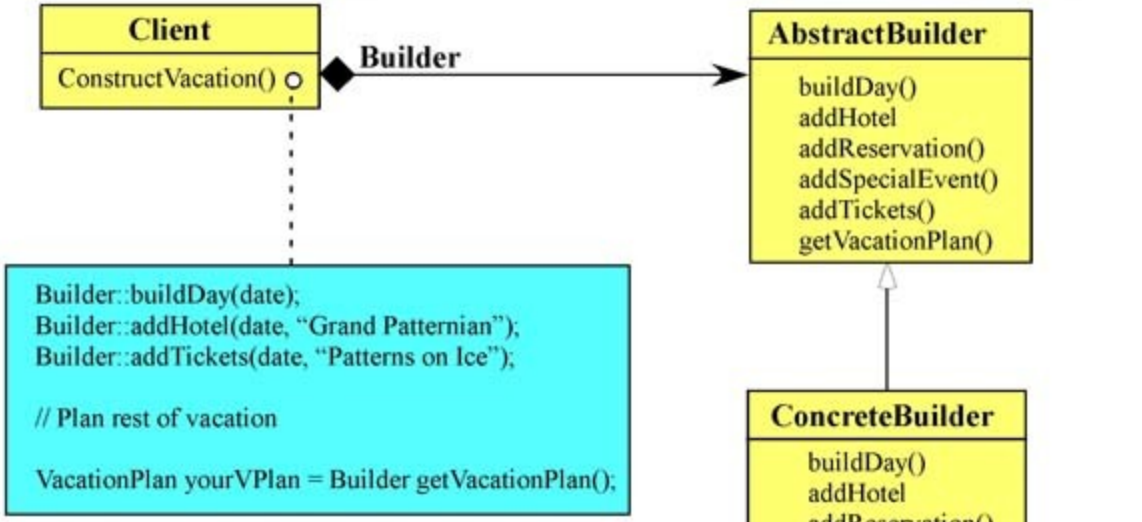
Design Patterns: The Builder Pattern



Design Patterns: The Builder Pattern

The client directs the builder to construct the object.

The client uses an abstract interface to build the object.



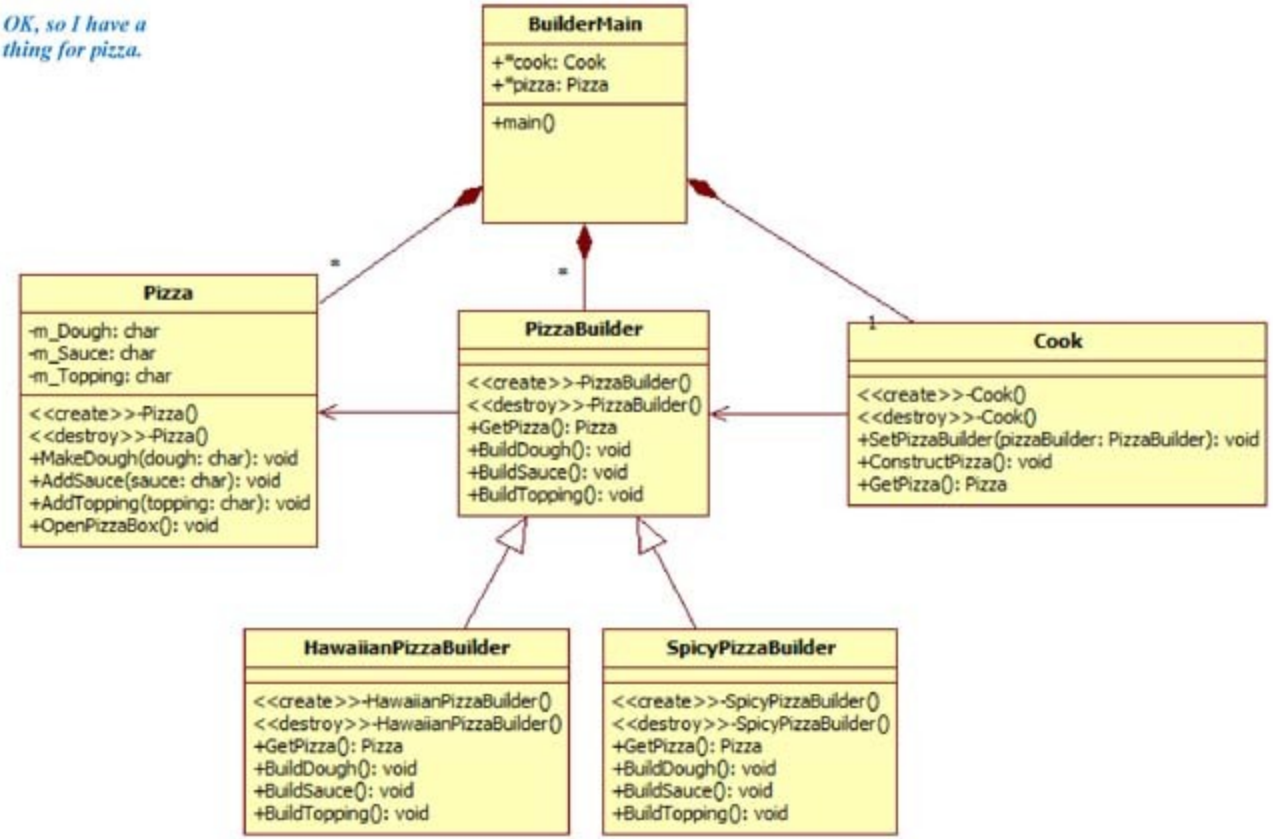
The client directs the builder to create the object in a number of steps and then calls the getResult() method to retrieve the complete object.

The concrete builder creates real products and stores them in the object composite structure.

Design Patterns: The Builder Pattern

Code Sample

OK, so I have a thing for pizza.



UML diagram drawn with StarUML

BuilderMain

Create instance of Cook

Cook

Creates instances of HawaiianPizzaBuilder and SpicyPizzaBuilder and uses them to build unique instances of different styles of Pizza.

Let's look at the code and run the demonstration.

Design Pattern Definitions from the GoF Book

The Chain of Responsibility Pattern

Avoids 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.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
  Separates the construction of a complex object from its representation so that the same construction process can create different representations.
- **The Prototype Pattern**

Structural Patterns

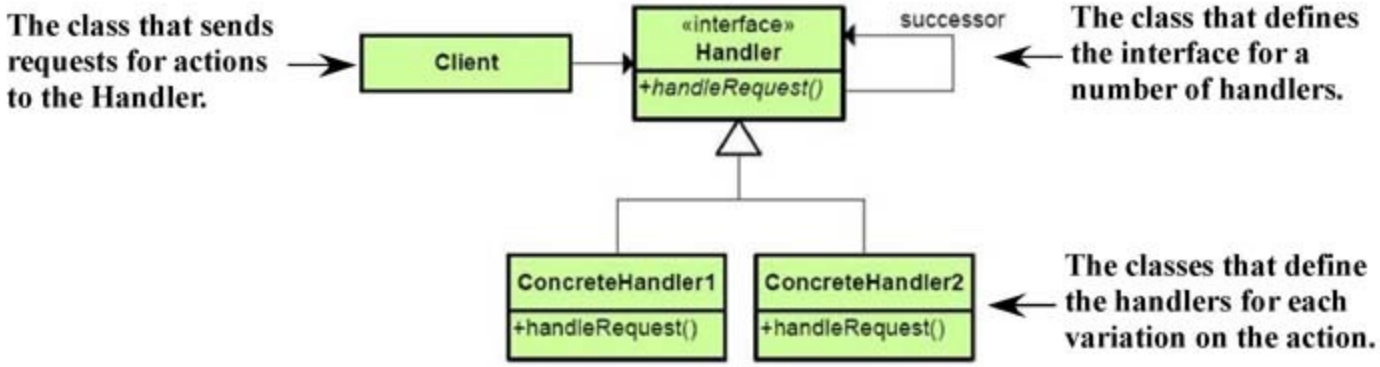
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **The Facade Pattern**
  Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **The Composite Pattern**
  Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **The Proxy Pattern**
  Provides a surrogate or place holder for another object to control access to it.
- **The Bridge Pattern**
  Decouples an abstraction from its implementation so that the two can vary independently
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
  Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets sub-classes redefine certain steps of an algorithm without changing the algorithm's structure.
- **The Iterator Pattern**
  Provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation.
- **The State Pattern**
  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

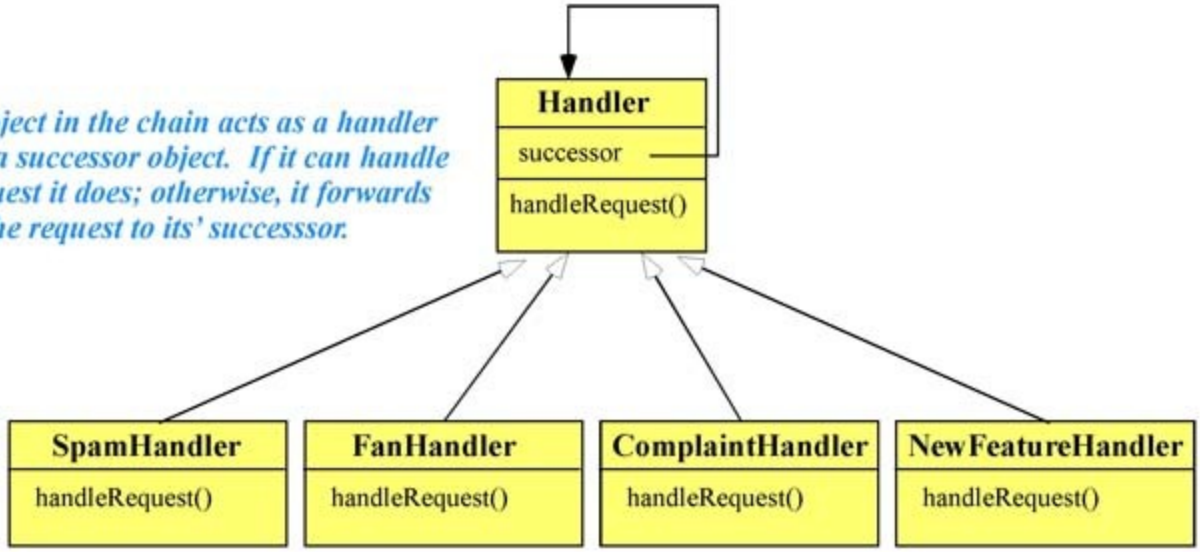
Design Patterns: The Chain of Responsibility Pattern
Quick Overview

Avoids 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.



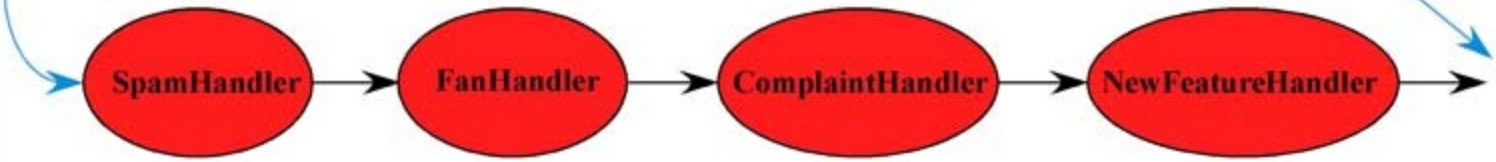
Design Patterns: The Chain of Responsibility Pattern

Each object in the chain acts as a handler and has a successor object. If it can handle the request it does; otherwise, it forwards the request to its' successsor.



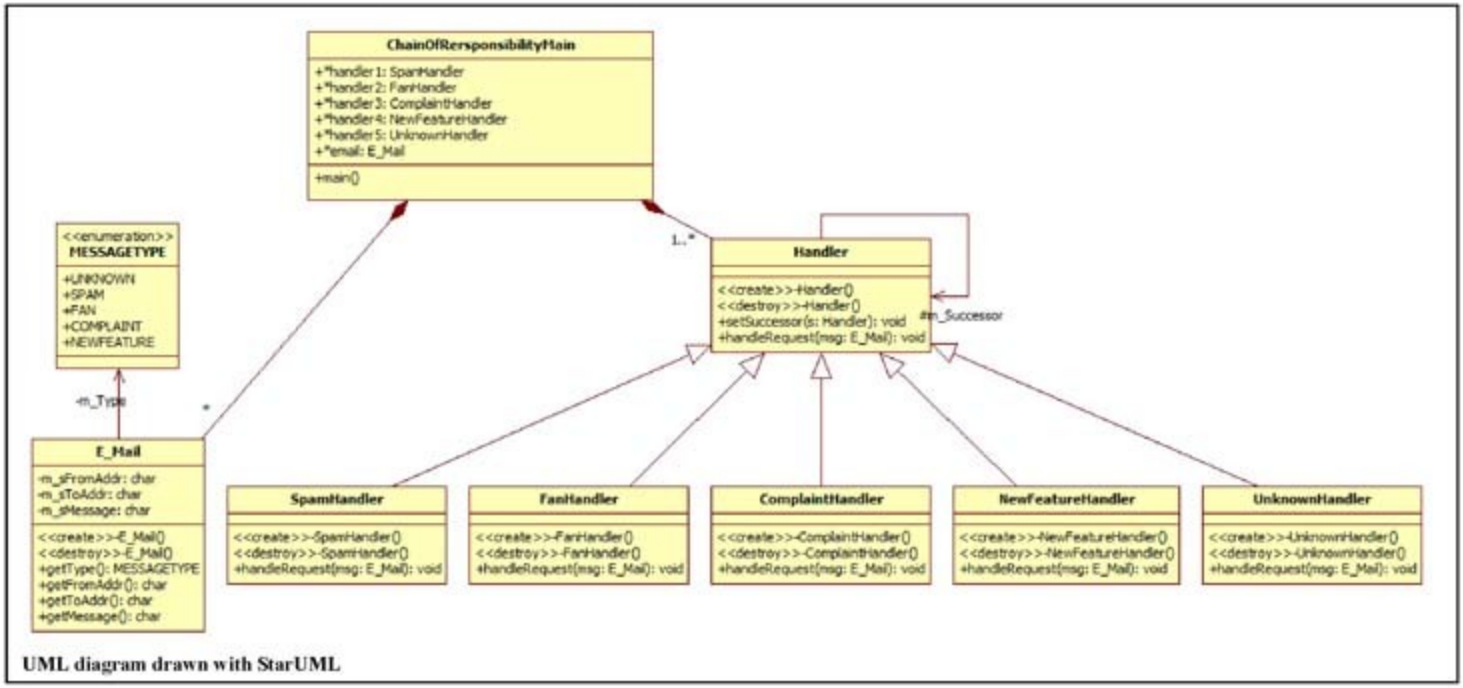
Each e-mail is passed to the first handler.

E-mail is not handled if it falls off the end of the chain, although you can always add a catch-all handler.



Design Patterns: The Chain of Responsibility Pattern

Code Sample



UML diagram drawn with StarUML

ChainOfResponsibilityMain

Create linked list of concrete handlers.

Generate random instances of E_Mail each set to a different type

Pass E_Mail instances to first Handler in the list

xxxHandler

If E_Mail is a type this Handler handles do so

else pass the E_Mail to the next Handler in the list.

Let's look at the code and run the demonstration.