Introduction

Instructor: Dr. Metin Özkan

Ref:

What Is A Pattern?

- Current use comes from the work of the architect Christopher Alexander
- Alexander studied ways to improve the process of designing buildings and urban areas
- "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."
- Hence, the common definition of a pattern: "A solution to a problem in a context."
- Patterns can be applied to many different areas of human endeavor, including software development

Ref:

Why Patterns?

- "Designing object-oriented software is hard and designing reusable object-oriented software is even harder." - Erich Gamma
- Experienced designers reuse solutions that have worked in the past
- Well-structured object-oriented systems have recurring patterns of classes and objects
- Knowledge of the patterns that have worked in the past allows a designer to be more productive and the resulting designs to be more flexible and reusable

Ref:

Software Patterns History

- 1987 - Cunningham and Beck used Alexander's ideas to develop a small pattern language for Smalltalk
- 1990 - The Gang of Four (Gamma, Helm, Johnson and Vlissides) begin work compiling a catalog of design patterns
- 1991 - Bruce Anderson gives first Patterns Workshop at OOPSLA
- 1993 - Kent Beck and Grady Booch sponsor the first meeting of what is now known as the Hillside Group
- 1994 - First Pattern Languages of Programs (PLoP) conference
- 1995 - The Gang of Four (GoF) publish the *Design Patterns book*

Ref:

The Unified Modeling Language (UML)

Instructor: Dr. Metin Özkan

The Unified Modeling Language - UML

- The UML is a visual language for specifying, constructing, and documenting the artifacts of a software.
- The UML is not a method to design systems, it is used to visualize the anaysis and design.
- It makes easier and to understand and document software systems.
- It supports teamwork because UML diagrams are more understandable than the program code.
- There are different kinds of UML diagrams, which are used in design and coding levels

The Unified Modeling Language - UML

- The current specification of the UML is available in the web site of the Obect Management Group (OMG). http://www.omg.org

The Unified Modeling Language - UML

- UML 2.2 has 14 types of diagrams divided into two categories: Seven diagram types represent *structural* information, and the other seven represent general types of *behavior*, including four that represent different aspects of *interactions*

Class Diagrams

- A class diagram shows the structure of the classes and the relationships between them.

- If necessary, access modes and data types may be also shown.

Ref: http://web.itu.edu.tr/buzluca/

Class Diagrams

- Comments: are placed in dog eared rectangles.
  - *You can use comments to put anything you want in a diagram. You can use comments to add application and program specific details.*
- Stereotypes: is a way of extending the UML in a uniform way, and remaining within the standart.
  - *You indicate a stereotype using: <<stereotype name>>*
- Constraints: is a text string in a curly braces ({usually language specific}) . The UML defines a language (Object Constrain Language - OCL) that you can use for writing constraints.

Class Diagrams (Relationship)

- A class diagram also shows the relationship between classes such as association, aggregation, composition, and inheritance
- Association: A general type of relationship. Objects of a class can send messages to objects of another class.

Class Diagrams (Relationship)

- Direction of the message flow:

Direction of messages is unspecified. Both may send messages to each other.

A can send messages to B.
A get a service from B.
B can not send messages to A.

- Multiplicity: indicates the number of any possible combination of objects of one class associated with objects from another class.
  - *In other words, it shows the number of objects from that class, that can be linked at runtime with one instance of the class at the other end of the association line*

  - *An instructor teaches zero or more courses. An association may also read in reverse order. A course is given exactly by one instructor.*

Class Diagrams (Relationship)

- Example: One object of class A is associated with one or more objects of class B at a time.
- Class A includes a list that can contain one or more objects of class B.

Class Diagrams (Relationship)

- Example

Class Diagrams (Relationship)

- Aggregation ("is part of"), composition ("is entirely made of"): Both are a type of association. They are qualified by a "has a" relationship.
- There is a small difference between them.
- Aggregation: It indicates a "Whole/part" relationship.
  - *A department of the faculty has instructors.*
  - *Parts (instructors) can still exist even if the whole (the department) does not exist.*
  - *The same part-object can belong to more than one object at a time.*

Class Diagrams (Relationship)

- Composition: is a strong kind of aggregation where the parts cannot exist independently of the "whole" object.
  - *Examples: A human has a head. A car has an engine.*
- A composition relation implies that:
  - *An instance of the part belongs to only one composite objects.*
  - *An instance of the part must belong to one composite object. It can not exist without the whole-object.*
  - *The composite is responsible for the creation and deletion of its part. If the composite is destroyed its parts must either be destroyed.*

Class Diagrams (Relationship)

- Example:

Class Diagrams (Relationship)

- Dependency: Relationship between the entities such that the proper operation of one entity depends on the presence of the other entity, and changes in one entity would affect the other entity.
- The common form of dependency is the *use* relation among classes.

Class Diagrams (Relationship)

- Example

Dependencies are most often omitted from the diagram unless they convey some significant information.

Class Diagrams (Relationship)

- Inheritance: The white triangular arrow point towards the class being extended.
- The arrow should point upwards. This is not a rule of UML, but it feels more logical and easier to read in this form.

- Abstract classes and pure virtual (abstract) methods are written with italic fonts.

Class Diagrams (Relationship)

- Example: Partial class diagram of a part of a system.

```
Class Teacher {
   private:
     Course * myCourses; //may be a linked list
   :
};

class Principal:public Teacher{
  private:
     School directedSchool;
     // or
     School *directedSchool;
};
```

```
Class School {
   private:
      vector<Student*> students;
   :
};

class Student{
  private:
     list<Course*> myCourses;
     :
};
```

Class Diagrams (Relationship)

- Example:

Class Diagrams (Relationship)

- Example:

Introduction to Design Patterns

Instructor: Dr. Metin Özkan

GoF Classification Of Design Patterns

- Purpose - what a pattern does
  - *Creational Patterns*
    - *Concern the process of object creation*
  - *Structural Patterns*
    - *Deal with the composition of classes and objects*
  - *Behavioral Patterns*
    - *Deal with the interaction of classes and objects*
- Scope - what the pattern applies to
  - *Class Patterns*
    - *Focus on the relationships between classes and their subclasses*
    - *Involve inheritance reuse*
  - *Object Patterns*
    - *Focus on the relationships between objects*
    - *Involve composition reuse*

GoF Classification Of Design Patterns

GoF Classification Of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

GoF Essential Elements Of Design Patterns

- Pattern Name
  - *Having a concise, meaningful name for a pattern improves communication among developers*
- Problem
  - *What is the problem and context where we would use this pattern?*
  - *What are the conditions that must be met before this pattern should be used?*
- Solution
  - *A description of the elements that make up the design pattern*
  - *Emphasizes their relationships, responsibilities and collaborations*
  - *Not a concrete design or implementation; rather an abstract description*
- Consequences
  - *The pros and cons of using the pattern*
  - *Includes impacts on reusability, portability, extensibility*

GoF Pattern Template

- Pattern Name and Classification
  - *A good , concise name for the pattern and the pattern's type*
- Intent
  - *Short statement about what the pattern does*
- Also Known As
  - *Other names for the pattern*
- Motivation
  - *A scenario that illustrates where the pattern would be useful*
- Applicability
  - *Situations where the pattern can be used*

GoF Pattern Template (Continued)

- Structure
  - *A graphical representation of the pattern*
- Participants
  - *The classes and objects participating in the pattern*
- Collaborations
  - *How to do the participants interact to carry out their responsibilities?*
- Consequences
  - *What are the pros and cons of using the pattern?*
- Implementation
  - *Hints and techniques for implementing the pattern*

GoF Pattern Template (Continued)

- Sample Code
  - *Code fragments for a sample implementation*
- Known Uses
  - *Examples of the pattern in real systems*
- Related Patterns
  - *Other patterns that are closely related to the pattern*

Welcome to Design Patterns

Let us start with a simple SimUDuck application

- Problem: A company makes a highly successful duck pond simulation game, SimUDuck.
- The game can show a large variety of duck species swimming and making quaking sounds.
- The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.

But now we need the ducks to FLY

- Problem: The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors.
- Solution: Just add a fly() method in the Duck class and all the ducks will inherit it.

But something went horribly wrong

- Problem: The executives failed to notice that not all subclasses of Duck should *fly*.

But something went horribly wrong

- Solution: Just override the fly() method in rubber duck.
- Problem: what happens when we add wooden decoy ducks to the program? They aren't supposed to be fly or quack…
- Solution: also override them.

How about an interface

- Problem: Inheritance probably wasn't the solution, because the executives now want to update the product every six months.
  - *The spec will keep changing and possibly override fly() and quack() for every new Duck subclass that's ever added to the program… forever.*
- Solution: make Flyable() interface with a fly() method. Only the ducks that are supposed to fly will implement that interface and have a fly() method…

How about an interface

- Problem: The Flyable and Quackable interface sounded promising at first.
  - *But, interfaces have no implementation code except Java, so no code reuse.*
  - *Also whenever you need to modify a behavior, you're forced to track down and change it in all the different subclasses where that behavior is defined. (probably introducing bugs)*
- Solution: take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't. (This concept forms the basis for almost every design pattern.)

Separating what changes from what stays the same

- Fly() and quack() are the parts of the Duck class that vary across ducks.
- To separate these behaviors from the Duck class, pull both methods out of the Duck class and create a new set of classes to represent each behavior.

Designing the Duck Behaviors

- The Duck behaviors will live in a separate class – a class that implements a particular behavior interface.
- That way, the Duck classes won't need to know any of the implementation details for their behaviors.
- We want to *assign* behaviors to the instances of Duck. For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific *type* of flying behavior.
- We can change the behavior of a Duck dynamically. In other words, we should include behavior setter methods in the Duck classes so that we can *change* the MallardDuck's flying behavior at *runtime*.
- Given these goals, let's look at the second design principle:
    - *Program to an interface, not an implementation.*

Implementing the Duck Behaviors

- With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes.
- New behavior can be added without modifying any existing behavior classes or touching any Duck classes.

Integrating the Duck Behavior

- The key is that a Duck will now delegate its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).
- **1) First we'll add two instance variables** to the Duck class.
  - *flyBehavior and quackBehavior*
  - ➢ *Replace the fly() and quack() methods with*
    - *performFly() and performQuack()*

Integrating the Duck Behavior

- **2) Now we implement performQuack:**

```
class Duck{
private:
        QuackBehavior *quackBehavior;
        // more
public:
        void performQuack(){quackBehavior->quack();}
};
```

Integrating the Duck Behavior

- **3)** Okay, **how the flyBehavior and quackBehavior instance variables are set.** Let's take a look at the MallardDuck class:

```
class MallardDuck:public Duck{
public:
        MallardDuck(){
                quackBehavior=new Quack();
                flyBehavior=new FlyWithWings();
        }
        void display(){
                cout<<"I'm a real Mallard duck"<<endl;
        }
};
```

Testing the Duck code

- **1)** Type and compile the Duck class below(Duck.h) and the MallardDuck class given before (MallardDuck.h):

```
class Duck{
protected:
        QuackBehavior *quackBehavior;
        FlyBehavior *flyBehavior;
public:
        Duck(){}
        virtual ~Duck() = 0;
        virtual void display()=0;
        void performQuack(){quackBehavior->quack();}
        void performFly(){flyBehavior->fly();}
        void swim(){
                cout<<"All ducks float, even decoys!"<<endl;
        }
};
```

Testing the Duck code

- **2)** Type and compile the FlyBehavior interface (FlyBehaviour.h) and the two behaviors (FlyWithWings.h and FlyNoWay.h):

```
class FlyBehavior{
public:
        virtual void fly()=0;
};

class FlyWithWings:public FlyBehavior{
public:
        void fly(){
                cout<<"I'm flying!!"<<endl;
        }
};

class FlyNoWay:public FlyBehavior{
public:
        void fly(){
                cout<<"I can't fly"<<endl;
        }
};
```

Testing the Duck code

- **3)** Type and compile the QuackBehavior interface (QuackBehaviour.h) and the three behaviors (Quack.h, MuteQuack.h and Squeak.h):

```
class QuackBehavior{
public:
        virtual void quack()=0;
};
class Quack:public QuackBehavior{
public:
        void quack(){
                cout<<"Quack"<<endl;
        }
};
class MuteQuack:public QuackBehavior{
public:
        void quack(){
                cout<<"<< silence >>"<<endl;
        }
};
class Squeak:public QuackBehavior{
public:
        void quack(){
                cout<<"Squeak"<<endl;
        }
};
```

Testing the Duck code

- **4)** Type and compile the test program (MiniDuckSimulator.cpp):

```
int main(){

        Duck *mallard=new MallardDuck();
        mallard->performQuack();
        mallard->performFly();

        return 0;
}
```

Setting behavior dynamically

- **1)Add two new methods to the Duck class:**
  - *Call these methods anytime to change the behavior of a Duck on the fly.*

```
void setFlyBehavior(FlyBehavior *fb){
        flyBehavior=fb;
}
void setQuackBehavior(QuackBehavior *qb){
        quackBehavior=qb;
}
```

Setting behavior dynamically

- **2)Make a new Duck type (ModelDuck.h)**:
  - *Call these methods anytime to change the behavior of a Duck on the fly.*

```
class ModelDuck:public Duck{
public:
        ModelDuck(){
                quackBehavior=new Quack();
                flyBehavior=new FlyNoWay();
        }
        void display(){
                cout<<"I'm a model duck"<<endl;
        }
};
```

Setting behavior dynamically

- **3)Make a new FlyBehavior type (FlyRocketPowered.h)**

```
class FlyRocketPowered:public FlyBehavior{
public:
        void fly(){
                cout<<"I'm flying with a rocket!"<<endl;
        }
};
```

Setting behavior dynamically

- **4)Change the test class (MiniDuckSimulator.cpp), add the ModelDuck, and make the ModelDuck rocket-enabled.**

```
int main(){
        Duck *mallard=new MallardDuck();
        mallard->performQuack();
        mallard->performFly();

        Duck *model=new ModelDuck();
        model->performFly();
        model->setFlyBehavior(new FlyRocketPowered());
        model->performFly();

        return 0;
}
```

The Big Picture on encapsulated behaviors

Think of each set of behaviors as a family of algorithms

Client makes use of an encapsulated family of algorithms for both flying and quacking

These behaviors "algorithms" are interchangeable.

Speaking of Design Patterns…

- Here, we applied a desig pattern called as

  STRATEGY pattern

- **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- References
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995*
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*

The Observer Patterns

Instructor: Dr. Metin Özkan

The Weather Monitoring Application Overview

- Problem: The three players in the system are
  - *the weather station (the physical device that acquires the actual weather data),*
  - *the WeatherData object (that tracks the data coming from the Weather Station and updates the display), and*
  - *the display that shows users the current weather condition.*

The Weather Monitoring Application Overview

- Task: Create an application that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

➢ *WeatherData class*

The Weather Monitoring Application Overview

- What do we know about the task
  - *The WeatherData class has getter methods for three measurement values: temperature, huminity, and barometric pressure.*
  - *The measurementChanged() method is called any time new measurement data is available. -> do not care how this method is called.*
  - *We need to implement three display elements that use the weather data. These displays must be updated each time WeatherData has new measurement.*
  - *The system must be expandable. Other developers can create new custom display elements. Users can change or add display elements to the application.*

`getTemperature(), getHumidity(), getPressure()`

`measurementChanged()`

The Weather Monitoring Application Overview

- Here is a first implementation possibility

```
class WeatherData{
        //instance variable declaration
public:
        void measurementChanged(){
                float temp=getTemperature();
                float humidity=getHumidity();
                float pressure= getPressure();

                currentConditionsDisplay.update(temp,huminity,pressure);
                statisticsDisplay.update(temp, huminity, pressure);
                forecastDisplay.update(temp, huminity, pressure);
        }
        //other WeatherData methods here
};
```

Grab the most recent measurements

Update display

The Weather Monitoring Application Overview

- What's wrong with the implementation?
  - *By this update methods, there is no way to add or remove other display elements without making changes to the program.*
  - *Update methods need to be encapsulated.*

  - *Look at Observer.*

The Observer Pattern

- To understand the observer pattern, consider the newspaper or magazine subscription work:
  - *1. A newspaper publisher publishes newspapers.*
  - *2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.*
  - *3. You unsubscribe when you don't want papers anymore, and they stop being delivered.*
  - *4. While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.*

The Observer Pattern

`Publishers + Subscribers = Observer Pattern`

- Only we call the publisher the SUBJECT and the subscribers the OBSERVERS.

D object registers/subscribes to subject object
A object removes/unsubscribes
Then,

The Observer Pattern defined

- **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.

Objects use this interface to register as observers and also to remove themselves from being observers

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Designing the Weather Station

Implementing the Weather Station

```
class Observer{
public:
        virtual void update(float temp, float humidity, float pressure)=0;
};

class Subject{
public:
        virtual void registerObserver(Observer *)=0;
        virtual void removeObserver(Observer *)=0;
        virtual void notifyObservers() const=0;
};

class DisplayElement{
public:
        virtual void display()const=0;
};
```

Implementing the Weather Station

```
class WeatherData:public Subject{
private:
          mutable list<Observer*> observers;
          float temperature;
          float humidity;
          float pressure;
public:
          WeatherData(){}
          void registerObserver(Observer* o){observers.push_back(o);}
          void removeObserver(Observer* o){observers.remove(o);}
          void notifyObservers() const{
                    for( list< Observer* >::iterator iter=observers.begin(); observers.end() !=
iter; ++iter ) {
                              Observer* observer = *iter;
                              observer->update( temperature, humidity, pressure );
                    }
          }
          void measurementChanged(){ notifyObservers();}

          void setMeasurements(float temperature, float humidity, float pressure){
                    this->temperature=temperature;
                    this->humidity=humidity;
                    this->pressure=pressure;
                    measurementChanged();
          }

          //Other WeatherData methods here
};
```

Implementing the Weather Station

```
class CurrentConditionsDisplay:private Observer, DisplayElement{
private:
          float temperature;
          float humidity;
          Subject  *weatherData;
public:
          CurrentConditionsDisplay(Subject* weatherData):temperature( 0.0 ), humidity( 0.0 ){
                    this->weatherData=weatherData;
                    weatherData->registerObserver(this);
          }

          ~CurrentConditionsDisplay() {
                    weatherData->removeObserver( this );
          }

          void update( float temperature, float humidity, float pressure ) {
                    this->temperature = temperature;
                    this->humidity = humidity;
                    display();
          }

          void display() const {
                    cout << "Current conditions: "          << temperature;
                    cout << " F degrees and " << humidity;
                    cout << "% humidity" << endl<< endl<< endl;
          }
};
```

Implementing the Weather Station

```
class StatisticsDisplay:private Observer, DisplayElement{
private:
        float maxTemp;
        float minTemp;
        float tempSum;
        int numReadings;
        WeatherData *weatherData;

public:
        StatisticsDisplay(WeatherData* weatherData)
        :maxTemp( 0.0 ), minTemp( 200.0 ),tempSum( 0.0 ), numReadings( 0 ){
                this->weatherData=weatherData;
                weatherData->registerObserver(this);
        }

        ~StatisticsDisplay() {
                weatherData->removeObserver( this );
        }
```

Implementing the Weather Station

```
        void update( float temp, float humidity, float pressure ) {
                tempSum += temp;
                numReadings++;

                if (temp > maxTemp) {
                        maxTemp = temp;
                }

                if (temp < minTemp) {
                        minTemp = temp;
                }
                display();
        }

        void display() const {
                cout << "Avg/Max/Min temperature = " << ( tempSum / numReadings );
                cout << "/" << maxTemp << "/" << minTemp << endl<< endl<< endl;
        }
};
```

Implementing the Weather Station

```
class ForecastDisplay:private Observer, DisplayElement{
private:
        float currentPressure;
        float lastPressure;
        WeatherData *weatherData;

public:
        ForecastDisplay(WeatherData* weatherData):currentPressure(29.92){
                this->weatherData=weatherData;
                weatherData->registerObserver(this);
        }

        ~ForecastDisplay() {
                weatherData->removeObserver( this );
        }

        void update( float temp, float humidity, float pressure ) {
                lastPressure = currentPressure;
                currentPressure = pressure;

                display();
        }
```

Implementing the Weather Station

```
        void display() const {
                cout << "Forecast: ";
                if( currentPressure > lastPressure ) {
                        cout << "Improving weather on the way!";
                }
                else if( currentPressure == lastPressure ) {
                        cout << "More of the same";
                }
                else if( currentPressure < lastPressure ) {
                        cout << "Watch out for cooler, rainy weather";
                }
                cout << endl<< endl<< endl;
        }
};
```

Create a test for the Weather Station

```
int main(){
        WeatherData *weatherData=new WeatherData();

        CurrentConditionsDisplay *currentDisplay=
                new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay *statisticsDisplay=
                new StatisticsDisplay(weatherData);
        ForecastDisplay *forecastDisplay=
                new ForecastDisplay(weatherData);

        weatherData->setMeasurements(80,65,30.4);
        weatherData->setMeasurements(82,70,29.2);
        weatherData->setMeasurements(78,90,29.2);

        return 0;
}
```

Create a test for the Weather Station

- References
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995*
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*

The Decorator Patterns

Instructor: Dr. Metin Özkan

The Starbuzz Coffee overview

- Problem: The Starbuzz Coffee is
  - *the fastest growing coffee shop around.*
  - *Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.*
  - *When they first went into business they designed their classes like this....*

The Starbuzz Coffee overview

- Problem: In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha, and have it all topped off with whipped milk.
  - *Starbuzz charges a bit for each of these, so they really need to get them built into their order system.*
  - *Here's their first attempt...*

The Starbuzz Coffee

- A Solution: Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha and whip...

The Starbuzz Coffee

- There are also problems and question marks.
  - *Price changes for condiments will force us to alter existing code.*
  - *New condiments will force us to add new methods and alter the cost method in the superclass.*
  - *We may have new beverages. For example, iced tea. The condiments may not be appropriate. For instance, tea inherits methods like hasWhip()*
  - *What if a customer wants a double mocha?*

The Open-Closed Principle

- Design Principle
  - *Classes should be open for extension, but closed for modification.*
- The goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.
- What do we get if we accomplish this?
  - *Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.*

Meet the Decorator Pattern

- representing the beverage plus condiment pricing scheme with inheritance has not worked out very well.
- So, here's what we'll do instead: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:
  1. *Take a DarkRoast object*
  2. *Decorate it with a Mocha object*
  3. *Decorate it with a Whip object*
  4. *Call the cost() method and rely on delegation to add on the condiment costs.*

Constructing a drink order with Decorators

1. We start with our DarkRoast object.

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

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

Constructing a drink order with Decorators

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

The Decorator Pattern defined

- **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The class diagram :**

Each component can be used on its own, or wrapped by a decorator.

The concreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

The ConcreteDecorator has an instance variable for the thing it decorate

Decorators can add new methods.

Decorators can extend the state of the component.

Decorating the Starbuzz Beverages

Beverage acts as our abstract component class.

The four concrete components, one per coffee type

Here are condiments decorators; they need to implement both cost() and getDescription().

An exercise

- Make a picture for what happens when the order is for a "double mocha soy latte with whip" beverage. Use the menu to get the correct prices, and draw your picture using the same format we used earlier (as follows):

Starbuzz Coffee

Coffees

House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99

Condiments

Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

Writing the Starbuzz code

- Let's start with Beverage and CondimentDecorator classes

```
class Beverage{
public:
        Beverage(){description="Unknown Beverage";}
        virtual string getDescription()
        {return description;}
        virtual double cost()=0;
protected:
        string description;
};


class CondimentDecorator:public Beverage{
public:
        virtual string getDescription()=0;
};
```

An abstract class with the two methods

getDescription() is already implemented; cost() will be implemented in the subclasses

Interchangeable with a Beverage, so the Beverage class may be extended.

Coding beverages

```
class Expresso:public Beverage{
public:
        Expresso(){description="Expresso";}
        double cost(){return 1.99;}
};

class HouseBlend:public Beverage{
public:
        HouseBlend(){description="House Blend Coffee";}
        double cost(){return .89;}
};

class DarkRoast:public Beverage{
public:
        DarkRoast(){description="Dark Roast Coffee";}
        double cost(){return .99;}
};

class Decaf:public Beverage{
public:
        Decaf(){description="Decaf Coffee";}
        double cost(){return 1.05;}
};
```

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. Now that we don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

Coding condiments

```
class SteamedMilk:public CondimentDecorator{
public:
        SteamedMilk(Beverage *beverage)
        {this->beverage=beverage;}
        string getDescription()
        {return beverage->getDescription()+", Steamed Milk";}
        double cost()
        {return .10+beverage->cost();}
private:
        Beverage *beverage;
};

class Mocha:public CondimentDecorator{
public:
        Mocha(Beverage *beverage)
        {this->beverage=beverage;}
        string getDescription()
        {return beverage->getDescription()+", Mocha";}
        double cost()
        {return .20+beverage->cost();}
private:
        Beverage *beverage;
};
```

We're going to instantiate Mocha with a reference to a Beverage using:

1. An instance variable to hold the beverage we are wrapping.

2. A way to set this instance variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

Coding condiments (continued)

```
class Soy:public CondimentDecorator{
public:
        Soy(Beverage *beverage)
        {this->beverage=beverage;}
        string getDescription()
        {return beverage->getDescription()+", Soy";}
        double cost()
        {return .15+beverage->cost();}
private:
        Beverage *beverage;
};

class Whip:public CondimentDecorator{
public:
        Whip(Beverage *beverage)
        {this->beverage=beverage;}
        string getDescription()
        {return beverage->getDescription()+", Whip";}
        double cost()
        {return .10+beverage->cost();}
private:
        Beverage *beverage;
};
```

Serving some coffees (An application)

- Here's some test code to make orders:

```
int main(){

        Beverage *beverage=new Expresso;
        cout<<beverage->getDescription()+"  $"<<beverage->cost()<<endl;


        Beverage *beverage2=new DarkRoast;
        beverage2=new Mocha(beverage2);
        beverage2=new Mocha(beverage2);
        beverage2=new Whip(beverage2);
        cout<<beverage2->getDescription()+"  $"<<beverage2->cost()<<endl;

        Beverage *beverage3=new HouseBlend;
        beverage3=new Soy(beverage3);
        beverage3=new Mocha(beverage3);
        beverage3=new Whip(beverage3);
        cout<<beverage3->getDescription()+"  $"<<beverage3->cost()<<endl;

        return 0;
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Serving some coffees (An application)

- The output of the application is

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995*

The Factory Patterns

Instructor: Dr. Metin Özkan

When you see “new”, think “concrete”.

- When you use `new` you are certainly instantiating a concrete class, so that’s definitely an implementation, not an interface.

```
Duck *duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!.

- When you have a whole set of related concrete classes, often you’re forced to write code like this:

```
Duck *duck;
if (picnic) {
    duck = new MallardDuck();
}
else if (hunting) {
    duck = new DecoyDuck();
}
else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don’t know until runtime which one we need to instantiate.

Identifying the aspects that vary

- Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

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

        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
        return pizza;
}
```

Identifying the aspects that vary

- **But you need more than one type of pizza...**
- So then you'd add some code that determines the appropriate type of pizza and then goes about making the pizza:

```
Pizza* 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;
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

But the pressure is on to add more pizza types

- You have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu:

```
Pizza* 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();
        } else if (type=="clam") {
           pizza = new ClamPizza();
        } else if (type=="veggie") {
           pizza = new eggiePizza();
       }
        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
        return pizza;
}
```

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Encapsulating object creation

- So now we know we'd be better off moving the object creation out of the orderPizza() method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas

```
if (type=="cheese")) {
        pizza = new CheesePizza();
 } else if (type=="pepperoni") {
        pizza = new PepperoniPizza();
 } else if (type=="clam") {
        pizza = new ClamPizza();
 } else if (type=="veggie") {
        pizza = new eggiePizza();
 }
```

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

        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
        return pizza;
}
```

First we pull the object creation code out of the orderPizza Method

place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

SimplePizza Factory

Encapsulating object creation

- **We've got a name for this new object: we call it a <u>Factory</u>.**
- Factories handle the details of object creation.
- Once we have a SimplePizzaFactory, our orderPizza() method just becomes a client of that object.
  - *Any time it needs a pizza it asks the pizza factory to make one. Gone are the days when the orderPizza() method needs to know about Greek versus Clam pizzas. Now the orderPizza() method just cares that it gets a pizza, which implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().*
- We've still got a few details to fill in here; for instance, what does the orderPizza() method replace its creation code with?
  - *Let's implement a simple factory for the pizza store and find out...*

Building a simple pizza factory

- We'll start with the factory itself. What we're going to do is define a class that encapsulates the object creation for all pizzas. Here it is...

Here's a new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

First define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

```
class SimplePizzaFactory {
public:
        Pizza* createPizza(string type) {
                Pizza *pizza;
                if (type=="cheese") {
                        pizza = new CheesePizza();
                } else if (type=="pepperoni") {
                        pizza = new PepperoniPizza();
                } else if (type=="clam") {
                        pizza = new ClamPizza();
                } else if (type=="veggie") {
                        pizza = new VeggiePizza();
                }
                return pizza;
        }
};
```

Reworking the PizzaStore class

- Now it's time to fix up our client code. What we want to do is rely on the factory to create the pizzas for us. Here are the changes:

```
class PizzaStore {
private:
        SimplePizzaFactory *factory;
public:
        PizzaStore(SimplePizzaFactory *factory) {
                this->factory = factory;
        }

        Pizza* orderPizza(string type) {
                Pizza *pizza;

                pizza = factory->createPizza(type);

                pizza->prepare();
                pizza->bake();
                pizza->cut();
                pizza->box();
                return pizza;
        }
        // other methods here
};
```

The Simple Factory defined

- Let's take a look at the class diagram of our new Pizza Store:

This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..

This is the product of the factory: pizza!

This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

These are our concrete products. Each product needs to implement the Pizza interface and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Franchising the pizza store

- Your Objectville PizzaStore has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.
- But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.

One franchise wants a factory that makes NY style pizzas: thin crust, tasty sauce and just a little cheese.

Another franchise wants a factory that makes Chicago style pizzas; their ccustomers like pizzas with thick crust, rich sauce, and tons of cheese.

Franchising the pizza store

```
NYPizzaFactory *nyFactory = new NYPizzaFactory();
PizzaStore *nyStore = new PizzaStore(nyFactory);
nyStore->order("Veggie");


ChicagoPizzaFactory *chicagoFactory = new ChicagoPizzaFactory();
PizzaStore *chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore->order("Veggie");
```

- But you'd like a little more quality control...
  - *So you test marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process: they'd bake things a little differently, they'd forget to cut the pizza and they'd use third-party boxes.*
  - *Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.*
  - *In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible. So, how can we have our pizza and eat it too?*

A framework for the pizza store

- There is a way to localize all the pizza making activities to the PizzaStore class, and yet give the franchises freedom to have their own regional style.
- What we're going to do is put the createPizza() method back into PizzaStore, but this time as an abstract method, and then create a PizzaStore subclass for each regional style.
  - *First, let's look at the changes to the PizzaStore:*

```
class PizzaStore {
public:
        Pizza* orderPizza(string type) {
                Pizza *pizza;
                pizza = createPizza(type);
                pizza->prepare();
                pizza->bake();
                pizza->cut();
                pizza->box();
                return pizza;
    }
protected:
    virtual Pizza* createPizza(string type)=0;
    // other methods here
};
```

PizzaStore is now abstract

All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

A framework for the pizza store

- Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza.

- Let's take a look at how this is going to work.

Allowing the subclasses to decide

- Remember, the PizzaStore already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.
- What varies among the regional PizzaStores is the style of pizzas they make – New York Pizza has thin crust, Chicago Pizza has thick , and so on – and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the createPizza() method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned orderPizza() method.

Allowing the subclasses to decide

```
Pizza* createPizza(type){
   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;
}
```

```
Pizza* createPizza(type){
   if(type=="cheese")
      pizza=new ChicagoStyleCheesePizza();
   else if(type=="pepperoni")
      pizza=new ChicagoStylePepperoniPizza();
   else if(type=="clam")
      pizza=new ChicagoStyleClamPizza();
   else if(type=="veggie")
      pizza=new ChicagoStyleVeggiePizza();
   return pizza;
}
```

Allowing the subclasses to decide

- Well, think about it from the point of view of the PizzaStore's orderPizza() method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.

- Now, to take this a little further, the orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved. In other words, it's decoupled!

Allowing the subclasses to decide

- When orderPizza() calls createPizza(), one of your subclasses will be called into action to create a pizza. Which kind of pizza will be made? Well, that's decided by the choice of pizza store you order from, NYStylePizzaStore or ChicagoStylePizzaStore.

- So, is there a real-time decision that subclasses make? No, but from the perspective of orderPizza(), if you chose a NYStylePizzaStore, that subclass gets to determine which pizza is made. So the subclasses aren't really "deciding" – it was you who decided by choosing which store you wanted – but they do determine which kind of pizza gets made.

Let's make a PizzaStore

- Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of Pizza. We'll take care of the big three pizza styles for the franchisees.
  - *Here's the New York regional style:*

The NYPizzaStore inherits from PizzaStore, so it inherits the orderPizza() method (among others).

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates

```
class NYPizzaStore:public PizzaStore {
    Pizza* createPizza(string item) {
        if (item =="cheese")  {
            return new NYStyleCheesePizza();
        } else if (item =="veggie") {
            return new NYStyleVeggiePizza();
        } else if (item =="clam") {
            return new NYStyleClamPizza();
        } else if (item =="pepperoni") {
            return new NYStylePepperoniPizza();
        } else return NULL;
    }
};
```

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

Declaring a factory method

- With just a couple of transformations to the PizzaStore we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility.
  - *Let's take a closer look:*

```
class PizzaStore {
public:
        Pizza* orderPizza(string type) {
                Pizza *pizza;
                pizza = createPizza(type);
                pizza->prepare();
                pizza->bake();
                pizza->cut();
                pizza->box();
                return pizza;
    }
protected:
    virtual Pizza* createPizza(string type)=0;
    // other methods here
};
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

All the responsibility for instantiating Pizzas has been moved into a method that acts as a factory.

Declaring a factory method

- A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a produc

```
virtual Product factoryMethod (String type) = 0
```

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

A factory method is abstract so the subclasses are counted on to handle object creation.

Let's see how it works: ordering pizzas with the pizza factory method

- Ethan needs to order his pizza from a NY pizza store.
- Joel needs to order his pizza from a Chicago pizza store. Same pizza ordering method, but different kind of pizza!
- So how do they order?
  - *(1) First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a NYPizzaStore.*
  - *(2) With a PizzaStore in hand, both Ethan and Joel call the orderPizza() method and pass in the type of pizza they want (cheese, veggie, and so on).*
  - *(3) To create the pizzas, the createPizza() method is called, which is defined in the two subclasses NYPizzaStore and ChicagoPizzaStore. As we defined them, the NYPizzaStore instantiates a NY style pizza, and the ChicagoPizzaStore instantiates Chicago style pizza. In either case, the Pizza is returned to the orderPizza() method.*
  - *(4) The orderPizza() method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.*

Let's check out how these pizzas are really made to order...

- (1) Let's follow Ethan's order: first we need a NY PizzaStore:

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

Creates a instance of NYPizzaStore.

- (2) Now that we have a store, we can take an order:

```
nyPizzaStore->orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs).

- (3) The orderPizza() method then calls the createPizza() method:

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

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza.

- (4) Finally we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

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

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore.

Let's implement some pizza

```
class Pizza {
protected:
        string name;
    string dough;
    string sauce;
    vector<string> toppings;
public:
    virtual void prepare() {
        cout<<"Preparing " + name<<endl;
        cout<<"Tossing dough..."<<endl;
        cout<<"Adding sauce..."<<endl;
        cout<<"Adding toppings: "<<endl;
                for (int i=0; i!=toppings.size(); i++) {
            cout<<"   " + toppings[i]<<endl;
        }
    }
    virtual void bake(){cout<<"Bake for 25 minutes at 350"<<endl;}
    virtual void cut() {cout<<"Cutting the pizza into diagonal slices"<<endl;}
    virtual void box() {cout<<"Place pizza in official PizzaStore box"<<endl; }
    string getName() { return name;}
};
```

We'll start with an abstract Pizza class and all the concrete pizzas will derive from this.

Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings.

Let's implement some pizza

```
class NYStyleCheesePizza:public Pizza {
public:
   NYStyleCheesePizza() {
         name = "NY Style Sauce and Cheese Pizza";
         dough = "Thin Crust Dough";
         sauce = "Marinara Sauce";
         toppings.push_back("Grated");
         toppings.push_back("Reggiano");
         toppings.push_back("Cheese");
   }
};
class ChicagoStyleCheesePizza:public Pizza {
public:
    ChicagoStyleCheesePizza() {
         name = "Chicago Style Deep Dish Cheese Pizza";
         dough = "Extra Thick Crust Dough";
         sauce = "Plum Tomato Sauce";
         toppings.push_back("Shredded");
         toppings.push_back("Mozzarella");
         toppings.push_back("Cheese");
    }
     void cut() {
        cout<<"Cutting the pizza into square slices"<<endl;
    }
};
```

Let's implement some pizza

```
int main() {
    PizzaStore *nyStore = new NYPizzaStore();
    PizzaStore *chicagoStore = new ChicagoPizzaStore();

    Pizza *pizza = nyStore->orderPizza("cheese");
    cout<<"Ethan ordered a " + pizza->getName() + "\n";

    pizza = chicagoStore->orderPizza("cheese");
    cout<<"Joel ordered a " + pizza->getName() + "\n";
}
```

It's finally time to meet the Factory Method Pattern

- All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes

The Product classes

Another perspective: parallel class hierarchies

- We've seen that the factory method provides a framework by supplying an orderPizza() method that is combined with a factory method. Another way to look at this pattern as a framework is in the way it encapsulates product knowledge into each creator.
  - *Let's look at the two parallel class hierarchies and see how they relate:*

Factory Method Pattern defined

- **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.
  - *As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.*
  - *As in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.*

Factory Method Pattern defined

- **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 Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.

The abstract factoryMethod() is what all Creator subclasses must implement.

The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.

All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.

The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

back at the PizzaStore... (Abstract Factory Pattern)

- The design for the PizzaStore is really shaping up: it's got a flexible framework and it does a good job of adhering to design principles.
- Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your procedures, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand!

Building the ingredient factories

- Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family.

```
class PizzaIngredientFactory {
public:
    virtual Dough* createDough() const=0;
    virtual Sauce* createSauce() const=0;
    virtual Cheese* createCheese() const=0;
    virtual std::vector<Veggies*> createVeggies() const =0;
    virtual Pepperoni* createPepperoni() const=0;
    virtual Clams* createClam() const=0;
};
```

- **Here's what we're going to do:**
  - *(1) Build a factory for each region. To do this, you'll create a subclass of PizzaIngredientFactory that implements each create method*
  - *(2) Implement a set of ingredient classes to be used with the factory, like ReggianoCheese, RedPeppers, and ThickCrustDough. These classes can be shared among regions where appropriate.*
  - *(3) Then we still need to hook all this up by working our new ingredient factories into our old PizzaStore code.*

Building the New York ingredient factory

- Okay, here's the implementation for the New York ingredient factory. This factory specializes in Marinara sauce, Reggiano Cheese, Fresh Clams...

```
class NYPizzaIngredientFactory:public PizzaIngredientFactory {
public:
    Dough* createDough() const{ return new ThinCrustDough(); }
    Sauce* createSauce() const{ return new MarinaraSauce();  }
    Cheese* createCheese() const{ return new ReggianoCheese(); }
    std::vector<Veggies*> createVeggies() const{
                std::vector< Veggies* > veggies;
                veggies.push_back( new Garlic() );
                veggies.push_back( new Onion() );
                veggies.push_back( new Mushroom() );
                veggies.push_back( new RedPepper() );
                return veggies;
    }
    Pepperoni* createPepperoni() const{ return new SlicedPepperoni();}
    Clams* createClam() const{ return new FreshClams(); }
};
```

Reworking the pizzas...

- We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract Pizza class:

```
class Pizza {
protected:
    string name;
    mutable Dough *dough;
    mutable Sauce *sauce;
    mutable std::vector<Veggies*> veggies;
    mutable Cheese *cheese;
    mutable Pepperoni *pepperoni;
    mutable Clams *clam;
public:
    virtual void prepare()const=0;
    virtual void bake() { cout<<"Bake for 25 minutes at 350"<<endl; }
    virtual void cut() { cout<<"Cutting the pizza into diagonal slices"<<endl; }
    virtual void box() { cout<<"Place pizza in official PizzaStore box"<<endl;  }
    virtual void setName(string name) { this->name = name; }
    string getName() { return name; }
    string toString() const{
        // code to print pizza here
    }
};
```

Reworking the pizzas, continued…

- Now that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas – only this time around they will get their ingredients straight from the factory.
- When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

Reworking the pizzas, continued…

- So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the Cheese Pizza:

```
class CheesePizza: public Pizza {
private:
    PizzaIngredientFactory *ingredientFactory;
public:
    CheesePizza(PizzaIngredientFactory *ingredientFactory) {
        this->ingredientFactory = ingredientFactory;
    }
    void prepare() const{
        cout<<"reparing" + name<<endl;
        dough = ingredientFactory->createDough();
        sauce = ingredientFactory->createSauce();
        cheese = ingredientFactory->createCheese();
    }
};
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Reworking the pizzas, continued…

- Let's check out the ClamPizza as well:

```
class ClamPizza: public Pizza {
private:
    PizzaIngredientFactory *ingredientFactory;
public:
    ClamPizza(PizzaIngredientFactory *ingredientFactory) {
        this->ingredientFactory = ingredientFactory;
    }
    void prepare() const {
        cout<<"Preparing" + name<<endl;
        dough = ingredientFactory->createDough();
        sauce = ingredientFactory->createSauce();
        cheese = ingredientFactory->createCheese();
        clam = ingredientFactory->createClam();
    }
};
```

Revisiting our pizza stores

- We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```
class NYPizzaStore:public PizzaStore {
protected:
    Pizza* createPizza(string item) const{
        Pizza *pizza = NULL;
        PizzaIngredientFactory *ingredientFactory =
            new NYPizzaIngredientFactory();
        if (item =="cheese") {
            pizza = new CheesePizza(ingredientFactory);
            pizza->setName("New York Style Cheese Pizza");
        } else if (item =="veggie") {
                        pizza = new VeggiePizza(ingredientFactory);
            pizza->setName("New York Style Veggie Pizza");
        } else if (item =="clam") {
                        pizza = new ClamPizza(ingredientFactory);
            pizza->setName("New York Style Clam Pizza");
        } else if (item =="pepperoni") {
                        pizza = new PepperoniPizza(ingredientFactory);
            pizza->setName("New York Style Pepperoni Pizza");
        }
                return pizza;
    }
};
```

What have we done?

- An Abstract Factory provides an interface for a family of products. What's a family? In our case it's all the things we need to make a pizza: dough, sauce, cheese, meats and veggies.

We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But, our client code stays the same.

More pizza for Ethan and Joel...

- Ethan and Joel can't get enough Objectville Pizza! What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...
- The first part of the order process hasn't changed at all. Let's follow Ethan's order again:
  - *(1) First we need a NY PizzaStore:*
    - `PizzaStore *nyPizzaStore = new NYPizzaStore();`
  - *(2) Now that we have a store, we can take an order:*
    - `nyPizzaStore->orderPizza("cheese");`
  - *(3) The orderPizza() method frst calls the createPizza() method:*
    - `Pizza *pizza  = createPizza("cheese");`

More pizza for Ethan and Joel...

- *(4) When the createPizza() method is called, that's when our ingredient factory gets involved:*
  - `Pizza *pizza = new CheesePizza(nyIngredientFactory);`

- *(5) Now that we have a store, we can take an order:*
  -
```
void prepare(){
  dough = factory->createDough();
  sauce = factory->createSauce();
  cheese = factory->createCheese();
  }
```

- *(6) Finally, we have the prepared pizza in hand and the orderPizza() method bakes, cuts, and boxes the pizza.*

Abstract Factory Pattern defined

- We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:
  - ***The Abstract Factory Pattern*** *provides an interface for creating families of related or dependent objects without specifying their concrete classes.*
- We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:

Abstract Factory Pattern defined

Abstract Factory Pattern defined

The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.

The abstract PizzaIngredientFactory is the interface that defnes how to make a family of related products - everything we need to make a pizza.

The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.

Each factory produces a different implementation for the family of products.

Factory Method and Abstract Factory compared

Factory Method and Abstract Factory compared

*Provides an abstract interface for creating a **family of products.***

*Methods to create products in an Abstract Factory are often implemented with **a Factory Method**...*

PizzaIngredientFactory is implemented as an Abstract Factory because we need to create families of products (the ingredients). Each subclass implements the ingredients using its own regional suppliers.

*Each concrete subclass creates **a family of products**.*

The subclass decides the type of dough …

… or the type of clams.

Each ingredient represents a product that is produced by a Factıry Method in te Abstract Factory

The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Singleton Pattern

Instructor: Dr. Metin Özkan

Singleton Pattern: Definition

- The Singleton pattern ensures a class has ony one instance (or a constrained set of instances), and provides a global point of access to it.
  - *Useful for objects that represents real-world resources, such as printer, in which you want to instantiate one and only one object to represent each resource*
  - *Also useful for "management" code, such as thread/connection pool.*

The classic singleton Pattern implementation

```
class Singleton{
private:
        static Singleton *uniqueInstance;

        //other useful instance variables here

        Singleton() {}
public:
        static Singleton* getInstance(){
                if(uniqueInstance == NULL){
                        uniqueInstance = new Singleton();
                }
                return uniqueInstance;
        }

        // other useful methods here
};
```

We have a static variable to hold our one instance of the class Singleton.

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of

Singleton is a normal class; it has other useful instance variables and methods.

The Chocolate Factory

- Everyone knows that all modern chocolate factories have computer controlled chocolate boilers. The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.
- Here's the controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler. Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 500 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler!

The Chocolate Factory

```
class ChocolateBoiler {
private:
   bool empty;
   bool boiled;
public:
   ChocolateBoiler() {
      empty = true;
      boiled = false;
   }
   void fill() {
      if (isEmpty()) {
            empty = false; boiled = false;
            // fill the boiler with a milk/chocolate mixture
      }
   }
   void drain() {
      if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
      }
   }
   void boil() {
      if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
      }
   }
   bool isEmpty() { return empty; }
   bool isBoiled() { return boiled; }
};
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled fags.

To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it 's boiled we set the boiled flag to true.

Improve the ChocolateBoiler class by turning into a singleton.

```
class ChocolateBoiler {
private:
   bool empty;
   bool boiled;
   static ChocolateBoiler *uniqueInstance;
public:
   ChocolateBoiler() {
      empty = true; boiled = false;
   }
public:
   static ChocolateBoiler* getInstance(){
      if(uniqueInstance == NULL){
            uniqueInstance = new ChocolateBoiler();
      }
      return uniqueInstance;
   }

   void fill() {
      if (isEmpty()) {
            empty = false; boiled = false;
            // fill the boiler with a milk/chocolate mixture
      }
   }

   // rest of the code ...
};
```

Singleton Pattern defined

- **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.
  - *What's really going on here? We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.*
  - *We are also providing a global access point to the instance: whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the singleton is created in a lazy manner, which is especially important for resource intensive objects.*

Singleton Pattern defined

- Check out the class diagram:

The uniqueInstance class variable holds our one and only instance of Singleton

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Implement a chocolate factory

```
ChocolateBoiler* ChocolateBoiler::uniqueInstance = 0;

int main() {

        ChocolateBoiler* boiler = ChocolateBoiler::getInstance();
        boiler->fill();
        boiler->boil();
        boiler->drain();

        // will return the existing instance
        ChocolateBoiler* boiler2 = ChocolateBoiler::getInstance();

        if( boiler == boiler2 )
            std::cout << "Got same boiler" << std::endl;
        else
            std::cout << "Oh oh! got a different boiler" << std::endl;

        return 0;
}
```

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Command Pattern

Instructor: Dr. Metin Özkan

Sample Application: the Remote Control...

There are "on" and "off" buttons for each of the seven slots.

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.

These two buttons are used to control the household device stored in slot one...

... and these two control the household device stored in slot two...

write your device names here .

Here's the global "undo" button that undoes the last button pressed.

Taking a look at the vendor classes

- Check out the vendor classes on the CD-R. These should give you some idea of the interfaces of the objects we need to control from the remote.

- It looks like we have quite a set of classes here, and not a lot of industry effort to come up with a set of common interfaces. Not only that, it sounds like we can expect more of these classes in the future. Designing a remote control API is going to be interesting.

A brief introduction to the Command Pattern

- Let's take a short detour to the diner and study the interactions between the customers, the waitress, the orders and the short-order cook. Through these interactions, the objects involved in the Command Pattern are going to be understand and also how the decoupling works. After that, we're going to knock out that remote control API.

the interaction in a little more detail...

The Objectville Diner roles and responsibilities

- An Order Slip encapsulates a request to prepare a meal.
  - *Think of the Order Slip as an object, an object that acts as a request to prepare a meal.*

- The Waitress's job is to take Order Slips and invoke the orderUp() method on them.
  - *The Waitress has it easy: take an order from the customer, continue helping customers until she makes it back to the order counter, then invoke the orderUp() method to have the meal prepared.*

- The Short Order Cook has the knowledge required to prepare the meal.
  - *The Short Order Cook is the object that really knows how to prepare meals.*

The Objectville Diner roles and responsibilities

- Think of the Diner as a model for an OO design pattern that allows us to separate an object making a request from the objects that receive and execute those requests.
  - *For instance, in our remote control API, we need to separate the code that gets invoked when we press a button from the objects of the vendor-specific classes that carry out those requests. What if each slot of the remote held an object like the Diner's order slip object? Then, when a button is pressed, we could just call the equivalent of the "orderUp()" method on this object and have the lights turn on without the remote knowing the details of how to make those things happen or what objects are making them happen.*

From the Diner to the Command Pattern

From the Diner to the Command Pattern

1. The client creates a command object.

2. The client does a setCommand() to store the command object in the invoker.

3. The client ask the invoker to execute the command.

The first command object: the Remote Control

- **Implementing the Command interface**
  - *All command objects implement the same interface, which consists of one method. In the Diner we called this method orderUp(); however, we typically just use the name execute().*
  - *Here's the Command interface:*

```
class Command{
public:
        virtual void execute()=0;
};
```

Simple. All we need is one method called execute().

The first command object: the Remote Control

- **Implementing a Command to turn a light on**

- *Let's implement a command for turning a light on. Referring to the set of vendor classes, the light class has two methods: on() and off().*
- *Here's the Light class implemented as a Command:*

```
class LightOnCommand: public Command {
        Light *light;
public:
        LightOnCommand(Light *light){
                this->light = light;
        }

        void execute(){
                light->on();
        }
};
```

This is a command, so we need to implement the Command interface.

The constructor is passed the specific light that this command is going to control – sat the living room light – and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

Using the command object

- Let's make things simple: say we've got a remote control with only one button and corresponding slot to hold a device to control:

```
class SimpleRemoteControl {
        Command *slot;
public:
        SimpleRemoteControl(){}

        void setCommand(Command *command){
                slot = command;
        }

        void buttonWasPressed(){
                slot ->execute();
        }
};
```

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control(1)

- Test out the simple remote control.

This is the client in Command Pattern.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

```
int main() {
        SimpleRemoteControl *remote = new SimpleRemoteControl;
        Light *light = new Light();
        LightOnCommand *lightOn = new LightOnCommand(light);

        remote -> setCommand(lightOn);
        remote -> buttonWasPressed();

        return 0;
}
```

Create a light object, this will be the Receiver of the request

Create a command and pass the Receiver to it.

Simulate the button being pressed.

Pass the command to the Invoker

The Command Pattern defined

- Similar to the Objectville Diner, the remote control API has been partly implemented, and in the process we've seen how the classes and objects interact in the Command Pattern. Now we're going to define the Command Pattern and nail down all the details.
- Let's start with its official definition:
  - ***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 Command Pattern defined

- Let's step through this. We know that **a command object encapsulates a request by binding together a set of actions on a specific receiver**.
  - *To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute(). When called, execute() causes the actions to be invoked on the receiver. From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced.*
- We've also seen a couple examples of **parameterizing an object with a command**.
  - *The Waitress was parameterized with multiple orders throughout the day. In the simple remote control, we loaded the button slot with a "light on" command. Like the Waitress, your remote slot didn't care what command object it had, as long as it implemented the Command interface.*

The Command Pattern defined

- What we haven't encountered yet is using commands to **implement queues and logs and support undo operations**. Those are pretty straightforward extensions of the basic Command Pattern and we will get to them soon. We can also easily support what's known as the **Meta Command Pattern** once we have the basics in place. The Meta Command Pattern allows you to create macros of commands so that you can execute multiple commands at once.

The Command Pattern defined (the class diagram)

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. A command is invoked through its execute() method, which asks a receiver to perform an action. This has an undo() method, which will be covered later .

The execute method invokes the action(s) on the receiver needed to fulfill the request.

The receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

```
public void execute(){
        receiver.action()
}
```

The ConcreteCommand defines a binding between an action and a receiver. The invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the receiver.

Assigning Commands to slots

- a plan: We're going to assign each slot to a command in the remote control. This makes the remote control our invoker. When a button is pressed the execute() method is going to be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans).

Implementing the Remote Control

```
class RemoteControl {
private:
        static const int SLOTS = 7;
private:
        Command* onCommands[SLOTS];
        Command* offCommands[SLOTS];
        Command* noCommand;
public:
        RemoteControl() {
                noCommand = new NoCommand();
                for( int i = 0; i < SLOTS; i++ ) {
                        onCommands[i]  = noCommand;
                        offCommands[i] = noCommand;
                }
        }
        ~RemoteControl() { delete noCommand; }
        void setCommand(int slot,Command* onCommand,Command* offCommand){
                onCommands[slot] = onCommand;
                offCommands[slot] = offCommand;
        }
```

This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the on and off arrays for later use.

Implementing the Remote Control

```
        void onButtonWasPushed( int slot ) const {
                onCommands[slot]->execute();
        }
        void offButtonWasPushed( int slot ) const {
                offCommands[slot]->execute();
        }
        string toString() const {
                stringstream value;
                value << "\n------ Remote Control -------\n" << endl;
                for( int i = 0; i < SLOTS; i++ ) {
                        value << "[slot " << i << "] ";
                        value << typeid( *onCommands[i] ).name();
                        value << "    ";
                        value << typeid( *offCommands[i] ).name();
                        value << endl;
                }
                return value.str();
        }
};
```

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed().

We've overwritten toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

Implementing the Commands

- Off commands are no different than LightOnCommand; in fact the LightOffCommand looks like this:

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

The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Implementing the Commands

- Let's try something a little more challenging; how about writing on and off commands for the Stereo? we just bind the Stereo to the off() method in the StereoOffCommand. On is a little more complicated; write a StereoOnWithCDCommand…

```
class StereoOnWithCDCommand : public Command {
private:
        const Stereo* stereo;
public:
        StereoOnWithCDCommand( const Stereo* stereo ):stereo(stereo) { }
        void execute() const {
                stereo->on();
                stereo->setCD();
                stereo->setVolume(11);
        }
};
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable

To carry out this request, we need to call three methods on the stereo.

Application : the Remote Control

```
int main() {
        RemoteControl *remoteControl      = new RemoteControl();
```

Create all the devices in their proper locations.

```
        Light *livingRoomLight    = new Light( "Living Room" );
        Light *kitchenLight       = new Light( "Kitchen" );
        CeilingFan *ceilingFan    = new CeilingFan( "Living Room" );
        GarageDoor *garageDoor    = new GarageDoor( "Garage" );
        Stereo *stereo            = new Stereo( "Living Room" );
```

Create all the Light Command objects.

```
        LightOnCommand *livingRoomLightOn = new LightOnCommand( livingRoomLight );
        LightOffCommand *livingRoomLightOff= new LightOffCommand( livingRoomLight );
        LightOnCommand *kitchenLightOn    = new LightOnCommand( kitchenLight );
        LightOffCommand *kitchenLightOff  = new LightOffCommand( kitchenLight );
```

Create on and off for the ceiling fan.

```
        CeilingFanOnCommand *ceilingFanOn = new CeilingFanOnCommand( ceilingFan );
        CeilingFanOffCommand *ceilingFanOff = new CeilingFanOffCommand( ceilingFan);
```

Create up and down commands for the garage.

```
        GarageDoorUpCommand *garageDoorUp = new GarageDoorUpCommand( garageDoor );
        GarageDoorDownCommand *garageDoorDown=new GarageDoorDownCommand(garageDoor);
```

Create the stereo on and off commands

```
        StereoOnWithCDCommand *stereoOnWithCD = new StereoOnWithCDCommand( stereo );
        StereoOffCommand  *stereoOff = new StereoOffCommand( stereo );
```

Application : the Remote Control

```
        remoteControl->setCommand( 0, livingRoomLightOn, livingRoomLightOff );
        remoteControl->setCommand( 1, kitchenLightOn, kitchenLightOff );
        remoteControl->setCommand( 2, ceilingFanOn, ceilingFanOff );
        remoteControl->setCommand( 3, stereoOnWithCD, stereoOff );
        remoteControl->setCommand( 4, garageDoorUp, garageDoorDown );

        cout << remoteControl->toString() << endl;

        remoteControl->onButtonWasPushed( 0 );
        remoteControl->offButtonWasPushed( 0 );
        remoteControl->onButtonWasPushed( 1 );
        remoteControl->offButtonWasPushed( 1 );
        remoteControl->onButtonWasPushed( 2 );
        remoteControl->offButtonWasPushed( 2 );
        remoteControl->onButtonWasPushed( 3 );
        remoteControl->offButtonWasPushed( 3 );
        remoteControl->onButtonWasPushed( 4 );
        remoteControl->offButtonWasPushed( 4 );

        return 0;
}
```

Now that we've got all our commands, we can load them into the remote slots.

Here's where we use our toString() method to print each remote slot and the command that it is assigned to.

All right, we are ready to roll! Now, we step through each slot and push its On and Off button.

Application : the Remote Control

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\Debug\OOPI...

```
------ Remote Control -------

[slot 0] class LightOnCommand     class LightOffCommand
[slot 1] class LightOnCommand     class LightOffCommand
[slot 2] class CeilingFanOnCommand     class CeilingFanOffCommand
[slot 3] class StereoOnWithCDCommand     class StereoOffCommand
[slot 4] class GarageDoorUpCommand     class GarageDoorDownCommand
[slot 5] class NoCommand     class NoCommand
[slot 6] class NoCommand     class NoCommand

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off
Garage Door is Up
Garage Door is Down
```

Our commands are in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living room light is on."

Time to write that documentation...

- Remote Control API Design for Home Automation or Bust, Inc.,

The RemoteLoader creates a number of Command Objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.

All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.

The RemoteControl manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home-automation work.

The vendor classes are used to perform the actual home-automation work of controlling devices. Here, we are using the Light class as an example.

Using the Command interface, each action that can be invoked by pressing a button on the remote is implemented with a simple Command object. The command object holds a reference to an object that is an instance of a Vendor class and implements an execute method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

Undo

- We need to add functionality to support the undo button on the remote.
- It works like this:
  - *Say the Living Room Light is off and you press the on button on the remote. The light turns on. No if you press the undo button then the last action will be reversed – in this case the light will turn on.*
  - *(1) an undo() method mirrors the execute() method.*

```
class Command{
public:
        void execute();
        void undo();
};
```

Undo

- *(2) Let's start with the LightOnCommand: if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() method.*

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

execute() turns the light on, so undo() simply turns the light back off.

Undo

- *Now for the LightOffCommand: undo() method just needs to call the Light's on() method.*

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

Undo

- *(3) To add support for the undo button we only have to make a few small changes to the Remote Control class. Here's how we're going to do it: we'll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.*

```
class RemoteControlWithUndo {
private:
        static const int SLOTS = 7;
private:
        Command* onCommands[SLOTS];
        Command* offCommands[SLOTS];
        Command* noCommand;
        mutable Command* undoCommand;
public:
        RemoteControlWithUndo() {
                noCommand = new NoCommand();
                for( int i = 0; i < SLOTS; i++ ) {
                        onCommands[i]  = noCommand;
                        offCommands[i] = noCommand;
                }
                undoCommand = noCommand;
        }
```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

Undo

```
~RemoteControlWithUndo() {
        delete noCommand;
}
void setCommand( int slot, Command* onCommand, Command* offCommand )
{
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
}
void onButtonWasPushed( int slot ) const
{
        onCommands[slot]->execute();
        undoCommand = onCommands[slot];
}
void offButtonWasPushed( int slot ) const
{
        offCommands[slot]->execute();
        undoCommand = offCommands[slot];
}
void undoButtonWasPushed(){
        undoCommand->undo();
}
```

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands..

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

Undo

```
        string toString() const {
                stringstream value;
                value << "\n------ Remote Control -------\n" << endl;
                for( int i = 0; i < SLOTS; i++ ) {
                        value << "[slot " << i << "] ";
                        value << typeid( *onCommands[i] ).name();
                        value << "    ";
                        value << typeid( *offCommands[i] ).name();
                        value << endl;
                }
                value << "[undo] " << typeid( *undoCommand ).name() << endl;

                return value.str();
        }
};
```

Test the application with undo method

```
int main() {
        RemoteControlWithUndo *remoteControl = new RemoteControlWithUndo();

        Light *livingRoomLight =  new Light( "Living Room" );

        LightOnCommand *livingRoomLightOn =new LightOnCommand(livingRoomLight);
        LightOffCommand *livingRoomLightOff=new LightOffCommand(livingRoomLight);

        remoteControl->setCommand( 0, livingRoomLightOn, livingRoomLightOff );

        remoteControl->onButtonWasPushed( 0 );
        remoteControl->offButtonWasPushed( 0 );
        cout << remoteControl->toString() << endl;
        remoteControl->undoButtonWasPushed(  );
        remoteControl->offButtonWasPushed( 0 );
        remoteControl->onButtonWasPushed( 0 );
        cout << remoteControl->toString() << endl;
        remoteControl->undoButtonWasPushed(  );

        return 0;
}
```

Test the application with undo method

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_...

```
Living Room light is on
Living Room light is off

------ Remote Control -------

[slot 0] class LightOnCommand    class LightOffCommand
[slot 1] class NoCommand    class NoCommand
[slot 2] class NoCommand    class NoCommand
[slot 3] class NoCommand    class NoCommand
[slot 4] class NoCommand    class NoCommand
[slot 5] class NoCommand    class NoCommand
[slot 6] class NoCommand    class NoCommand
[undo] class LightOffCommand

Living Room light is on
Living Room light is off
Living Room light is on

------ Remote Control -------

[slot 0] class LightOnCommand    class LightOffCommand
[slot 1] class NoCommand    class NoCommand
[slot 2] class NoCommand    class NoCommand
[slot 3] class NoCommand    class NoCommand
[slot 4] class NoCommand    class NoCommand
[slot 5] class NoCommand    class NoCommand
[slot 6] class NoCommand    class NoCommand
[undo] class LightOnCommand

Living Room light is off
```

Turn the light on, then off

Undo holds the last command invoked.

Undo was pressed…

Using state to implement Undo

- We need to manage a bit of state to implement undo. Let's try something a little more interesting, like the CeilingFan from the vendor classes. The ceiling fan allows a number of speeds to be set along with an off method.
- Here's the source code for the CeilingFan:

```
class CeilingFan {
private:
        mutable int level;
        string location;
public:
        static const int HIGH = 3;
        static const int MEDIUM = 2;
        static const int LOW = 1;
        static const int OFF = 0;

        CeilingFan(const string location ) : level( LOW ),location(location ) { }
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

Using state to implement Undo

```
public:
        void high() const {
                level = HIGH;
                cout << location << " ceiling fan is on high" << endl;
        }
        void medium() const {
                level = MEDIUM;
                cout << location << " ceiling fan is on medium" << endl;
        }
        void low() const {
                level = LOW;
                cout << location << " ceiling fan is on low" << endl;
        }
        void off() const {
                level = OFF;
                cout << location << " ceiling fan is off" << endl;
        }
        int getSpeed() const {
                return level;
        }
};
```

Adding Undo to the ceiling fan commands

- Now let's tackle adding undo to the various CeilingFan commands. To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting. Here's the code for the CeilingFanHighCommand:

```
class CeilingFanHighCommand : public Command {
private:
        const CeilingFan* ceilingFan;
        mutable int prevSpeed;
public:
      CeilingFanHighCommand(const CeilingFan* ceilingFan ):ceilingFan(ceilingFan )
         { prevSpeed = ceilingFan->getSpeed(); }
         void execute() const {
                 prevSpeed = ceilingFan->getSpeed();
                 ceilingFan->high();
         }
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

Adding Undo to the ceiling fan commands

```
void undo() const {
        if( prevSpeed == CeilingFan::HIGH ) {
                ceilingFan->high();
        } else if(prevSpeed == CeilingFan::MEDIUM ) {
                ceilingFan->medium();
        } else if( prevSpeed == CeilingFan::LOW ) {
                ceilingFan->low();
        } else if( prevSpeed == CeilingFan::OFF ) {
                ceilingFan->off();
        }
}
};
```

To undo we set the speed of the fan back to its previous speed.

test the ceiling fan

- Time to load up our remote control with the ceiling fan commands. We're going to load slot zero's on button with the medium setting for the fan and slot one with the high setting. Both corresponding off buttons will hold the ceiling fan off command.
- Here's our test program:

test the ceiling fan

```
int main() {
        RemoteControlWithUndo *remoteControl = new RemoteControlWithUndo();

        CeilingFan *ceilingFan = new CeilingFan("Living Room”);

      CeilingFanMediumCommand *ceilingFanMedium =new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand *ceilingFanHigh = new CeilingFanHighCommand( ceilingFan);
        CeilingFanOffCommand *ceilingFanOff = new CeilingFanOffCommand( ceilingFan);

        remoteControl->setCommand( 0, ceilingFanMedium, ceilingFanOff );
        remoteControl->setCommand( 1, ceilingFanHigh, ceilingFanOff);

        remoteControl->onButtonWasPushed( 0 );
        remoteControl->offButtonWasPushed( 0 );
        cout << remoteControl->toString() << endl;
        remoteControl->undoButtonWasPushed();

        remoteControl->onButtonWasPushed( 1 );
        cout << remoteControl->toString() << endl;
        remoteControl->undoButtonWasPushed();

        return 0;
}
```

test the ceiling fan

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\Debug\OOPII_Ap...

```
Living Room ceiling fan is on medium
Living Room ceiling fan is off

------ Remote Control -------

[slot 0] class CeilingFanMediumCommand    class CeilingFanOffCommand
[slot 1] class CeilingFanHighCommand    class CeilingFanOffCommand
[slot 2] class NoCommand    class NoCommand
[slot 3] class NoCommand    class NoCommand
[slot 4] class NoCommand    class NoCommand
[slot 5] class NoCommand    class NoCommand
[slot 6] class NoCommand    class NoCommand
[undo] class CeilingFanOffCommand

Living Room ceiling fan is on medium
Living Room ceiling fan is on high

------ Remote Control -------

[slot 0] class CeilingFanMediumCommand    class CeilingFanOffCommand
[slot 1] class CeilingFanHighCommand    class CeilingFanOffCommand
[slot 2] class NoCommand    class NoCommand
[slot 3] class NoCommand    class NoCommand
[slot 4] class NoCommand    class NoCommand
[slot 5] class NoCommand    class NoCommand
[slot 6] class NoCommand    class NoCommand
[undo] class CeilingFanHighCommand

Living Room ceiling fan is on medium
```

Macro Command

- What's the point of having a remote if you can't push one button and have the lights dimmed, the stereo and TV turned on and set to a DVD and the hot tub fired up?

A new kind of Command that can execute more than one command.

Take an array of Commands and store them in the MacroCommand.

```
class MacroCommand: public Command{
private:
        std::vector<Command*> commands;
public:
        MacroCommand(std::vector<Command*> command):commands(command){ }
        void execute()const{
                for(int i=0;i<commands.size();i++){
                        commands[i]->execute();
                }
        }
};
```

Execute the commands one at a time.

Using a macro command

- Let's step through how we use a macro command:

  - *(1) First we create the set of commands we want to go into the macro:*

```
RemoteControl *remoteControl = new RemoteControl();

Light *light = new Light( "Living Room" );
Stereo *stereo = new Stereo("Living Room");
Hottub *hottub = new Hottub();

LightOnCommand *lightOn = new LightOnCommand(light);
StereoOnCommand *stereoOn = new StereoOnCommand(stereo);
HottubOnCommand *hottubOn = new HottubOnCommand(hottub);

LightOffCommand *lightOff = new LightOffCommand(light);
StereoOffCommand *stereoOff = new StereoOffCommand(stereo);
HottubOffCommand *hottubOff = new HottubOffCommand(hottub);
```

Create all the devices, a light, stereo, and hot tub.

Now create all the On commands to control them.

Using a macro command

- *(2) Next we create two arrays, one for the On commands and one for Off commands, and load them with the corresponding commands:*

```
std::vector<Command*> partyOn;
partyOn.push_back(lightOn);
partyOn.push_back(stereoOn);
partyOn.push_back(hottubOn);
std::vector<Command*> partyOff;
partyOff.push_back(lightOff);
partyOff.push_back(stereoOff);
partyOff.push_back(hottubOff);

MacroCommand *partyOnMacro = new MacroCommand(partyOn);
MacroCommand *partyOffMacro = new MacroCommand(partyOff);
```

Create an array for On and an array for Off commands …

and create two corresponding macros to hold themcommands to control them.

Using a macro command

- *(3) We assign MacroCommand to a button like we always do:*

```
remoteControl->setCommand(0,partyOnMacro,partyOffMacro);
```

Assign the macro command to a button as we would ant command.

- *(4) Finally, we just need to push some buttons and see if this works.*

```
cout << remoteControl->toString() << endl;
cout<<"---- Pushing Macro on ----"<<endl;
remoteControl->onButtonWasPushed(0);
cout<<"---- Pushing Macro off ----"<<endl;
remoteControl->offButtonWasPushed(0);
```

Using a macro command

c:\Users\metis\Documents\Visual Studio 2008\Projects\O...

```
------ Remote Control -------

[slot 0] class MacroCommand    class MacroCommand
[slot 1] class NoCommand    class NoCommand
[slot 2] class NoCommand    class NoCommand
[slot 3] class NoCommand    class NoCommand
[slot 4] class NoCommand    class NoCommand
[slot 5] class NoCommand    class NoCommand
[slot 6] class NoCommand    class NoCommand

---- Pushing Macro on ----
Living Room light is on
Living Room stereo is on
Hottub is heating to a steaming 105 degrees
Hottub is bubbling!
---- Pushing Macro off ----
Living Room light is off
Living Room stereo is off
Hottub is cooling to 98 degrees
```

More uses of the Command Pattern: queuing requests

- Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object.
- Now, the computation itself may be invoked long after some client application creates the command object. In fact, it may even be invoked by a different thread.
- We can take this scenario and apply it to many useful applications such as schedulers, thread pools and job queues, to name a few.
  - *Imagine a job queue: you add commands to the queue on one end, and on the other end sit a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.*

More uses of the Command Pattern: queuing requests

More uses of the Command Pattern: logging requests

- The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions. The Command Pattern can support these semantics with the addition of two methods: store() and load().
- How does this work? As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their execute() methods in batch and in order.

More uses of the Command Pattern: logging requests

- for example, a spreadsheet application: we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs.
- In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.

More uses of the Command Pattern: logging requests

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Adapter Patterns

Instructor: Dr. Metin Özkan

Adapters all around us

- The real world is full of adapters. For example:
  - *Have you ever needed to use a US-made laptop in a European country? Then you've probably needed an AC power adapter...*

European Wall Outlet

The European wall outlet exposes one interface for getting power.

AC Power Adapter

The adapter converts one interface into another

Standard AC Plug

The US laptop expects another interface.

Object oriented adapters

- Assume that you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:

Object oriented adapters

- You don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.

Object oriented adapters

- The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.

An adapter

- It's time to see an adapter in action. Remember the ducks from previous lectures? Let's review a slightly simplifed version of the Duck interfaces and classes:

```
class Duck {
public:
        virtual ~Duck() = 0 { }
        virtual void fly() const = 0;
        virtual void quack() const = 0;
};
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

- Here's a subclass of Duck, the MallardDuck.

```
class MallardDuck : public Duck {
public:
        void fly() const {
                cout << "I'm flying" << endl;
        }
        void quack() const {
                cout << "Quack" << endl;
        }
};
```

Simple implementations: the duck just prints out what it is doing.

An adapter

- Now it's time to meet the newest fowl on the block:

```
class Chicken {
public:
        virtual ~Chicken() = 0 { }
        virtual void cluck() const = 0;
        virtual void fly() const = 0;
};
```

Chickens don't quack, they cluck.

Chickens can fly, although they can only fly short distances.

```
class WildChicken : public Chicken {
public:
        void fly() const {
                cout << "I'm flying a short distance" << endl;
        }
        void cluck() const {
                cout << "Cluck cluck" << endl;
        }
};
```

Here's a concrete implementation of Chicken; like Duck, it just prints out its actions.

An adapter

- Now, let's say you're short on Duck objects and you'd like to use some Chicken objects in their place. Obviously we can't use the chickens outright because they have a different interface.
- So, let's write an Adapter:

```
class ChickenAdapter : public Duck {
private:
        const Chicken* chicken;

public:
        ChickenAdapter( const Chicken* chicken )
        {
                this->chicken=chicken;
        }
        void fly() const {
                for( int i = 0; i < 5; i++ ) {
                        chicken->fly();
                }
        }
        void quack() const {
                chicken->cluck();
        }
};
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now, we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the cluck() method.

Test drive the adapter

- Now we just need some code to test drive our adapter:

```
void testDuck( const Duck* duck ) {
        duck->quack();
        duck->fly();
}

int main() {
        MallardDuck *duck = new MallardDuck();

        WildChicken *chicken = new WildChicken();
        Duck *chickenAdapter = new ChickenAdapter(chicken);

        cout << "The Chicken says..." << endl;
        chicken->cluck();
        chicken->fly();

        cout << endl << "The Duck says..." << endl;
        testDuck( duck );

        cout << endl << "The ChickenAdapter says..." << endl;
        testDuck( chickenAdapter );

        return 0;
}
```

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

wrap the chicken in a ChickenAdapter, which makes it look like a Duck.

Then, let's test the Chicken: make it cluck, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the chicken as a duck...

Test drive the adapter

```
c:\Users\metis\Documents\Visual Stud...

The Chicken says...
Cluck cluck
I'm flying a short distance

The Duck says...
Quack
I'm flying

The ChickenAdapter says...
Cluck cluck
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

And the adapter clucks when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a chicken disguised as a duck!

The Adapter Pattern explained

- Now that we have an idea of what an Adapter is, let's step back and look at all the pieces again.

The Adapter Pattern explained

- Here's how the Client uses the Adapter

1. *The client makes a request to the adapter by calling a method on it using the target interface.*

2. *The adapter translates the request into one or more class on the adaptee using the adaptee interface.*

3. *The client receives the results of the call and never knows there is an adapter doing the translation.*

Adapter Pattern defined

- Let's look at the official definition of the Adapter Pattern:

  - ***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.*

- Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion.
- This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

Adapter Pattern defined

- We've taken a look at the runtime behavior of the pattern; let's take a look at its class diagram as well:

Object and class adapters

- There are actually two kinds of adapters:
  - *object adapters (the class diagram on the previous slides)*
  - *class adapters (will be given)*
- You need multiple inheritance to implement class adapter, which isn't possible in Java.
- Let's look at the class diagram for multiple inheritance.

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Facade Patterns

Instructor: Dr. Metin Özkan

Home Sweet Home Theater

- You've done your research and you've assembled a system complete with a DVD player, a projection video system, an automated screen, surround sound and even a popcorn popper.

Watching a movie (the hard way)

- to watch the movie, you need to perform a few tasks:
  - *Turn on the popcorn popper*
  - *Start the popper popping*
  - *Dim the lights*
  - *Put the screen down*
  - *Turn the projector on*
  - *Put the projector on wide-screen mode*
  - *Turn the sound amplifer on*
  - *Set the amplifer to DVD input*
  - *Set the amplifer to surround sound*
  - *Set the amplifer volume to medium(5)*
  - *Turn the DVD Player on*
  - *Start the DVD Player playing*

Watching a movie (the hard way)

- Let's check out those same tasks in terms of the classes and the method calls needed to perform them:

```
popper.on();
popper.pop();

lights.dim(10);

screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode();

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

- When the movie is over, how do you turn everything off?
- If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure.

Facade!

- with the Facade Pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface. Don't worry; if you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface.
- Let's take a look at how the Facade operates:

Facade

1. Create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().
2. The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.
3. The client code calls methods on the home theater Facade, not on the subsystem. So to watch a movie we just call one method, watchMovie(), and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker.
4. The facade leaves the subsystem accessable to be used directly. If you need the advanced functionality of the subsystem classes, they are available for use.

Constructing your home theater facade

- Let's step through the construction of the HomeTheaterFacade:
- The first step is to use composition so that the facade has access to all the components of the subsystem:

```
class HomeTheaterFacade {
private:
        Amplifier* _amp;
        Tuner* _tuner;
        DvdPlayer* _dvd;
        CdPlayer* _cd;
        Projector* _projector;
        TheaterLights* _lights;
        Screen* _screen;
        PopcornPopper* _popper;
public:
        HomeTheaterFacade( Amplifier* amp, Tuner* tuner, DvdPlayer* dvd,
                CdPlayer* cd, Projector* projector, Screen* screen,
                TheaterLights* lights, PopcornPopper* popper ) :
                _amp( amp ), _tuner( tuner ), _dvd( dvd ),
                _cd( cd ), _projector( projector ),
                _lights( lights ), _screen( screen ), _popper( popper ) {    }

        //Other methods here
};
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade assigns each to the corresponding instance variable.

Implementing the simplified interface

- Let's implement the watchMovie() and endMovie() methods:

```
void watchMovie(string movie ) {
     cout << "Get ready to watch a movie..." << endl;
     _popper->on();
     _popper->pop();
     _lights->dim( 10 );
     _screen->down();
     _projector->on();
     _projector->wideScreenMode();
     _amp->on();
     _amp->setDvd( _dvd );
     _amp->setSurroundSound();
     _amp->setVolume( 5 );
     _dvd->on();
     _dvd->play( movie );
}
void endMovie() {
     cout << "Shutting movie theater down..." << endl;
     _popper->off();
     _lights->on();
     _screen->up();
     _projector->off();
     _amp->off();
     _dvd->stop();
     _dvd->eject();
     _dvd->off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

endMovie() takes care of shutting everything down. Again, each task is delegated to the appropriate component in the subsystem.

Time to watch a movie (the easy way)

```
int main( int argc, char* argv[] ) {
    Amplifier *amp = new Amplifier( "Top-O-Line Amplifier" );
    Tuner *tuner = new Tuner( "Top-O-Line AM/FM Tuner", amp);
    DvdPlayer *dvd = new DvdPlayer( "Top-O-Line DVD Player", amp);
    CdPlayer *cd = new CdPlayer( "Top-O-Line CD Player", amp);
    Projector *projector = new Projector( "Top-O-Line Projector", dvd);
    TheaterLights *lights = new TheaterLights( "Theater Ceiling Lights" );
    Screen *screen = new Screen( "Theater Screen" );
    PopcornPopper *popper = new PopcornPopper( "Popcorn Popper" );
    HomeTheaterFacade *homeTheater = new HomeTheaterFacade( amp,
                                                            tuner,
                                                            dvd,
                                                            cd,
                                                            projector,
                                                            screen,
                                                            lights,
                                                            popper );

        homeTheater->watchMovie( "Raiders of the Lost Ark" );
        homeTheater->endMovie();

        return 0;
}
```

Time to watch a movie (the easy way)

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\Debug\OO...

```
Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
```

Facade Pattern defined

- To use the Facade Pattern, we create a class that simplifies and unifies a set of more complex classes that belong to some subsystem.
- Unlike a lot of patterns, Facade is fairly straightforward; there are no mind bending abstractions to get your head around.
- But that doesn't make it any less powerful: the Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object oriented principle.
- Before we introduce that new principle, let's take a look at the official definition of the pattern:
  - ***The Facade Pattern*** *provides a unified interface to a set of interfaces in a subsytem. Facade defines a higher-level interface that makes the subsystem easier to use.*

Facade Pattern defined

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Template Method Patterns

Instructor: Dr. Metin Özkan

It's time for some more caffeine

- Tea and coffee are made in very similar ways.
- Let's check it out:
  - *Coffee Recipe*
    - *Boil some water*
    - *Brew coffee in boiling water*
    - *Pour coffee in cup*
    - *Add sugar and milk*
  - *Tea Recipe*
    - *Boil some water*
    - *Steep tea in boiling water*
    - *Pour tea in cup*
    - *Add lemon*

Whipping up some coffee and tea classes

- Write some code for creating coffee and tea.
  - *Here's the coffee:*

```
class Coffee{
public:
        void prepareRecipe(){
                boilWater();
                brewCoffeeGrinds();
                pourInCup();
                addSugarAndMilk();
        }
        void boilWater(){
                cout<<"Boiling water"<<endl;
        }
        void brewCoffeeGrinds(){
                cout<< "Dripping Coffee through filter“ <<endl;
        }
        void pourInCup(){
                cout<<"Pouring into cup"<<endl;
        }
        void addSugarAndMilk(){
                cout<<"Adding Sugar and Milk"<<endl;
        }
};
```

Whipping up some coffee and tea classes

- and now the Tea...
  - *Here's the tea:*

```
class Tea{
public:
        void prepareRecipe(){
                boilWater();
                steepTeaBag();
                pourInCup();
                addLemon();
        }
        void boilWater(){
                cout<<"Boiling water"<<endl;
        }
        void steepTeaBag(){
                cout<<"Steeping the tea"<<endl;
        }
        void pourInCup(){
                cout<<"Pouring into cup"<<endl;
        }
        void addLemon(){
                cout<<"Adding Lemon"<<endl;
        }
};
```

Abstract the Coffee, Tea

- It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:

Abstract the Coffee, Tea

- Notice that both recipes follow the same algorithm:
  - *Boil some water.*
  - *Use the hot water to extract the coffee or tea.*
  - *Pour the resulting beverage into a cup.*
  - *Add the appropriate condiments to the beverage.*

These two are already abstracted into the base class

These are not abstracted, but are the same, just apply different beverages.

Abstracting prepareRecipe()

1. Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...
   - *The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.*

Coffee

```
void prepareRecipe(){
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
}
```

Tea

```
void prepareRecipe(){
        boilWater();
        stepTeaBag();
        pourInCup();
        addLemon();
}
```

   - *Let's think through this: steeping and brewing aren't so different; they're pretty analogous.*

Abstracting prepareRecipe()

```
void prepareRecipe(){
        boilWater();
        brew();
        pourInCup();
        addCondiments();
}
```

Abstracting prepareRecipe()

2 Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass:

```
class CaffeineBeverage{
public:
        virtual void prepareRecipe(){
                boilWater();
                brew();
                pourInCup();
                addCondiments();
        }
protected:
        virtual void brew() const = 0;
        virtual void addCondiments() const = 0;
private:
        void boilWater() const { cout << "Boiling water" << endl; }
        void pourInCup() const { cout << "Pouring into cup" << endl;}
};
```

Abstracting prepareRecipe()

3 Finally we need to deal with the Coffee and Tea classes.

```
class Tea: public CaffeineBeverage{
public:
        void brew() const {
                cout << "Steeping the tea" << endl;
        }
        void addCondiments() const {
                 cout << "Adding Lemon" << endl;
        }
}

class Coffee: public CaffeineBeverage{
public:
        void brew() const {
                cout << "Dripping Coffee through filter" << endl;
        }
        void addCondiments() const {
                 cout << "Adding Sugar and Milk" << endl;
        }
}
```

Meet the Template Method

- We've just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method:"

```
class CaffeineBeverage{
public:
        virtual void prepareRecipe(){

                boilWater();

                brew();

                pourInCup();

                addCondiments();
        }
protected:
        virtual void brew() const = 0;
        virtual void addCondiments() const = 0;
private:
        void boilWater() const {// implementation }
        void pourInCup() const {// implementation }
}
```

Let's make some tea...

- Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...
  - *Okay, first we need a Tea object...*

    ```
    CaffeineBeverage *myTea = new Tea();
    ```
  - *Then we call the template method:*

    ```
    myTea->prepareRecipe();
    ```
  - *First we boil water:*

    ```
    boilWater();
    ```
  - *Next we need to brew the tea:*

    ```
    brew();
    ```
  - *Pour the tea in the cup:*

    ```
    poutInCup();
    ```
  - *Add the condiment:*

    ```
    addCondiments();
    ```

```
boilWater();
brew();
pourInCup();
addCondiments();
```

The prepareRecipe() method controls the algorithm, no one can change this, and it counts on subclasses to provide some or all of the implementation.

Template Method Pattern defined

- You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:
  - ***The Template Method Pattern*** *defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*
- This pattern is all about creating a template for an algorithm. What's a template?
  - *As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.*

Template Method Pattern defined

- *Let's check out the class diagram:*

Template Method Pattern defined

```
class AbstractClass {
public:
        void templateMethod() {
                primitiveOperation1();
                primitiveOperation2();
                concreteOperation();
        }

        virtual void primitiveOperation1() = 0;

        virtual void primitiveOperation2() = 0;

        void concreteOperation(){
                //implementation here
        }
};
```

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...

Hooked on Template Method...

- A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook.
- There are several uses of hooks; let’s take a look at one now:

Hooked on Template Method...

```
class CaffeineBeverageWithHook {
public:
        virtual void prepareRecipe() {
                boilWater();
                brew();
                pourInCup();
                if( customerWantsCondiments() ) {
                        addCondiments();
                }
        }
        virtual void brew() const = 0;
        virtual void addCondiments() const = 0;
protected:
        virtual void boilWater() const {
                cout << "Boiling water" << endl;
        }
        virtual void pourInCup() const {
                cout << "Pouring into cup" << endl;
        }
        virtual bool customerWantsCondiments() const {
                return true;
        }
};
```

This is a hook because the subclass can override this method, but doesn't have to.

Using the hook

- To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.

Here's where you override the hook and provide your own functionality.

```
class CoffeeWithHook : public CaffeineBeverageWithHook {
public:
        void brew() const {cout << "Dripping Coffee through filter" << endl;}
        void addCondiments() const {cout << "Adding Sugar and Milk" << endl;}
        bool customerWantsCondiments() const {
                bool value = false;
                string answer = getUserInput();
                if( answer=='y') {
                        value = true;
                }
                return value;
        }
 private:
         string getUserInput() const {
                string value;
                cout<<"Would you like milk and sugar with your coffee (y/n)?"<<endl;
                cin >> value;
                return value;
        }
};
```

Let's run the TestDrive

```
int main() {
        CaffeineBeverageWithHook *teaHook = new TeaWithHook();
         CaffeineBeverageWithHook *coffeeHook = new CoffeeWithHook();

        cout << endl << "Making tea..."<< endl;
        teaHook->prepareRecipe();

        cout << endl << "Making coffee..."<< endl;
        coffeeHook->prepareRecipe();

        return 0;
}
```

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The Iterator & Composite Patterns

Instructor: Dr. Metin Özkan

Objectville Diner and Objectville Pancake House Merge

- Pancake House and the Diner will merge. But, there seems to be a slight problem...

- The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price

Objectville Diner and Objectville Pancake House Merge

```
class MenuItem {
private:
        string _name;
        string _description;
        bool _vegetarian;
        double _price;
public:
        MenuItem(string name,string description,bool vegetarian, double price )
        :_name(name),_description(description)
         ,_vegetarian(vegetarian),_price( price){}

        string getName() { return _name; }

        string getDescription() { return _description; }

        double getPrice() { return _price; }

        bool isVegetarian() { return _vegetarian; }

        string toString() {
                stringstream value ( _name );
                value << " $" << _price << endl << _description;
                return value.str();
        }
};
```

A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

Objectville Diner and Objectville Pancake House Merge

```
class PancakeHouseMenu{
private:
   std::vector< MenuItem* > _menuItems;
public:
   PancakeHouseMenu() {
         addItem("K&B's Pancake Breakfast","Pancakes with scrambled eggs, and toast“
                 , true,2.99);
         addItem( "Regular Pancake Breakfast","Pancakes with fried eggs, sausage“
                 , false, 2.99 );
         addItem( "Blueberry Pancakes","Pancakes made with fresh blueberries“
                 , true, 3.49 );
         addItem( "Waffles","Waffles, with your choice of blueberries or strawberries“
                 , true, 3.59 );
  }
  void addItem( string name, string description, bool vegetarian, double price ) {
         MenuItem* menuItem = new MenuItem( name, description, vegetarian, price );
         _menuItems.push_back( menuItem );
  }
  std::vector< MenuItem* > getMenuItems() const {
         return _menuItems;
  }

  // other menu methods here
};
```

Objectville Diner and Objectville Pancake House Merge

```
class DinerMenu
private:
   static const int MAX_ITEMS = 6;
   int _numberOfItems;
   MenuItem** _menuItems;
public:
   DinerMenu() : _numberOfItems( 0 ) {
       _menuItems = new MenuItem*[MAX_ITEMS + 1];
       for( int i = 0; i <= MAX_ITEMS; i++ ) _menuItems[i] = 0;
       addItem("Vegetarian BLT","(Fakin') Bacon with lettuce & tomato on whole wheat“,true,2.99);
       addItem( "BLT","Bacon with lettuce & tomato on whole wheat", false, 2.99 );
       addItem( "Soup of the day","Soup of the day, with a side of potato salad“, false, 3.29 );
       addItem("Hotdog","A hot dog, with saurkraut, relish, onions, topped with cheese“,false,3.05);
       addItem("Steamed Veggies and Brown Rice","Steamed vegetables over brown rice“, true, 3.99 );
       addItem("Pasta","Spaghetti with Marinara Sauce, and a slice of sourdough bread“,true, 3.89 );
   }
   void addItem( string name, string description, bool vegetarian, double price ) {
       MenuItem* menuItem = new MenuItem( name, description, vegetarian, price );
       if( _numberOfItems >= MAX_ITEMS ) {
             cout << "Sorry, menu is full!  Can't add item to menu" << endl;
       } else {
             _menuItems[_numberOfItems] = menuItem;
             _numberOfItems++;        }
   }
   MenuItem** getMenuItems() const {
       return _menuItems;
   }
   int getNumberOfItems(){return _numberOfItems;}
          // other menu methods here
};
```

What's the problem with having two different menu representations?

- To see why having two different menu representations complicates things, let's try implementing a client (called as waitress) that uses the two menus.

```
printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is
```

What's the problem with having two different menu representations?

1. Let's start by stepping through how we'd implement the printMenu() method:

```
PancakeHouseMenu *pancakeHouseMenu = new PancakeHouseMenu();
vector< MenuItem* > breakfastItems = pancakeHouseMenu->getMenuItems();

DinerMenu *dinerMenu = new DinerMenu();
MenuItem** lunchItems = dinerMenu->getMenuItems();
```

The method looks the same, but the calls are returning different types.

What's the problem with having two different menu representations?

2 Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems vector. And to print out the Diner items we'll loop through the Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem *menuItem = breakfastItems[i];
    cout<<menuItem->getName()<<" ";
    cout<<menuItem->getPrice()<<" ";
    cout<<menuItem->getDescription();
}

for (int i = 0; i < lunchItems.getNumberOfItems(); i++)
    MenuItem *menuItem = lunchItems[i];
    cout<<menuItem->getName() << " ";
    cout<<menuItem->getPrice() << " ";
    cout<<menuItem->getDescription();
}
```

Now, we have to implement two different loops to step through the two implementations of the menu items...

...one loop for the vector...

and another for the Array.

What's the problem with having two different menu representations?

3 Implementing every other method is going to be a variation of this theme. We're always going to need to get both menus and use two loops to iterate through their items. If another restaurant with a different implementation is acquired then we'll have three loops.

- We don't want to change the implementations because it would mean rewriting a lot of code that is in each respective menu class.
- It would really be nice if we could find a way to allow to implement the same interface for the menus.

Can we encapsulate the iteration?

1. To iterate through the breakfast items we use the size() method on the vector:

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem *menuItem = breakfastItems[i];
}
```

2. And to iterate through the lunch items we use the Array length field on the MenuItem Array.

```
for (int i = 0; i < lunchItems.getNumberOfItems(); i++) {
    MenuItem *menuItem = lunchItems[i];
}
```

Can we encapsulate the iteration?

3. Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the vector

```
Iterator *iterator = breakfastMenu->createIterator();

while (iterator->hasNext()) {
    MenuItem *menuItem = (MenuItem *)iterator->next();
}
```

4. Let's try that on the Array too:

```
Iterator *iterator = lunchMenu->createIterator();
while (iterator->hasNext()) {
    MenuItem *menuItem = (MenuItem*)iterator->next();
}
```

Meet the Iterator Pattern

- Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

- The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:

Meet the Iterator Pattern

- Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashtables, …
- Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:

- Let's go ahead and implement this Iterator and hook it into the DinerMenu to see how this works...

Adding an Iterator to DinerMenu

- To add an Iterator to the DinerMenu we first need to define the Iterator Interface:

```
template <class T>
class Iterator {
public:
        virtual ~Iterator() = 0 {}
        virtual bool hasNext() const = 0;
        virtual T* next() const = 0;
};
```

Adding an Iterator to DinerMenu

- And now we need to implement a concrete Iterator that works for the Diner menu:

```
class DinerMenuIterator : public Iterator< MenuItem > {
private:
        MenuItem** _items;
        mutable int _position;
public:
        DinerMenuIterator( MenuItem** items ) : _items( items ), _position( 0 ) {}
        MenuItem* next() const {
                MenuItem* menuItem = _items[_position];
                _position++;
                return menuItem;
        }
        bool hasNext() const {
                bool value = true;
                if( _items[_position] == 0 ) {
                        value = false;
                }
                return value;
        }
};
```

Reworking the Diner Menu with Iterator

- Time to work it into the DinerMenu; all we need to do is add one method to create a DinerMenuIterator and return it to the client:

```
class DinerMenu
private:
        static const int MAX_ITEMS = 6;
        int _numberOfItems;
        MenuItem** _menuItems;
public:
        //Constructor here

        //addItem here

        MenuItem** getMenuItems() const {
                return _menuItems;
        }

        Iterator<MenuItem>* createIterator() const {
            return (Iterator< MenuItem >*)( new DinerMenuIterator( _menuItems ) );
        }
        // other menu methods here
};
```

We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

Fixing up the Client code

- Now we need to integrate the iterator code into the Client. We should be able to get rid of some of the redundancy in the process.

- Integration is pretty straightforward: first we create a printMenu() method that takes an Iterator, then we use the createIterator() method on each menu to retrieve the Iterator and pass it to the new method.

Fixing up the Client code

```
class Waitress {
        PancakeHouseMenu* _pancakeHouseMenu;
        DinerMenu* _dinerMenu;
public:
        Waitress(PancakeHouseMenu* pancakeHouseMenu, DinerMenu* dinerMenu) :
                _pancakeHouseMenu(pancakeHouseMenu), _dinerMenu(dinerMenu) {}
        void printMenu() {
             Iterator<MenuItem>* pancakeIterator=_pancakeHouseMenu->createIterator();
                Iterator<MenuItem>* dinerIterator = _dinerMenu->createIterator();

                cout << "MENU\n----\nBREAKFAST" << endl;
                printMenu(pancakeIterator);
                cout << "\nLUNCH" << endl;
                printMenu(dinerIterator);
        }
private:
        void printMenu( Iterator<MenuItem>* iterator ) {
                while( iterator->hasNext() ) {
                        MenuItem* menuItem=dynamic_cast<MenuItem*>(iterator->next());
                          cout << menuItem->getName() << ", ";
                          cout << menuItem->getPrice() << " -- ";
                          cout << menuItem->getDescription() << endl;
                }
        }
};
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Testing our code

```
int main( int argc, char* argv[] ) {
        PancakeHouseMenu *pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu *dinerMenu = new DinerMenu();

        Waitress *waitress = new Waitress(pancakeHouseMenu, dinerMenu );

        waitress->printMenu();

        return 0;
}
```

What we have so far...

These two menus implement the same exact set of methods, but they aren't implementing the same Interface.

The Iterator allows the Waitress to be decoupled from the actual implementation of the concrete classes. She doesn't need to know if a Menu is implemented with an Array, an vector, or with PostIt notes. All she cares is that she can get an Iterator to do her iterating.

PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items implementations.

Iterator Pattern defined

- You've already seen how to implement the Iterator Pattern.
- Now it's time to check out the official definition of the pattern:

  - ***The Iterator Pattern*** *provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

- This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers.

Iterator Pattern defined

- You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important:
  - *once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with any of these aggregates just like the printMenu() method, which doesn't care if the menu items are held in an Array or vector (or anything else that can create an Iterator), as long as it can get hold of an Iterator.*
- The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Iterator Pattern defined

The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

The ConcreteIterator is responsible for managing the current position of the iteration.

What did we do?

We wanted to give the Waitress an easy way to iterate over menu items...

waitress

... and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.

... And the Waitress is more extensible

Which is better for waitress, because now it can use the same code to iterate over any group of objects. And it's better for us because the implementation details aren't exposed.

Just when we thought it was safe...

- Now they want to add a dessert submenu.
  - *Now we have to support not only multiple menus, but menus within menus.*
  - *It would be nice if we could just make the dessert menu an element of the DinerMenu collection, but that won't work as it is now implemented.*

Just when we thought it was safe...

- What we want (something like this):

We can't assign a dessert menu to a MenuItem array. We need a change!

We need

- We need some kind of a tree shaped structure that will accommodate menus, submenus and menu items.

- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.

- We may need to be able traverse the items in a more flexible manner. For instance, we might need to iterate over only the Diner's dessert menu, or we might need to iterate over the Diner's entire menu, including the dessert submenu.

We need

Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.

The Composite Pattern defined

- That's right, we'e going to introduce another pattern to solve this problem. We didn't give up on Iterator – it will still be part of our solution - however, the problem of managing menus has taken on a new dimension that Iterator doesn't solve. So, we're going to step back and solve it with the **Composite Pattern**.
  - ***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.*
- Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus and menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy; that is, a tree of objects that is made of parts (menus and menu items) but that can be treated as a whole, like one big menu.

The Composite Pattern defined

The Client uses the component interface to manipulate the objects in the composition.

The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.

A Leaf has no children.

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.

The Composite's role is to define behavior of the components having children and to store child components.

Designing Menus with Composite

The Waitress is going to use the MenuComponent interface to access both Menus and MenuItems.

Here are the methods for manipulating the components. The components are MenuItem and Menu.

We have some of the same methods you'll remember from our previous versions of MenuItem and Menu, and we've added print(), add(),remove() and getChild().

MenuItem overrides the methods that make sense, and uses the default implementations in MenuComponent for those that don't make sense (like add() - it doesn't make sense to add a component to a MenuItem... we can only add components to a Menu).

Menu also overrides the methods that make sense, like a way to add and remove menu items (or other menus!) from its menuComponents. In addition, we'll use the getName() and getDescription() methods to return the name and description of the menu.

Implementing the Menu Component

MenuComponent provides default implementations for every method.

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, they don't have to do anything, they can just inherit the default implementation.

```
class MenuComponent {
public:
        MenuComponent() {}
        virtual ~MenuComponent() = 0 {}
        virtual void add( MenuComponent* menuComponent )
                {throw new UnsupportedOperationException();}
        virtual void remove( MenuComponent* menuComponent )
                {throw new UnsupportedOperationException();}
        virtual const MenuComponent* getChild( int i ) const
                {throw new UnsupportedOperationException();}
        virtual string getName() const
                {throw new UnsupportedOperationException();}
        virtual string getDescription() const
                {throw new UnsupportedOperationException();}
        virtual double getPrice() const
                {throw new UnsupportedOperationException();}
        virtual bool isVegetarian() const
                {throw new UnsupportedOperationException();}
        virtual void print() const
                {throw new UnsupportedOperationException();}
};
```

Implementing the Menu Item

```
class MenuItem : public MenuComponent {
private:
        string _name;
        string _description;
        bool _vegetarian;
        double _price;
public:
   MenuItem(const string name,const string description,bool vegetarian,double price)
     :_name(name),_description(description),_vegetarian(vegetarian),_price(price) {}
        string getName() const { return _name; }
        string getDescription() const { return _description; }
        double getPrice() const { return _price; }
        bool isVegetarian() const { return _vegetarian; }
        void print() const {
                cout << "  " << getName();
                if( isVegetarian() ) {
                        cout << "(v)";
                }
                cout << ", " << getPrice() << endl;
                cout << "     -- " << getDescription() << endl;
        }
};
```

First we need to extend the MenuComponent interface.

The constructor just takes the name, description, etc. and keeps a reference to them all. This is pretty much like our old menu item implementation.

This is different from the previous implementation. Here we're overriding the print() method in the MenuComponent class. For MenuItem this method prints the complete menu entry: name, description, price and whether or not it's veggie.

Implementing the Composite Menu

Menu is also a MenuComponent, just like MenuItem.

Menu can have any number of children of type MenuComponent, we'll use an internal ArrayList to hold these.

```
class Menu : public MenuComponent {
private:
        string _name;
        string _description;
        mutable vector< MenuComponent* > _menuComponents;
public:
        Menu(const string name,const string description):_name(name),_description(description){}
        void add( MenuComponent* menuComponent ) {
                _menuComponents.push_back( menuComponent );
        }
        void remove( MenuComponent* menuComponent ) {
                remove( _menuComponents.begin(), _menuComponents.end(), menuComponent );
        }
        MenuComponent* getChild( int i ) const { return _menuComponents[i]; }
        string getName() const { return _name; }
        string getDescription() const { return _description; }
        void print() const {
                cout << endl << getName();
                cout << ", " << getDescription() << endl;
                cout << "---------------------" << endl;

                vector< MenuComponent* >::iterator iterator = _menuComponents.begin();
                while( iterator != _menuComponents.end() ) {
                        MenuComponent* menuComponent = *iterator++;
                        menuComponent->print();
                }
        }
};
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Getting ready for a test drive...

```
class Waitress {
private:
        const MenuComponent* _allMenus;
public:
        Waitress( const MenuComponent* allMenus ) :
                _allMenus( allMenus ) { }
        void printMenu() const {
                _allMenus->print()
        }
};
```

The Waitress code really is this simple. Now we just hand her the top level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy - all the menus, and all the menu items - is call print() on the top level menu.

The top level menu holds all menus and items.

Getting ready for a test drive...

```
int main() {
    MenuComponent *pancakeHouseMenu = new Menu( "PANCAKE HOUSE MENU", "Breakfast" );
    MenuComponent *dinerMenu = new Menu( "DINER MENU", "Lunch" );
    MenuComponent *cafeMenu = new Menu( "CAFE MENU", "Dinner" );
    MenuComponent *dessertMenu = new Menu( "DESSERT MENU", "Dessert of course!" );
    MenuComponent *coffeeMenu =
                new Menu( "COFFEE MENU", "Stuff to go with your afternoon coffee" );
    MenuComponent *allMenus = new Menu( "ALL MENUS", "All menus combined" );
    allMenus->add( pancakeHouseMenu );
    allMenus->add( dinerMenu );
    allMenus->add( cafeMenu );
    //add menu items here
    dinerMenu->add( new MenuItem("Pasta",
                  "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
                  true, 3.89) );
    dinerMenu->add( dessertMenu );

    dessertMenu->add( new MenuItem( "Apple Pie",
                  "Apple pie with a flakey crust, topped with vanilla icecream",
                  true, 1.59) );
    Waitress *waitress = new Waitress(allMenus);
    waitress->printMenu();

    return 0;
}
```

Getting ready for a test drive... (NOTE: this output is based on the complete source.)

d:\Courses\Lecturer\152115002_Object_Oriented_Programming_II\book_resources\HeadFirstDesig...

```
ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
```

Getting ready for a test drive... (NOTE: this output is based on the complete source.)

```
DESSERT MENU, Dessert of course!
--------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
--------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole

COFFEE MENU, Stuff to go with your afternoon coffee
--------------------
  Coffee Cake(v), 1.59
     -- Crumbly cake topped with cinnamon and walnuts
  Bagel, 0.69
     -- Flavors include sesame, poppyseed, cinnamon raisin, pumpkin
  Biscotti(v), 0.89
     -- Three almond or hazelnut biscotti cookies
```

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

The State Pattern

Instructor: Dr. Metin Özkan

Gumball Machines

- Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

State machines

- How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines:

1. First, gather up your states:

2. Next, create an instance variable to hold the current state, and define values for each of the states:

```
static int SOLD_OUT = 0;
static int NO_QUARTER = 1;
static int HAS_QUARTER = 2;
static int SOLD = 3;

int state = SOLD_OUT;
```

Here's each state represented as a unique integer...

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

State machines

3 Now we gather up all the actions that can happen in the system:

turns cranck

inserts quarter

ejects quarter

dispence gumball

These actions are the gumball machine's interface - the things you can do with it.

Dispense is more of an internal action the machine invokes on itself.

State machines

4 Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this:

Each possible state is checked with a conditional statement...

```
void insertQuarter(){

   if(state == HAS_QUARTER)
       cout << "You can't insert another quarter" << endl;
   if(state == SOLD_OUT)
       cout << "You can't insert a quarter, the machine is sold out"<<endl;
   if(state == SOLD)
       cout << "Please wait, we're already giving you a gumball " << endl;
   if(state == NO_QUARTER){
        state = HAS_QUARTER;
        cout << "You inserted a quarter " << endl;
   }
}
```

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unflled when it's frst taken out of its box and turned on.

Writing the code

```
class GumballMachine{
private:
    static int SOLD_OUT;
    static int NO_QUARTER;
    static int HAS_QUARTER;
    static int SOLD;
    int state ;
    int count ;
public:
    GumballMachine(int count) {
        state = SOLD_OUT;
        this->count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
```

Writing the code

```
void insertQuarter() {
     if (state == HAS_QUARTER) {
         cout << "You can't insert another quarter" <<endl;
     } else if (state == NO_QUARTER) {
         state = HAS_QUARTER;
         cout <<"You inserted a quarter"<<endl;
     } else if (state == SOLD_OUT) {
        cout <<"You can't insert a quarter, the machine is sold out" <<endl;
     } else if (state == SOLD) {
         cout <<"Please wait, we're already giving you a gumball"<<endl;
     }
 }

 void ejectQuarter() {
     if (state == HAS_QUARTER) {
         cout <<"Quarter returned"<<endl;
         state = NO_QUARTER;
     } else if (state == NO_QUARTER) {
         cout <<"You haven't inserted a quarter"<<endl;
     } else if (state == SOLD) {
         cout <<"Sorry, you already turned the crank"<<endl;
     } else if (state == SOLD_OUT) {
         cout <<"You can't eject, you haven't inserted a quarter yet"<<endl;
     }
 }
```

Writing the code

```
void turnCrank() {
     if (state == SOLD) {
         cout <<"Turning twice doesn't get you another gumball!"<<endl;
     } else if (state == NO_QUARTER) {
         cout <<"You turned but there's no quarter"<<endl;
     } else if (state == SOLD_OUT) {
        cout <<"You turned, but there are no gumballs"<<endl;
     } else if (state == HAS_QUARTER) {
         cout <<"You turned..."<<endl;
         state = SOLD;
         dispense();
     }
 }
```

Writing the code

```
    void dispense() {
        if (state == SOLD) {
            cout <<"A gumball comes rolling out the slot"<<endl;
            count = count - 1;
            if (count == 0) {
                cout <<"Oops, out of gumballs!"<<endl;
                state = SOLD_OUT;
            } else {
                state = NO_QUARTER;
            }
        }
         else if (state == NO_QUARTER) { cout <<"You need to pay frst"<<endl; }
         else if (state == SOLD_OUT) { cout <<"No gumball dispensed"<<endl; }
         else if (state == HAS_QUARTER) { cout <<"No gumball dispensed"<<endl; }
    }
    string toString() const {
         stringstream value; value << endl;
 if(state == HAS_QUARTER) value << "Machine is "<<"waiting for turn of crank"<<endl;
 if(state == NO_QUARTER)value << "Machine is " << "waiting for quarter" << endl;
 if(state == SOLD_OUT) value << "Machine is " << "sold out" << endl;
 if(state == SOLD) value << "Machine is " << "dispensing a gumball" << endl;
         return value.str();}

    // other methods here like toString() and refll()
};
```

Test the code

```
int main() {
        GumballMachine *gumballMachine = new GumballMachine(5);
        cout<<gumballMachine->toString();

        gumballMachine->insertQuarter();
        gumballMachine->turnCrank();
        cout<<gumballMachine->toString();

        gumballMachine->insertQuarter();
        gumballMachine->ejectQuarter();
        gumballMachine->turnCrank();
        cout<<gumballMachine->toString();

        gumballMachine->insertQuarter();
        gumballMachine->turnCrank();
        gumballMachine->insertQuarter();
        gumballMachine->turnCrank();
        gumballMachine->ejectQuarter();
        cout<<gumballMachine->toString();

        gumballMachine->insertQuarter();  gumballMachine->insertQuarter();
        gumballMachine->turnCrank();    gumballMachine->insertQuarter();
        gumballMachine->turnCrank();    gumballMachine->insertQuarter();
        gumballMachine->turnCrank();     cout<<gumballMachine->toString();
        return 0;
}
```

The output

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\...

```
Machine is waiting for quarter
You inserted a quarter
You turned...
A gumball comes rolling out the slot

Machine is waiting for quarter
You inserted a quarter
Quarter returned
You turned but thereÆs no quarter

Machine is waiting for quarter
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You havenÆt inserted a quarter

Machine is waiting for quarter
You inserted a quarter
You canÆt insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You canÆt insert a quarter, the machine is sold out
You turned, but there are no gumballs

Machine is sold out
```

There is a change request!

- Gumball, Inc. think that by turning “gumball buying” into a game for increasing the sales.
- 10% of the time, when the crank is turned, the customer gets two gumballs instead of one.
- In fact, when you go back and look at your code and think about what you’ll have to do to modify it.

There is a change request!

```
static int SOLD_OUT = 0;
static int NO_QUARTER = 1;
static int HAS_QUARTER = 2;
static int SOLD = 3;

public:
        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'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

The new design

- It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.
  1. *First, we're going to define a State interface that contains a method for every action in the Gumball Machine.*
  2. *Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.*
  3. *Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.*

Defining the State interfaces and classes

- First let's create an interface for State, which all our states implement:

Implementing our State classes

```
class NoQuarterState : public State {
private:
        GumballMachine* _gumballMachine;
public:
        NoQuarterState( GumballMachine* gumballMachine ) :
                _gumballMachine( gumballMachine ) { }
        void insertQuarter() const {
                cout << "You inserted a quarter" << endl;
        _gumballMachine->setState(_gumballMachine->getHasQuarterState());
        }
        void ejectQuarter() const {
                cout << "You haven't inserted a quarter" << endl;
        }
        void turnCrank() const {
                cout << "You turned, but there's no quarter" << endl;
        }
        void dispense() {
                cout << "You need to pay first" << endl;
        }
        string toString() const { return "waiting for quarter"; }
};
```

Reworking the Gumball Machine

- Before we finish the State classes, we're going to rework the Gumball Machine - that way you can see how it all fits together. We'll start with the state-related instance variables and switch the code from using integers to using state objects:

```
class GumballMachine{
   static int SOLD_OUT = 0;
   static int NO_QUARTER = 1;
   static int HAS_QUARTER = 2;
   static int SOLD = 3;

   int state = SOLD_OUT;
   int count = 0;
```

Old code

```
class GumballMachine{
   State soldOutState;
   State noQuarterState;
   State hasQuarterState;
   State soldState;

   State state = soldOutState;
   int count = 0;
```

New code

Now, let's look at the complete GumballMachine class...

```
class GumballMachine {
private:
        State* _soldOutState;
        State* _noQuarterState;
        State* _hasQuarterState;
        State* _soldState;
        State* _state;
        int _count;
public:
        GumballMachine(int numberGumballs );
        ~GumballMachine() {
                delete _soldOutState;
                delete _noQuarterState;
                delete _hasQuarterState;
                delete _soldState;
        }
        void insertQuarter() const {
                _state->insertQuarter();
        }
        void ejectQuarter() const {
                _state->ejectQuarter();
        }
```

Now, let's look at the complete GumballMachine class...

```
void turnCrank() const {
        _state->turnCrank();
        _state->dispense();
}
void setState( State* state ) {
        _state = state;
}
void releaseBall() {
        cout << "A gumball comes rolling out the slot..." << endl;
        if( _count != 0) {
                _count--;
        }
}
int getCount() const {
        return _count;
}
void refill( int count ) {
        _count = count;
        _state = _noQuarterState;
}
State* getState() const { return _state; }
State* getSoldOutState() const { return _soldOutState; }
```

Now, let's look at the complete GumballMachine class...

```
    State* getNoQuarterState() const {
        return _noQuarterState;
    }
    State* getHasQuarterState() const {
        return _hasQuarterState;
    }
    State* getSoldState() const {
        return _soldState;
    }
    string toString() const {
         stringstream value;
         value << endl << "Mighty Gumball, Inc.";
         value << endl << "C++-enabled Standing Gumball Model #2004";
         value << endl << "Inventory: " << _count << " gumball";
         if( _count > 1 ) {
                 value << "s";
         }
         value << endl;
         value << "Machine is " << _state->toString() << endl;
         return value.str();
    }
};
```

Implementing more states

```
class HasQuarterState : public State {
private:
        GumballMachine* _gumballMachine;
public:
        HasQuarterState( GumballMachine* gumballMachine ) :
                _gumballMachine( gumballMachine ){}
        void insertQuarter() const {
                cout << "You can't insert another quarter" << endl;
        }
        void ejectQuarter() const {
                cout << "Quarter returned" << endl;
                _gumballMachine->setState( _gumballMachine->getNoQuarterState() );
        }
        void turnCrank() const {
                cout << "You turned..." << endl;
                _gumballMachine->setState( _gumballMachine->getSoldState() );

        }
        void dispense() { cout << "No gumball dispensed" << endl; }
        string toString() const { return "waiting for turn of crank"; }
};
```

Implementing more states

```
class SoldOutState : public State {
private:
        GumballMachine* _gumballMachine;
public:
        SoldOutState(GumballMachine* gumballMachine ) :
                _gumballMachine( gumballMachine ) {    }
        void insertQuarter() const {
             cout << "You can't insert a quarter, the machine is sold out" << endl;
        }
        void ejectQuarter() const {
             cout << "You can't eject, you haven't inserted a quarter yet" << endl;
        }
        void turnCrank() const {
                cout << "You turned, but there are no gumballs" << endl;
        }
        void dispense() {
                cout << "No gumball dispensed" << endl;
        }
        string toString() const {
                return "sold out";
        }
};
```

Let's take a look at what we've done so far...

- For starters, you now have a Gumball Machine implementation that is structurally quite different from the first version, and yet functionally it is exactly the same. By structurally changing the implemention you've:
  - *Localized the behavior of each state into its own class.*
  - *Removed all the troublesome if statements that would have been difficult to maintain.*
  - *Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes (and we'll do this in a second).*
  - *Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.*

Let's take a look at what we've done so far...

Let's take a look at what we've done so far...

In this case the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.

TRANSITION TO SOLD STATE

Let's take a look at what we've done so far...

The State Pattern defined

- ***The State Pattern*** *allows an object to alter its behavior when its internal state changes. The object will appear to change its class.*

- The first part of this description makes a lot of sense. Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state.
- What about the second part of the definition? What does it mean for an object to "appear to change its class?" Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

The State Pattern defined

The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.

The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.

Whenever the request() is made on the Context it is delegated to the state to handle.

ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

The State Pattern defined

- Strategy Pattern is EXACTLY the same?
- Yes, the class diagrams are essentially the same, but the two patterns differ in their intent.
- With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

The State Pattern defined

- With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

The State Pattern defined

- In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.
- Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

We still need to finish the Gumball 1 in 10 game

```
class GumballMachine {
private:
        State* _soldOutState;
        State* _noQuarterState;
        State* _hasQuarterState;
        State* _soldState;
        State* _winnerState;

        State* _state;
        int _count;
public:
        // methods here
};
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

We still need to finish the Gumball 1 in 10 game

```
class WinnerState : public State {
    // instance variables and constructor

    // insertQuarter error message

    // ejectQuarter error message

    // turnCrank error message

    void dispense() {
        cout << "YOU'RE A WINNER! You get two gumballs for your quarter" << endl;
        _gumballMachine->releaseBall();
        if( _gumballMachine->getCount() == 0 ) {
            _gumballMachine->setState( _gumballMachine->getSoldOutState() );
        } else {
            _gumballMachine->releaseBall();
            if( _gumballMachine->getCount() > 0 ) {
                _gumballMachine->setState( _gumballMachine->getNoQuarterState() );
            } else {
                cout << "Oops, out of gumballs!" << endl;
                _gumballMachine->setState( _gumballMachine->getSoldOutState() );
            }
        }
    }
};
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

Finishing the game

```
class HasQuarterState : public State {
private:
        int _random;
        GumballMachine* _gumballMachine;
public:
        HasQuarterState( GumballMachine* gumballMachine ) :
                _gumballMachine( gumballMachine ),_random(0) {srand( _getpid() ); }
        void insertQuarter() const {
                cout << "You can't insert another quarter" << endl;
        }
        void ejectQuarter() const {
                cout << "Quarter returned" << endl;
                _gumballMachine->setState( _gumballMachine->getNoQuarterState() );
        }
        void turnCrank() const {
                cout << "You turned..." << endl;
                int winner = rand() % 5;
                if( ( winner == 0 ) && ( _gumballMachine->getCount() > 0 ) ) {
                     _gumballMachine->setState(_gumballMachine->getWinnerState() );
                } else {
                     _gumballMachine->setState( _gumballMachine->getSoldState() );
                }
        }
        void dispense() { cout << "No gumball dispensed" << endl; }
        string toString() const { return "waiting for turn of crank"; }
};
```

First we add a random number generator to generate the 10% chance of winning...

...then we determine if this customer won.

Demo for the CEO of Mighty Gumball, Inc.

This code really hasn't changed at all;

```
int main() {
        GumballMachine *gumballMachine = new GumballMachine( 10 );
        cout << gumballMachine->toString() << endl;

        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        cout << gumballMachine->toString() << endl;

        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        cout << gumballMachine->toString() << endl;

        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        cout << gumballMachine->toString() << endl;

        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        cout << gumballMachine->toString() << endl;

        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        gumballMachine->insertQuarter();  gumballMachine->turnCrank();
        cout << gumballMachine->toString() << endl;

        return 0;
}
```

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\Debug\OOPII_App.exe

```
Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 10 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 7 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
```

```
Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
Oops, out of gumballs!

Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 0 gumball
Machine is sold out

You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs
No gumball dispensed
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs
No gumball dispensed

Mighty Gumball, Inc.
C++-enabled Standing Gumball Model #2004
Inventory: 0 gumball
Machine is sold out
```

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*

Compound Patterns

Instructor: Dr. Metin Özkan

Working together

- One of the best ways to use patterns is to get them out of the house so they can interact with other patterns.
- The more you use patterns the more you're going to see them showing up together in your designs.
- We have a special name for a set of patterns that work together in a design that can be applied over many problems: a compound pattern.
- That's right, we are now talking about patterns made of patterns!

- Patterns are often used together and combined within the same design solution.
- A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.

Duck reunion

- We're going to get to work with the ducks again.
- This time the ducks are going to show you how patterns can coexist and even cooperate within the same solution.
- We're going to rebuild our duck simulator from scratch and give it some interesting capabilities by using a bunch of patterns.

1. **First, we'll create a Quackable interface.**
   - We're starting from scratch. This time around, the Ducks are going to implement a Quackable interface. That way we'll know what things in the simulator can quack() – like Mallard Ducks, Redhead Ducks, Duck Calls, and we might even see the Rubber Duck.

```
class Quackable{
public:
        virtual void quack();
};
```

Quackables only need to do one thing well: Quack!

Duck reunion

2 Now, some Ducks that implement Quackable

- Time to create some concrete.

Your standard Mallard duck.

```
class MallardDuck: public Quackable{
public:
        void quack(){ cout << "Quack" << endl;}
};

Class RedheadDuck: public Quackable{
public:
        void quack(){ cout << "Quack" << endl;}
};
```

We've got to have some variation of species if we want this to be an interesting simulator.

Duck reunion

- **This wouldn't be much fun if we didn't add other kinds of Ducks too.** We had duck calls (those things hunters use, they are definitely quackable) and rubber ducks.

```
class DuckCall: public Quackable{
public:
        void quack(){ cout << "Kwak" << endl;}
};

Class RubberDuck: public Quackable{
public:
        void quack(){ cout << "Squeak" << endl;}
};
```

A DuckCall that quacks but doesn't sound quite like the real thing.

A RubberDuck that makes a squeak when it quacks.

Duck reunion

3 Okay, we've got our ducks; now all we need is a simulator.

- Let's cook up a simulator that creates a few ducks and makes sure their quackers are working...

```
void simulate( const Quackable* duck ) {
        duck->quack();
}
int main() {
        Quackable *mallardDuck = new MallardDuck();
        Quackable *redheadDuck = new RedheadDuck();
        Quackable *duckCall = new DuckCall();
        Quackable *rubberDuck = new RubberDuck();

        cout << endl << "Duck Simulator" << endl;
        simulate( mallardDuck );
        simulate( redheadDuck );
        simulate( duckCall );
        simulate( rubberDuck );

        return 0;
}
```

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

Duck reunion

- It looks like everything is working; so far, so good.

Duck reunion (Adapter)

4 When ducks are around, geese can't be far.

- Here's a Goose class that has been hanging around the simulator.

```
class Goose {
public:
        void honk() {
                cout << "Honk" << endl;
        }
};
```

A Goose is a honker, not a quacker.

5 We need a goose adapter.

- Our simulator expects to see Quackable interfaces. Since geese aren't quackers (honkers), we can use an adapter to adapt a goose to a duck.

```
class GooseAdapter : public Quackable {
private:
        Goose *_goose;
public:
        GooseAdapter( Goose* goose ) : _goose( goose ) { }
        void quack() const { _goose->honk(); }
};
```

The constructor takes the goose we are going to adapt.

When quack is called, the call is delegated to the goose's honk() method.

Duck reunion (Adapter)

6 Now geese should be able to play in the simulator, too.

- All we need to do is create a Goose, wrap it in an adapter that implements Quackable, and we should be good to go.

```
void simulate( const Quackable* duck ) {
        duck->quack();
}
int main() {
        Quackable *mallardDuck = new MallardDuck();
        Quackable *redheadDuck = new RedheadDuck();
        Quackable *duckCall = new DuckCall();
        Quackable *rubberDuck = new RubberDuck();
        Quackable *gooseDuck = new GooseAdapter( new Goose() );
        cout << endl << "Duck Simulator: With Goose Adapter" << endl;
        simulate( mallardDuck );
        simulate( redheadDuck );
        simulate( duckCall );
        simulate( rubberDuck );
        simulate( gooseDuck );
        return 0;
}
```

We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackables.

Duck reunion (Adapter)

7 Now let's give this a quick run....

- This time when we run the simulator, the list of objects passed to the simulate() method includes a Goose wrapped in a duck adapter. The result? We should see some honking!

There's the goose! Now the Goose can quack with the rest of the Ducks.

Duck reunion (Decorator)

8 **We're going to add the ability to count duck quack without having to change the duck classes.**

- How? Let's create a decorator that gives the ducks some new behavior (the behavior of counting) by wrapping them with a decorator object. We won't have to change the Duck code at all.

```
class QuackCounter : public Quackable {
private:
        Quackable *_duck;
        static int _numberOfQuacks;
public:
        QuackCounter ( Quackable* duck ) : _duck( duck ) { }
        void quack() const {
                _duck->quack();
                _numberOfQuacks++;
        }
        static int getQuacks() {
                return _numberOfQuacks;
        }
};
```

QuackCounter is a decorator

We've got an instance variable to hold on to the quacker we're decorating.

And we're counting ALL quacks, so we'll use a static variable to keep track.

We get the reference to the Quackable we're decorating in the constructor.

Duck reunion (Decorator)

9 We need to update the simulator to create decorated ducks.

- Now, we must wrap each Quackable object we instantiate in a QuackCounter decorator.

```
void simulate( const Quackable* duck ) {
        duck->quack();
}
int main() {
        Quackable *mallardDuck = new QuackCounter( new MallardDuck());
        Quackable *redheadDuck = new QuackCounter( new RedheadDuck());
        Quackable *duckCall = new QuackCounter( new DuckCall());
        Quackable *rubberDuck = new QuackCounter( new RubberDuck());
        Quackable *gooseDuck = new GooseAdapter( new Goose());
        cout << endl << "Duck Simulator: With Decorator" << endl;
        simulate(mallardDuck );
        simulate(redheadDuck );
        simulate( duckCall );
        simulate(rubberDuck );
        simulate( gooseDuck );
cout<<"The ducks quacked "<<QuackCounter::getQuacks()<<" times"<<endl;
        return 0;
}
```

Nothing changes here; the decorated objects are still Quackables.

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

Each time we create a Quackable, we wrap it with a new decorator.

Duck reunion (Decorator)

Duck reunion (Factory)

10 We need a factory to produce ducks!

- Okay, we need some quality control to make sure our ducks get wrapped. We're going to build an entire factory just to produce them. The factory should produce a family of products that consists of different types of ducks, so we're going to use the Abstract Factory Pattern.

```
class AbstractDuckFactory {
public:
        virtual ~AbstractDuckFactory() = 0 { }
        virtual Quackable* createMallardDuck() const = 0;
        virtual Quackable* createRedheadDuck() const = 0;
        virtual Quackable* createDuckCall() const = 0;
        virtual Quackable* createRubberDuck() const = 0;
};
```

We're defining an abstract factory that subclasses will implement to create different families.

Each method creates one kind of duck.

Duck reunion (Factory)

- Now let's create the factory we really want, the CountingDuckFactory:

CountingDuckFactory also extends the abstract factory.

```
class CountingDuckFactory : public AbstractDuckFactory {
public:
        Quackable* createMallardDuck() const {
                return new QuackCounter( new MallardDuck() );
        }
        Quackable* createRedheadDuck() const {
                return new QuackCounter( new RedheadDuck() );
        }
        Quackable* createDuckCall() const {
                return new QuackCounter( new DuckCall() );
        }
        Quackable* createRubberDuck() const {
                return new QuackCounter( new RubberDuck() );
        }
};
```

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks are being counted.

Duck reunion (Factory)

11 Let's set up the simulator to use the factory.

- We create a polymorphic method that takes a factory and uses it to create objects. By passing in different factories, we get to use different product families in the method.

First we create the factory that we're going to pass into the simulate() method.

```
int main() {
        AbstractDuckFactory *duckFactory = new CountingDuckFactory();
        Quackable *mallardDuck = duckFactory->createMallardDuck();
        Quackable *redheadDuck = duckFactory->createRedheadDuck();
        Quackable *duckCall = duckFactory->createDuckCall();
        Quackable *rubberDuck = duckFactory->createRubberDuck();
        Quackable *gooseDuck = new GooseAdapter( new Goose() );
       cout << endl << "Duck Simulator: With Abstract Factory" << endl;
        simulate( mallardDuck );
        simulate( redheadDuck );
        simulate( duckCall );
        simulate( rubberDuck );
        simulate( gooseDuck );
cout<<"The ducks quacked "<<QuackCounter::getQuacks() << " times" << endl;
        return 0;
}
```

Duck reunion (Factory)

- Here's the output using the factory...

Same as last time, but this time we're ensuring that the ducks are all decorated because we are using the CountingDuckFactory.

Duck reunion (Composite)

- It's getting a little difficult to manage all these different ducks separately. Is there any way you can help us manage ducks as a whole, and perhaps even allow us to manage a few duck "families" that we'd like to keep track of?

12 **Let's create a flock of ducks (well, actually a flock of Quackables).**

- Remember the Composite Pattern that allows us to treat a collection of objects in the same way as individual objects? What better composite than a flock of Quackables!

Duck reunion (Composite)

```
class Flock : public Quackable {
private:
        mutable vector<  Quackable* > _quackers;
public:
        Flock() { }
        void add ( Quackable* quacker ) {
                _quackers.push_back( quacker );
        }
        void quack() const {
                for(vector<Quackable*>::iterator iterator=_quackers.begin()
                        ;_quackers.end()!=iterator;++iterator){
                        Quackable* quacker = *iterator;
                        quacker->quack();
                }
        }
};
```

Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

We're using an ArrayList inside each Flock to hold the Quackables that belong to the Flock.

The add() method adds a Quackable to the Flock..

Now for the quack() method - after all, the Flock is a Quackable too. The quack() method in Flock needs to work over the entire Flock. Here we iterate through the ArrayList and call quack() on each element.

Duck reunion (Composite)

13 Now we need to alter the simulator.

- Our composite is ready; we just need some code to round up the ducks into the composite structure.

Create all the Quackables, just like before.

```
int main() {

        AbstractDuckFactory *duckFactory = new CountingDuckFactory();

        Quackable *redheadDuck = duckFactory->createRedheadDuck();
        Quackable *duckCall = duckFactory->createDuckCall();
        Quackable *rubberDuck = duckFactory->createRubberDuck();
        Quackable *gooseDuck = new GooseAdapter( new Goose());

        cout << endl << "Duck Simulator: With Composite - Flocks" << endl;

        Flock *flockOfDucks = new Flock();

        flockOfDucks->add( redheadDuck );
        flockOfDucks->add( duckCall );
        flockOfDucks->add( rubberDuck);
        flockOfDucks->add( gooseDuck );
```

First we create a Flock, and load it up with Quackables.

Duck reunion (Composite)

```
        Flock *flockOfMallards = new Flock();

        Quackable *mallardOne = duckFactory->createMallardDuck();
        Quackable *mallardTwo = duckFactory->createMallardDuck();
        Quackable *mallardThree = duckFactory->createMallardDuck();
        Quackable *mallardFour = duckFactory->createMallardDuck();

        flockOfMallards->add( mallardOne );
        flockOfMallards->add( mallardTwo );
        flockOfMallards->add( mallardThree );
        flockOfMallards->add( mallardFour );
        flockOfDucks->add( flockOfMallards );

        cout << endl << "Duck Simulator: Whole Flock Simulation" << endl;
        simulate( flockOfDucks );

        cout << endl << "Duck Simulator: Mallard Flock Simulation" << endl;
        simulate( flockOfMallards );

cout<<endl<<"The ducks quacked “<<QuackCounter::getQuacks()<<" times”<<endl;
        return 0;
}
```

Duck reunion (Composite)

c:\Users\metis\Documents\Visual Studio 2008\Projects\OOPII_App\Debug\OOPII_App.exe

```
Duck Simulator: With Composite - Flocks

Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

Duck reunion (Observer)

- The Composite is working great! Now we have the opposite request: we also need to track individual ducks. Can you give us a way to keep track of individual duck quacking in real time?

14 **First we need an Observable interface.**

- Remember that an Observable is the object being observed. An Observable needs methods for registering and notifying observers. We could also have a method for removing observers, but we'll keep the implementation simple here and leave that out.

```
class QuackObservable {
public:
        virtual ~QuackObservable() = 0 {}
        virtual void registerObserver(Observer* observer) = 0;
        virtual void notifyObservers() const = 0;
};
```

QuackObservable is the interface that Quackables should implement if they want to be observed.

It also has a method for notifying the observers.

It has a method for registering Observers. Any object implementing the Observer interface can listen to quacks. We'll define the Observer interface in a sec.

Duck reunion (Observer)

- Now we need to make sure all Quackables implement this interface...

```
class Quackable : public QuackObservable {
public:
        virtual ~Quackable() = 0 { }
        virtual void quack() const= 0;
        virtual string toString() const = 0;
};
```

So, we extend the Quackable interface with QuackObserver.

Duck reunion (Observer)

15 **Now, we need to make sure all the concrete classes that implement Quackable can handle being a QuackObservable.**

- We could approach this by implementing registration and notification in each and every class. But we're going to do it a little differently this time: we're going to encapsulate the registration and notification code in another class, call it Observable, and compose it with a QuackObservable. That way we only write the real code once and the QuackObservable just needs enough code to delegate to the helper class Observable.
- Let's start with the Observable helper class...

Duck reunion (Observer)

Observable implements all the functionality a Quackable needs to be an observable. We just need to plug it into a class and have that class delegate to Observable.

```
class Observable : public QuackObservable {
private:
        mutable vector< Observer* > _observers;
        QuackObservable* _duck;
public:
        Observable( QuackObservable* duck ) :_duck( duck ) { }

        void registerObserver( Observer* observer ) {
                _observers.push_back( observer );
        }

        void notifyObservers() const {
                vector< Observer* >::iterator iterator=_observers.begin();
                while( iterator != _observers.end() ) {
                        Observer* observer = *iterator++;
                        observer->update( _duck );
                }
        }
};
```

Here's the code for registering an observer.

And the code for doing the notifications.

Duck reunion (Observer)

16 Integrate the helper Observable with the Quackable classes.

- This shouldn't be too bad. All we need to do is make sure the Quackable classes are composed with an Observable and that they know how to delegate to it. After that, they're ready to be Observables. Here's the implementation of MallardDuck; the other ducks are the same.

```
class MallardDuck : public Quackable {
private:
        Observable *_observable;
public:
        MallardDuck(){ _observable = new Observable( this ); }
        void quack() const{
                cout << "Quack" << endl;
                notifyObservers();
        }
        void registerObserver( Observer* observer ) {
                _observable->registerObserver( observer );
        }
        void notifyObservers() const { _observable->notifyObservers();}
};
```

Each Quackable has an Observable instance variable.

In the constructor, we create an Observable and pass it a reference to the MallardDuck object.

When we quack, we need to let the observers know about it.

Duck reunion (Observer)

17 We're almost there! We just need to work on the Observer side of the pattern.

- We've implemented everything we need for the Observables; now we need some Observers. We'll start with the Observer interface:

```
class Observer {
public:
        virtual void update(QuackObservable* duck ) = 0;
        virtual ~Observer() = 0 {}
};
```

The Observer interface just has one method, update(), which is passed the QuackObservable that is quacking.

- Now we need an Observer: where are those Quackologists?!

```
class Quackologist : public Observer {
public:
        void update( QuackObservable* duck ) {
          cout << "Quackologist: " << (Quackable*)( duck )->toString()
                << " just quacked." << endl;
        }
        string toString() const { return "Quackologist"; }
};
```

Duck reunion (Observer)

18 **We're ready to observe. Let's update the simulator :**

```
int main() {
        AbstractDuckFactory *duckFactory = new CountingDuckFactory();

        Quackable *redheadDuck = duckFactory->createRedheadDuck();
        Quackable *duckCall = duckFactory->createDuckCall();
        Quackable *rubberDuck = duckFactory->createRubberDuck();
        Quackable *gooseDuck = new GooseAdapter( new Goose() );

        Flock *flockOfDucks = new Flock();

        flockOfDucks->add(redheadDuck );
        flockOfDucks->add( duckCall );
        flockOfDucks->add(rubberDuck );
        flockOfDucks->add( gooseDuck );

        Flock *flockOfMallards = new Flock();

        Quackable *mallardOne = duckFactory->createMallardDuck();
        Quackable *mallardTwo = duckFactory->createMallardDuck();
        Quackable *mallardThree = duckFactory->createMallardDuck();
        Quackable *mallardFour = duckFactory->createMallardDuck();
```

Duck reunion (Observer)

```
        flockOfMallards->add( mallardOne );
        flockOfMallards->add( mallardTwo );
        flockOfMallards->add( mallardThree );
        flockOfMallards->add( mallardFour );

        flockOfDucks->add(flockOfMallards );

        cout << endl << "Duck Simulator: With Observer" << endl;

        Quackologist *quackologist = new Quackologist();
        flockOfDucks->registerObserver( quackologist );

        simulate( flockOfDucks );

        cout<<endl<<"The ducks quacked "
               << QuackCounter::getQuacks() << " times" << endl;

        return 0;
}
```

All we do here is create a Quackologist and set him as an observer of the flock.

Duck reunion (Observer)

- This is the big finale. Five, no, six patterns have come together to create this amazing Duck Simulator. Without further ado, we present the DuckSimulator!

A duck's eye view: the class diagram

- We'e packed a lot of patterns into one small duck simulator! Here's the big picture of what we did:

A duck's eye view: the class diagram

- References
  - *Head First Design Patterns, Freeman and Freeman, O'Reilly, 2004*
  - *Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995.*