CSC 7322 : Object Oriented Development

J Paul Gibson, A207

paul.gibson@int-edu.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/

Design Patterns

…/~gibson/Teaching/CSC7322/L5-DesignPatterns.pdf

Introduction

pattern noun **1** a model, guide or set of instructions for making something • *a dress pattern*. **2** a decorative design, often consisting of repeated motifs, eg on wallpaper or fabric. **3** a piece, eg of fabric, as a sample. **4** any excellent example suitable for imitation. **5** a coherent series of occurrences or set of features • *a pattern of events*. verb (***patterned***, ***patterning***) (*usually* **pattern something on another thing**) to model it on another type, design, etc.

ETYMOLOGY: 14c as *patron*; French, from Latin *patronus* example or defender.

SOURCE - Chambers 21st Century Dictionary

Introduction

design verb (***designed***, ***designing***) **1** to develop or prepare a plan, drawing or model of something before it is built or made. **2** *formal* to plan, intend or develop something for a particular purpose. noun **1** a plan, drawing or model showing how something is to be made. **2** the art or job of making such drawings, plans, etc. **3** the way in which something has been made. **4** a picture, pattern, arrangement of shapes, etc used eg as decoration. **5** a plan, purpose or intention. **designable** adj. **designedly** adverb intentionally; on purpose. **designing** adj, *derog* using cunning and deceit to achieve a purpose. **designingly** adverb. **by design** intentionally. **have designs on someone** or **something** to have plans to appropriate them or it.

ETYMOLOGY: 16c: from French *désigner*.

SOURCE - Chambers 21st Century Dictionary

Introduction

design pattern

programming

A description of an object-oriented design technique which names, abstracts and identifies aspects of a design structure that are useful for creating an object-oriented design. The design pattern identifies classes and instances, their roles, collaborations and responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in the presence of other design constraints, and the consequences and trade-offs of its use.

Home (http://st-www.cs.uiuc.edu/users/patterns/patterns.html)

["Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides].

(1997-07-21)

SOURCE – Free On-Line Dictionary Of Computing (http://foldoc.org/)

Introduction

1970's - **Christopher Alexander** (an architect) worked on « *Pattern Languages* » which were applied in many domains:

> *a structured method of describing good design practices within a field of expertise.*

Using a pattern language permits non-experts in a field to successfully solve very large, complex design problems.

A single problem is documented with its typical place (the syntax), and use (the grammar) with the most common and recognized good solution seen in the real world, like the entries seen in dictionaries.

Such an entry is considered to be a single design pattern; and a rich set of patterns form a language.

Introduction

Early research:

THING-MODEL-VIEW-EDITOR an Example from a planning system, Trygve Reenskaug, 1979.

Using Pattern Languages for Object-Oriented Programs,Kent Beck and Ward Cunningham, 1987.

A Cookbook for Using View-Controller User the Model-Interface Paradigm in Smalltalk-80, Krasner and Pope, 1988.

Introduction

Breakthrough into mainstream: Gang of Four (GOF)

Design Patterns: Abstraction and Reuse of Object-Oriented Design, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1993

[BOOK] Design patterns: elements of reusable object-oriented software
E Gamma, R Helm, R Johnson... - 1995 - cs.up.ac.za
Designing **object-oriented software** is hard, and designing **reusable object- oriented software** is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key **...**
Cited by 21300 - Related articles - View as HTML - SUDOC Catalogue - All 22 versions

Introduction

Further reading:

•*Design Patterns for Object-Oriented Software Development*, Pree and Sikora, 1997

•*The origins of pattern theory: the future of the theory, and the generation of a living world*, C. Alexander, 1999

•*Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools*, Greenfield and Short, 2003.

•*The Model-View-Controller (MVC) Its Past and Present*, Trygve Reenskaug, 2003.

•*What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*, O'Reilly, 2007

Definitions

Alexander :

As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Definitions

Erich Gamma (GoF):

Patterns provide you with tools that help you with design problems. They do so not by giving a pat solution but by explaining trade-offs. Even though patterns are abstracted from concrete uses, they also provide you valuable implementation hints. From my perspective it is the fact that patterns are implementable that makes them so valuable.

Patterns are distilled from the experiences of experts. They enable you to repeat a successful design done by someone else. However, since patterns enable many implementation variations you still have to keep the brain turned on.

Since patterns provide you with names for design building blocks they provide you with a vocabulary to describe and discuss a particular design.

I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior.

Elements of Design Patterns

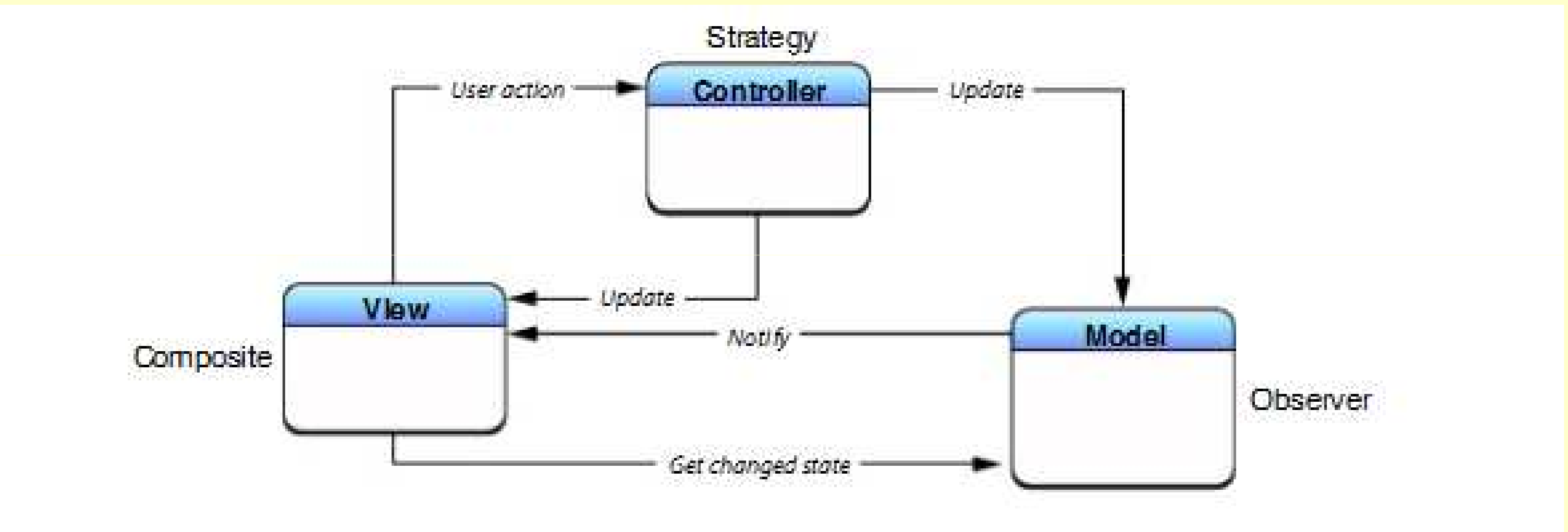
- Pattern Name
- Problem statement -context where it might be applied
- Solution - elements of the design, their relations, responsibilities, and collaborations (including a template of the solution)
- Consequences - Results and trade-offs

The 23 GoF patterns are generally considered the foundation for all other patterns. They are categorized in three groups: *Creational, Structural, and Behavioral.*

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

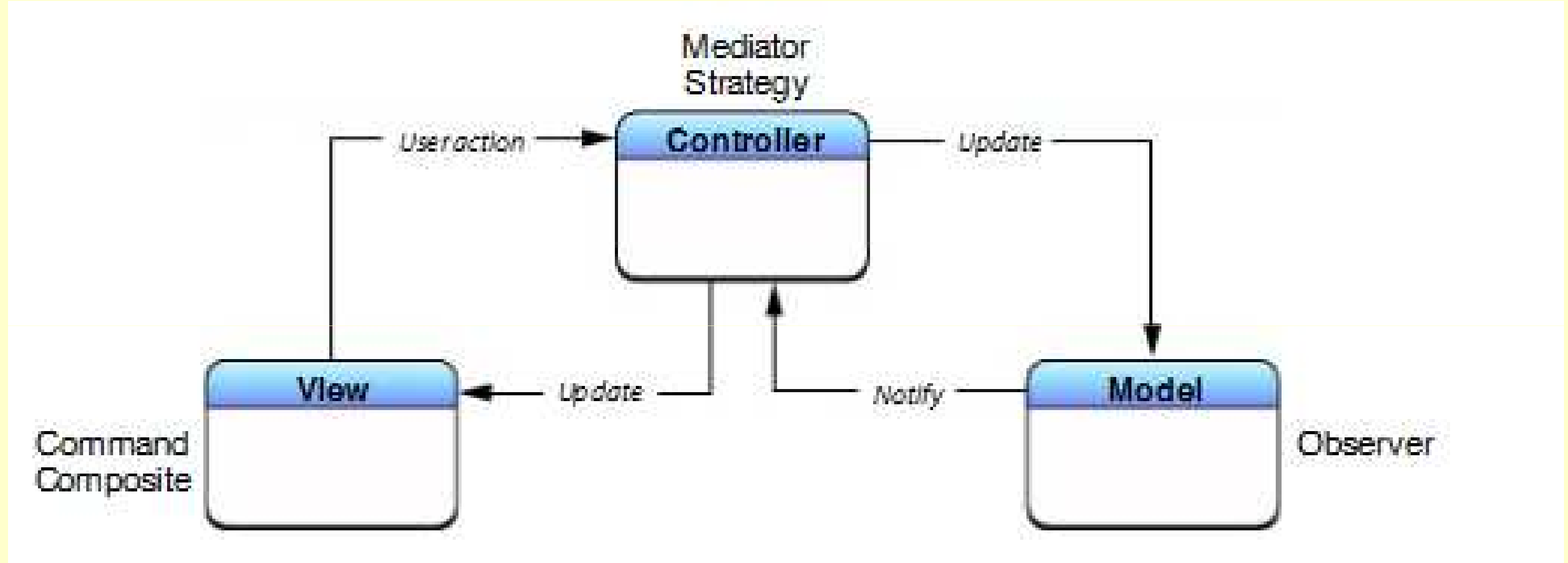
Design pattern composition: MVC example

Traditional version of MVC as a compound pattern



Design pattern composition: MVC example

Alternative version of MVC as a compound pattern



Learning by PBL

- The best way to learn about patterns is to look at examples.

- We shall do these in UML/Java, but any (OO) language can be used to model/implement/re-use a pattern.

- It is best if you discover patterns yourself, rather than being shown them – but this is not guaranteed to happen!

- You can also read about them. A good web site is: http://sourcemaking.com/design_patterns

Learning by PBL – the patterns selected

1. Singleton - creational
2. Decorator - structural
3. Factory - creational
4. Iterator - behavioural
5. Visitor - behavioural
6. MVC – a composite pattern

NOTE: you should have a reasonable understanding of all 23 patterns, and a good understanding of implementation concerns in at least 2 different OO languages.

If you wish to be considered an expert in a particularly OO language then you should have complete mastery of the patterns in that language.

The Singleton Design Pattern

See - http://sourcemaking.com/design_patterns/singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated “just-in-time initialization” or “initialization on first use”.
- Problem
  - Application needs one, and only one, instance of an object.
  - Additionally, lazy initialization and global access are necessary.

The Singleton Design Pattern

See - http://sourcemaking.com/design_patterns/singleton

Relation to other patterns

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.

- Facade objects are often Singletons because only one Facade object is required.

- The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton.

- The Singleton design pattern is one of the most inappropriately used patterns. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable.

The Singleton Design Pattern

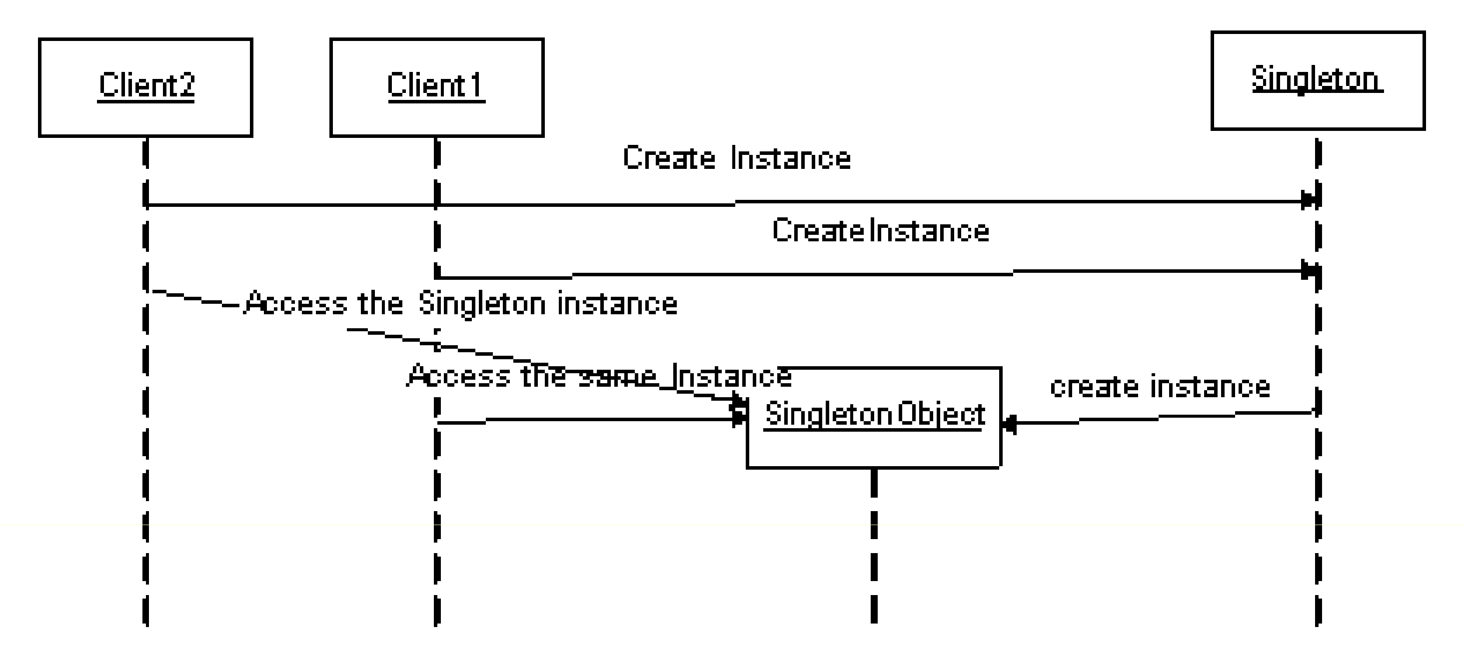
Singleton

- instance: Singleton

- Singleton ()
+ getInstance () : Singleton

UML class diagram

The Singleton Design Pattern



UML sequence diagram

The Singleton Design Pattern

Problem: Examine the 4 Singleton Java implementations in the folder Patterns (http://www-public.int-evry.fr/~gibson/Teaching/CSC7322/Code/Patterns.zip)

Question: what are the differences between the implementations, and which best corresponds to our requirements/design?

The Singleton Design Pattern – example implementation (1)

```
package p_singleton;

public class Singleton1 {

protected static Singleton1 uniqueInstance = null;

private int data;

public synchronized static Singleton1 instance() {
if(uniqueInstance == null) uniqueInstance = new Singleton1();
return uniqueInstance;
}

protected Singleton1() {data=0;}

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (2)

```
package p_singleton;

public class Singleton2 {

public static final Singleton2 uniqueinstance = new Singleton2();

private int data;

private Singleton2() {data=0;}

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (3)

```
public class Singleton3 {

private static final Singleton3 instance = new Singleton3();

private int data;

private Singleton3() { data=0; }

public static Singleton3 instance() {return instance;}

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (4)

```
public class Singleton4 {

private int data;

private Singleton4() { data=0; }

private static class SingletonHolder {
    private static final Singleton4 INSTANCE = new Singleton4();
  }

public static Singleton4 getInstance() {
    return SingletonHolder.INSTANCE;
  }

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – what about subclassing?

The Singleton Design Pattern is meant to give you control over access to the Singleton class. But subclassing allows other code to access your class without you having direct control

The uniqueness of the class cannot be imposed as a compile-time constraint on the subclass unless you use a private constructor (or declare the class to be final).

If you want to allow subclassing, for example, you might make the constructor protected, but then a subclass could provide a public constructor, allowing anyone to make instances.

QUESTION: which of these 2 *subclassable singleton* designs do you prefer:?

Supposing we have a Singleton class A and a class B that is a sub-class of A:

1). You can have a single instance of A *OR* a single instance of B, but not both.

2). You can have exactly one instance of A *AND* exactly one instance of B.

TO DO: Can you implement and test one of these designs?

QUESTION: How can/should this design/code be extended to multiple subclasses?

Decorator Pattern

See - http://sourcemaking.com/design_patterns/decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client-specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.
- Problem
  - You want to add behavior or state to individual objects at run-time.
  - Inheritance is not feasible because it is static and applies to an entire class.

Decorator Pattern

See - http://sourcemaking.com/design_patterns/decorator

Relation to other patterns

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

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

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

A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.

Decorator Pattern

See - http://sourcemaking.com/design_patterns/decorator

Relation to other patterns

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

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

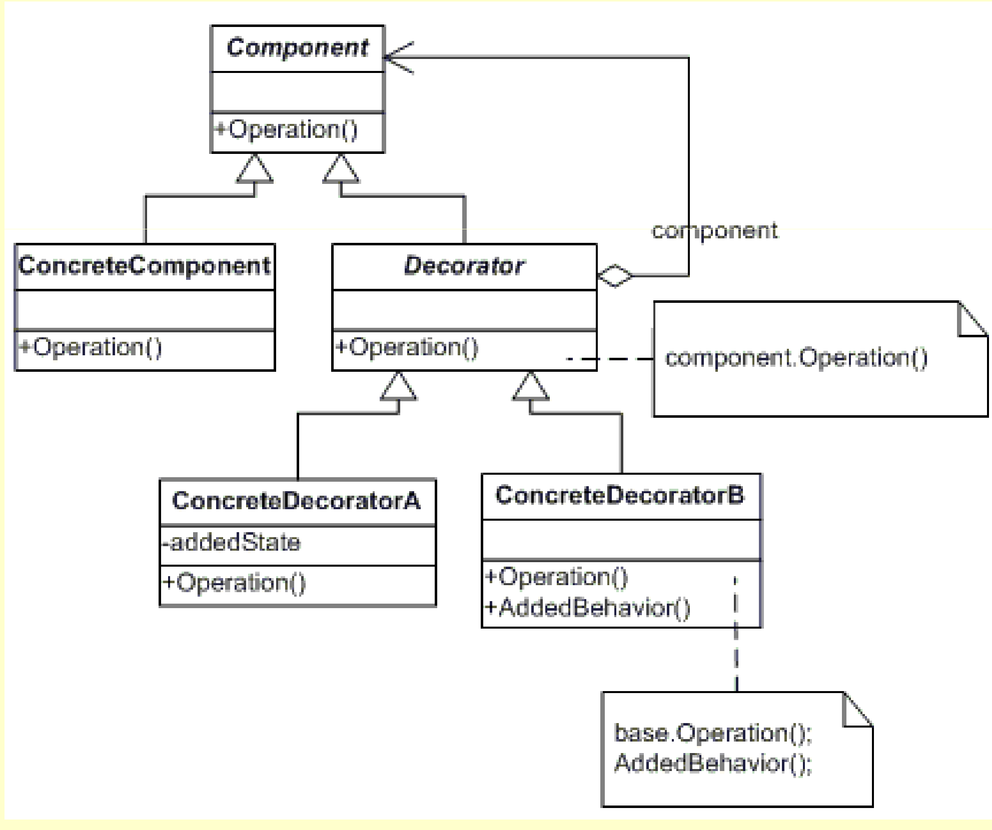
Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

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

Decorator Pattern

Class Diagram UML

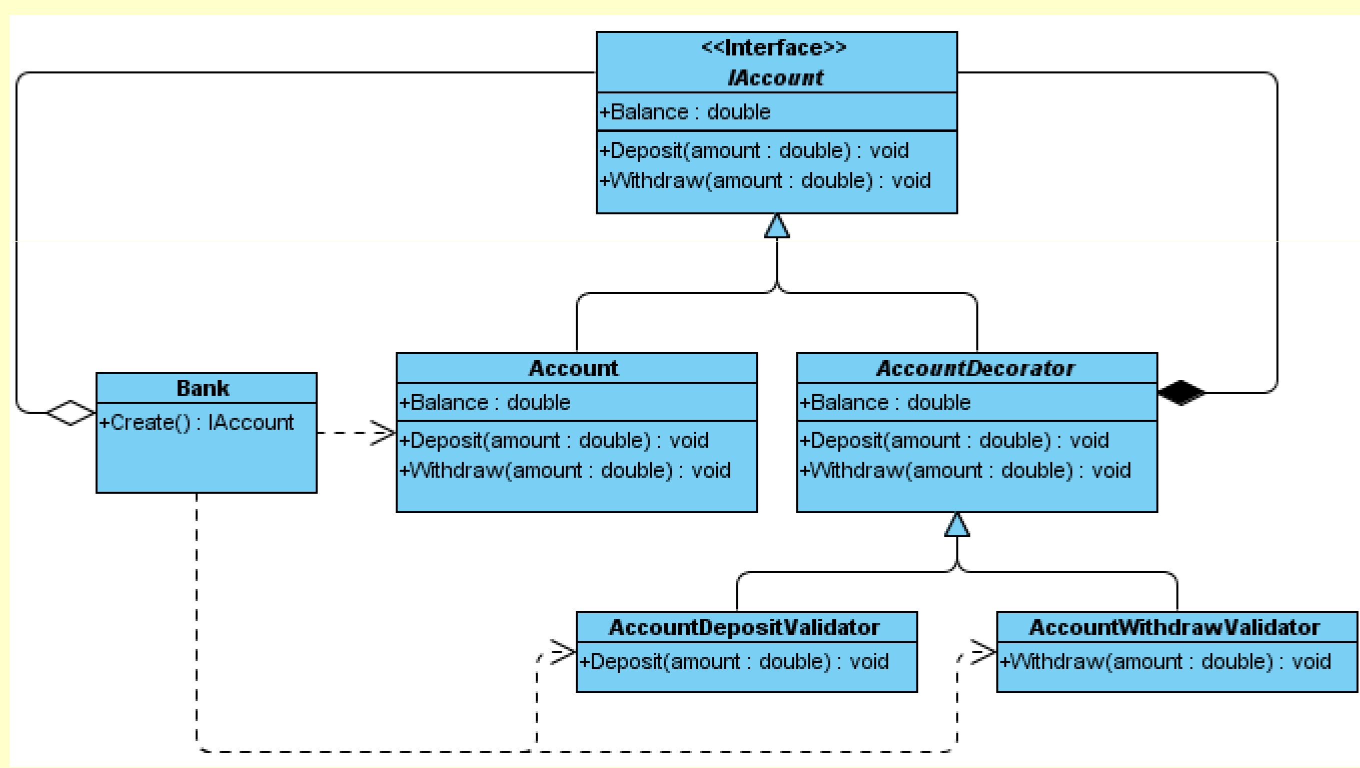
The following diagram shows structure of typical decorator pattern, the diagram was taken dofactory.com



Decorator Pattern

Class Diagram UML

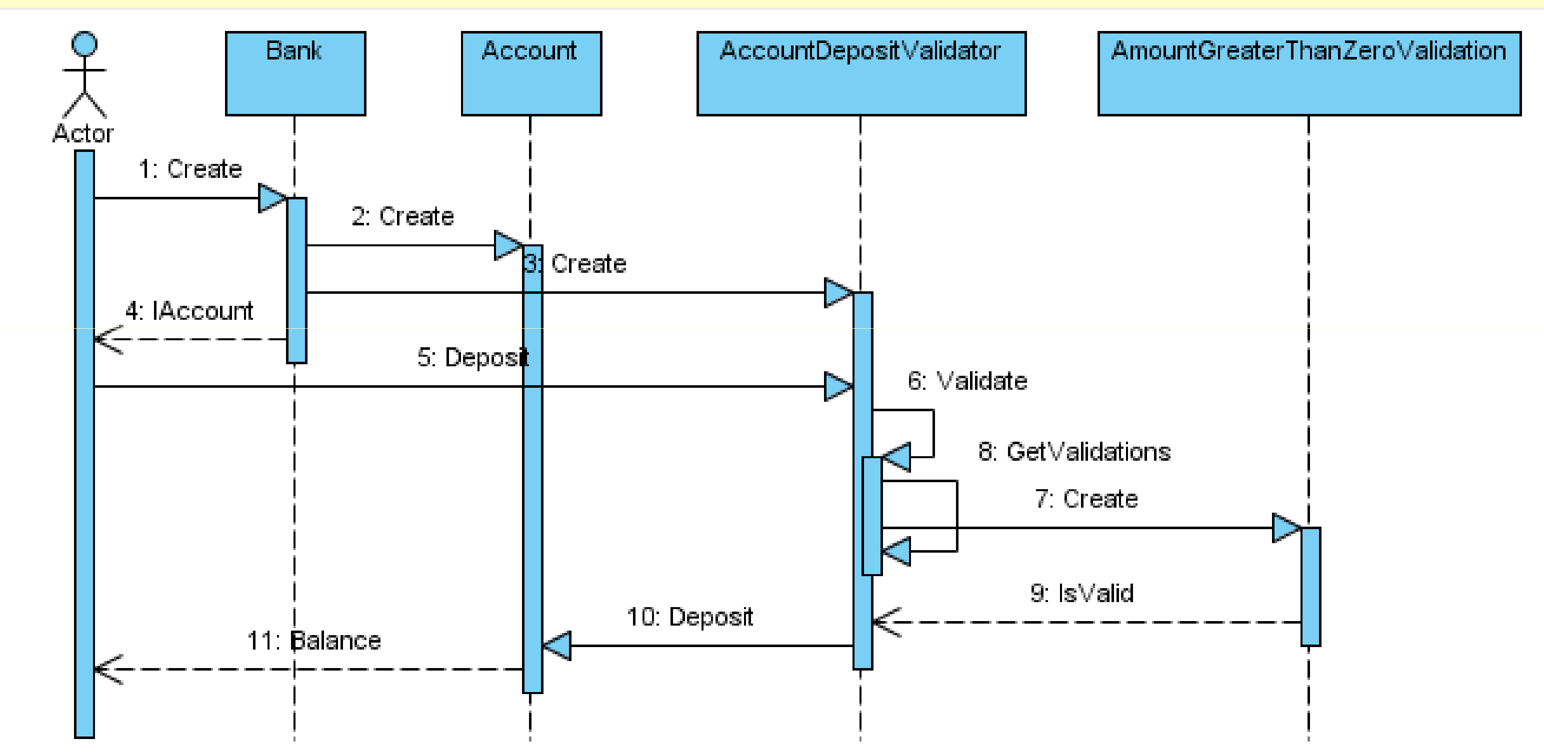
The following diagram shows a concrete example of an Account (from http://blog.decarufel.net/2009/09/using-decorator-or-wrapper-design.html)



Decorator Pattern

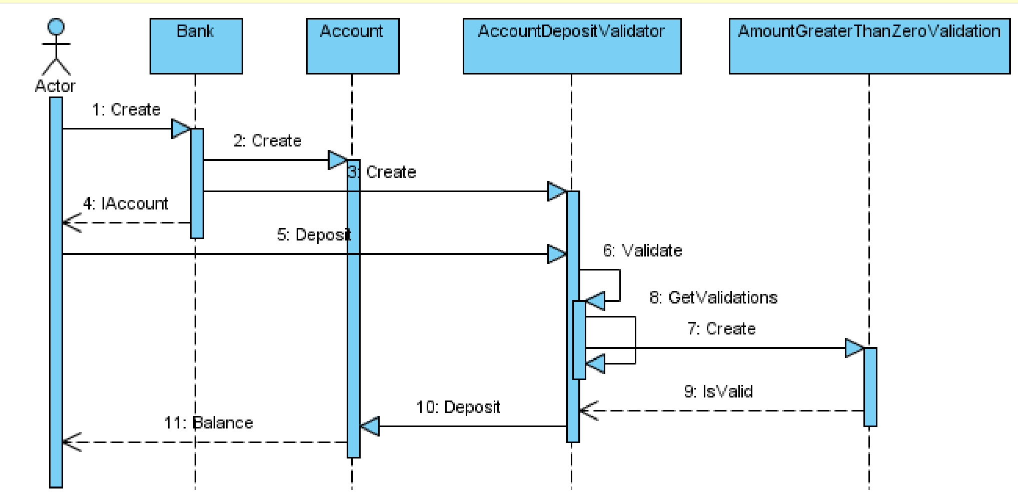
Sequence Diagram UML

The following diagram shows a concrete example of Account dynamic behaviour



Decorator Pattern

Sequence Diagram UML: explanation



1. Call Bank.CreateAccount.
2. The Bank instantiates an Account class.
3. The Bank creates an AccountDepositValidator and wrap Account with it
4. The Bank return an instance of IAccount.
5. Deposit is called on IAccount which is an instance of AccountDepositValidator
6. AccountDepositValidator calls Validate
7. AccountDepositValidator call GetValidations to retrieve the list of validation to evaluate
8. An AmountGreaterThaZeroValidation is created
9. IsValid returns true
10. AccountDepositValidator call base Deposit method
11. Balance value is updated

Decorator Problem : The classic Christmas tree (revisited)

Problem: Examine the Java code in the package p_decorator in the folder Patterns (http://www-public.int-evry.fr/~gibson/Teaching/CSC7322/Code/Patterns.zip)

- p_decorator
  - models
    - BallDecorator.java
    - LightsDecorator.java
    - UndecoratedBranch.java
  - specifications
    - BranchDecoratorSpecification.java
    - BranchOfTreeSpecification.java
  - tests
    - TestBallDecorator.java
- p_factory
- p_Iterator
- p_MVC
- p_singleton
- p_visitor

Decorator – Christmas Tree

```
package specifications;

public interface
BranchOfTreeSpecification {

public void animate();
public String getDecorations();

}
```

Decorator – Christmas Tree

```
package specifications;

public abstract class BranchDecoratorSpecifiation implements
BranchOfTreeSpecification{

protected BranchOfTreeSpecification decoratedBranch;

public BranchDecoratorSpecifiation(BranchOfTreeSpecification
branchToDecorate){
decoratedBranch= branchToDecorate;
}

public void animate(){decoratedBranch.animate(); }



}
```

Decorator - Christmas Tree

```
package models;

import specifications.BranchOfTreeSpecification;

public class UndecoratedBranch implements
BranchOfTreeSpecification{

public void animate(){}

public String getDecorations(){
return "";
}

}
```

Decorator - Christmas Tree

```
package models;
import specifications.BranchDecoratorSpecifiation;
import specifications.BranchOfTreeSpecification;

public class BallDecorator extends BranchDecoratorSpecifiation{

boolean spinning;

         public BallDecorator(BranchOfTreeSpecification branchOfTree){
         super(branchOfTree);spinning = true;
         }

        public String getDecorations(){
        String str = decoratedBranch.getDecorations();
        if (spinning) str = str +" Spinning";
        str = str+ " Ball.";
        return str;
         }

        public void animate(){
        decoratedBranch.animate();
        spinning = !spinning;
        }
}
```

Decorator– Christmas Tree

```
package models;
import specifications.BranchDecoratorSpecifiation;
import specifications.BranchOfTreeSpecification;
public class LightsDecorator extends BranchDecoratorSpecifiation{

String colour;public LightsDecorator(BranchOfTreeSpecification branchOfTree){

         super(branchOfTree);
         colour =  "red";
         }

        public String getDecorations(){
        String str = decoratedBranch.getDecorations();
        str =str+ " colour = "+colour;
         return str;
         }

        public void animate(){
        decoratedBranch.animate();
        if (colour.equals("red")) colour ="white";
        else if (colour.equals("white")) colour ="blue";
        else colour = "red";
        }
}
```

Decorator– Christmas Tree

```
public class TestBallDecorator {
public static void main (String [] args){

BranchOfTreeSpecification plainBranch = new UndecoratedBranch();
BranchOfTreeSpecification decoratedBranch = new BallDecorator( plainBranch);
BranchOfTreeSpecification reDecoratedBranch = new BallDecorator( new BallDecorator(
plainBranch));

System.out.println("plainBranch.getDecorations() ="+plainBranch.getDecorations());
System.out.println("decoratedBranch.getDecorations() ="+decoratedBranch.getDecorations());
System.out.println("reDecoratedBranch.getDecorations() ="
            +reDecoratedBranch.getDecorations());

plainBranch.animate();
System.out.println("plainBranch.animate.getDecorations() ="+plainBranch.getDecorations());

decoratedBranch.animate();
System.out.println("decoratedBranch.animate.getDecorations()
="+decoratedBranch.getDecorations());

reDecoratedBranch.animate();
System.out.println("reDecoratedBranch.animate.getDecorations() ="
            +reDecoratedBranch.getDecorations());
}
}
```

Decorator - Christmas Tree

Javadoc | Declaration | Console

<terminated> TestBallDecorator [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (4 janv. 2011 06:30:19)

```
plainBranch.getDecorations() =
decoratedBranch.getDecorations() = Spinning Ball.
reDecoratedBranch.getDecorations() = Spinning Ball. Spinning Ball.
plainBranch.animate.getDecorations() =
decoratedBranch.animate.getDecorations() = Ball.
reDecoratedBranch.animate.getDecorations() = Ball. Ball.
```

Decorator – Christmas Tree

TO DO:

- Test Lights Decoration
- Test Lights with Balls Decoration
- Add new Decoration (Stars, eg)
- Test 3 Decorations together

Create XMAS tree with 4 branches:

- 2 Lights and 2 Balls
- 2 Lights and 3 Stars
- 2 Stars and 3 Balls
- 2 Lights and 2 Stars and 2 Balls

PROBLEM: Ensure that the state of decorations on the same branch of tree is co-ordinated (but not necessarily on different branches)

OPTION: Maximum number of decorations on any branch is 6

Decorator– Christmas Tree

What to learn from the problem:

If you really need multiple instances of same decoration then this pattern is probably not correct.

The following 4 sentences seem equivalent but may lead to different designs/implementations:

The branch « is a » branch with lights and stars
The branch « is a » branch with lights and « is a » branch with stars

The branch « has » lights and stars
The branch « has » lights and the branch « has » stars

Knowing which design is best depends on knowing precisely what is the intended/required behaviour.

The decorator is not intended to allow statements like:
The branch « is a » branch with lights and « is a » branch with lights

Decorator – Christmas Tree

Solving the problem:

If we really need multiple decorations of the same type on a single branch do not use the decorator pattern in this way

We should fix the decorator pattern so that decorating a branch with ball behaviour that has already been decorated with ball behaviour is either:

- Handled by an exception, or
- Ignored, or

- *Advanced - Find another solution (based on another design pattern, perhaps?)*

TO DO – implement one of these solutions, and retest the new behaviour

NOTE: As requirements change you may be tempted to compromise your system by holding on to a pattern which is no longer appropriate

Factory Pattern

See - http://sourcemaking.com/design_patterns/factory_method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - Defining a “virtual” constructor.

- Problem
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

NOTE: The implementation of Factory Method discussed in the largely overlaps with that of Abstract Factory.

Factory Pattern

See - http://sourcemaking.com/design_patterns/factory_method

Relation to other patterns

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

Factory Methods are usually called within Template Methods.

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

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

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

The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.

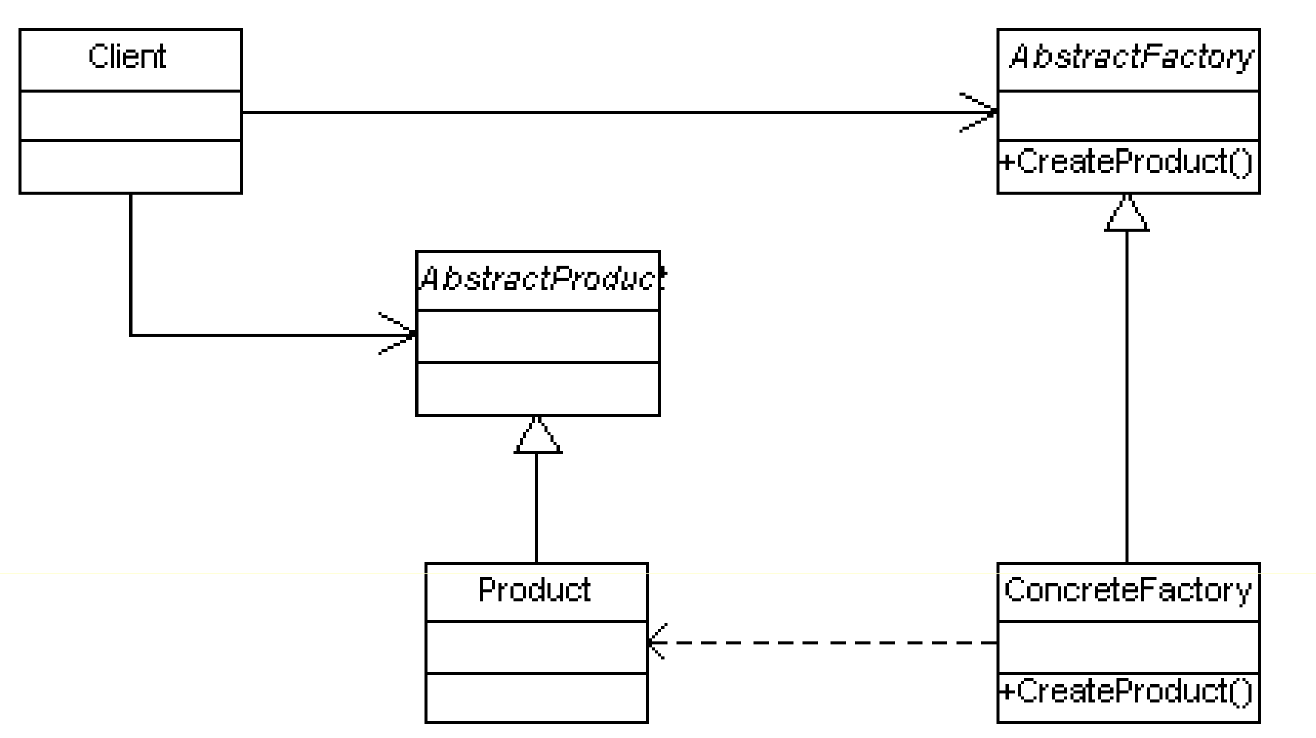
Factories: Additional Motivation

Factories are key to ***Software Product Lines***

See:

S*oftware Factories Assembling Applications with Patterns, Models, Frameworks and Tools*, Greenfield and Short, OOPSLA, 2003.

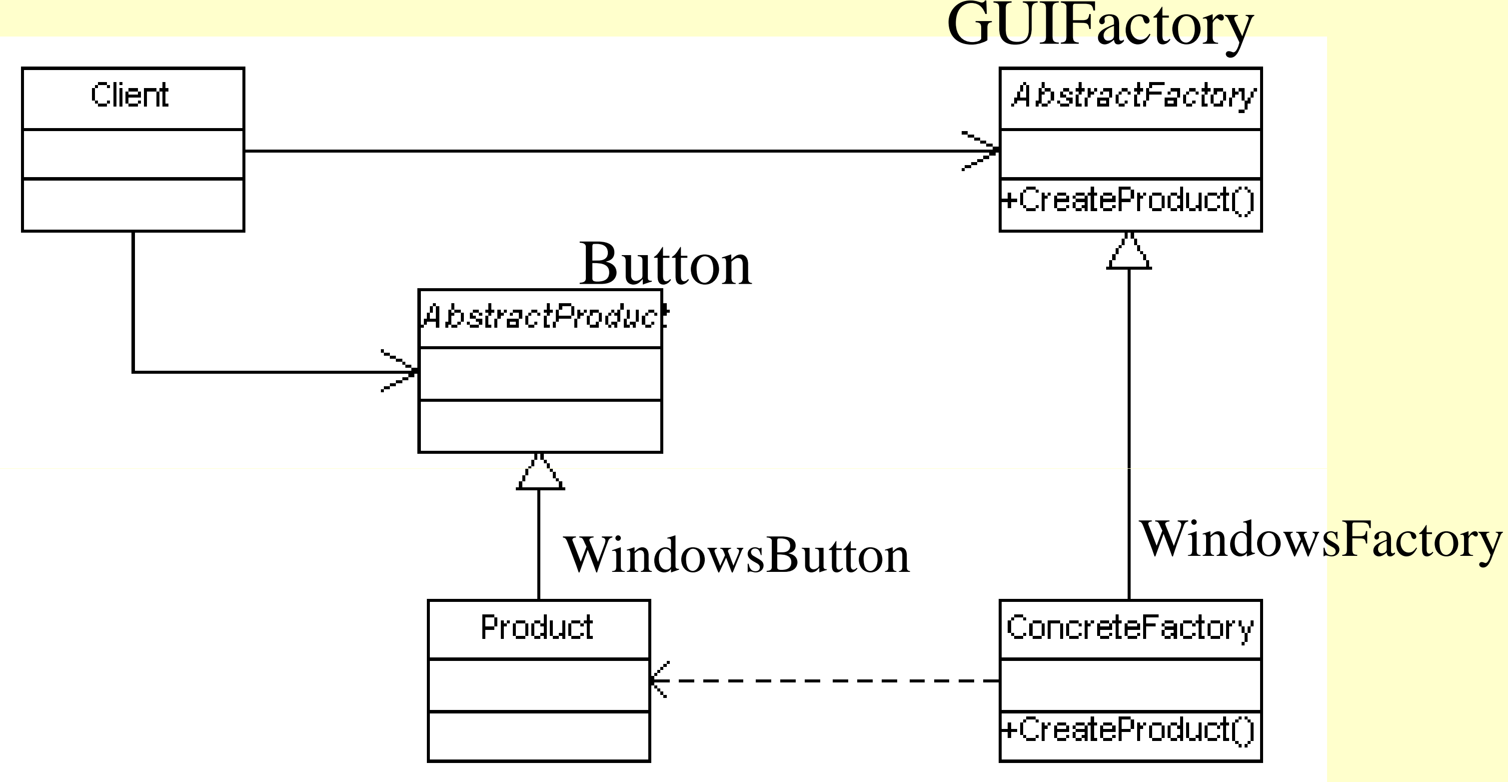
Patron: Factory (Fabrique): UML (generic)



Can be generalised to:

- multiple products (by subclassing)
- multiple clients (by association)

Factory UML: concrete example – WindowsButtonFactory



You can find the files in the Patterns folder in the **p_factory** package

Factory package

- p_factory
  - Button.java
  - GUIFactory.java
  - GUIFactory2.java
  - GUIFactoryChoice.java
  - GUIFactoryChoice2.java
  - Menu.java
  - OSXButton.java
  - OSXFactory.java
  - OSXFactory2.java
  - OSXMenu.java
  - OSXorWindowsFactory.java
  - OSXorWindowsFactory2.java
  - WindowsButton.java
  - WindowsButtonFactory.java
  - WindowsFactory.java
  - WindowsFactory2.java
  - WindowsMenu.java

Factory - Windows GUI in Java

p_factory.WindowsButtonFactory

Version:

1 Test for simplest factory behaviour:

- Make a Windows Factory and print identifier to screen
- Make a button using this factory
- Make a second Windows Factory and print identifier to screen (it should be the same as the first)
- Make a second button using this factory
- Write state of buttons to screen

EXPECTED (TYPICAL) OUTPUT

```
Using factory p_factory.WindowsFactory@9304b1 to construct aButton
WindowsButton: Push a
Using factory p_factory.WindowsFactory@9304b1 to construct bButton
WindowsButton: Push b
```

Author:

J Paul Gibson

Factory - Windows GUI in Java

```
public class WindowsButtonFactory {

    public static void main(String[] args){

        GUIFactory aFactory = GUIFactory.getFactory();
        System.out.println("Using factory "+ aFactory+" to construct aButton");
        Button aButton = aFactory.createButton();
        aButton.setCaption("Push a");
        aButton.paint();

        GUIFactory bFactory = GUIFactory.getFactory();
        System.out.println("Using factory "+ bFactory+" to construct bButton");
        Button bButton = bFactory.createButton();
        bButton.setCaption("Push b");
        bButton.paint();
    }

}
```

TO DO: Compile and execute to test for expected output

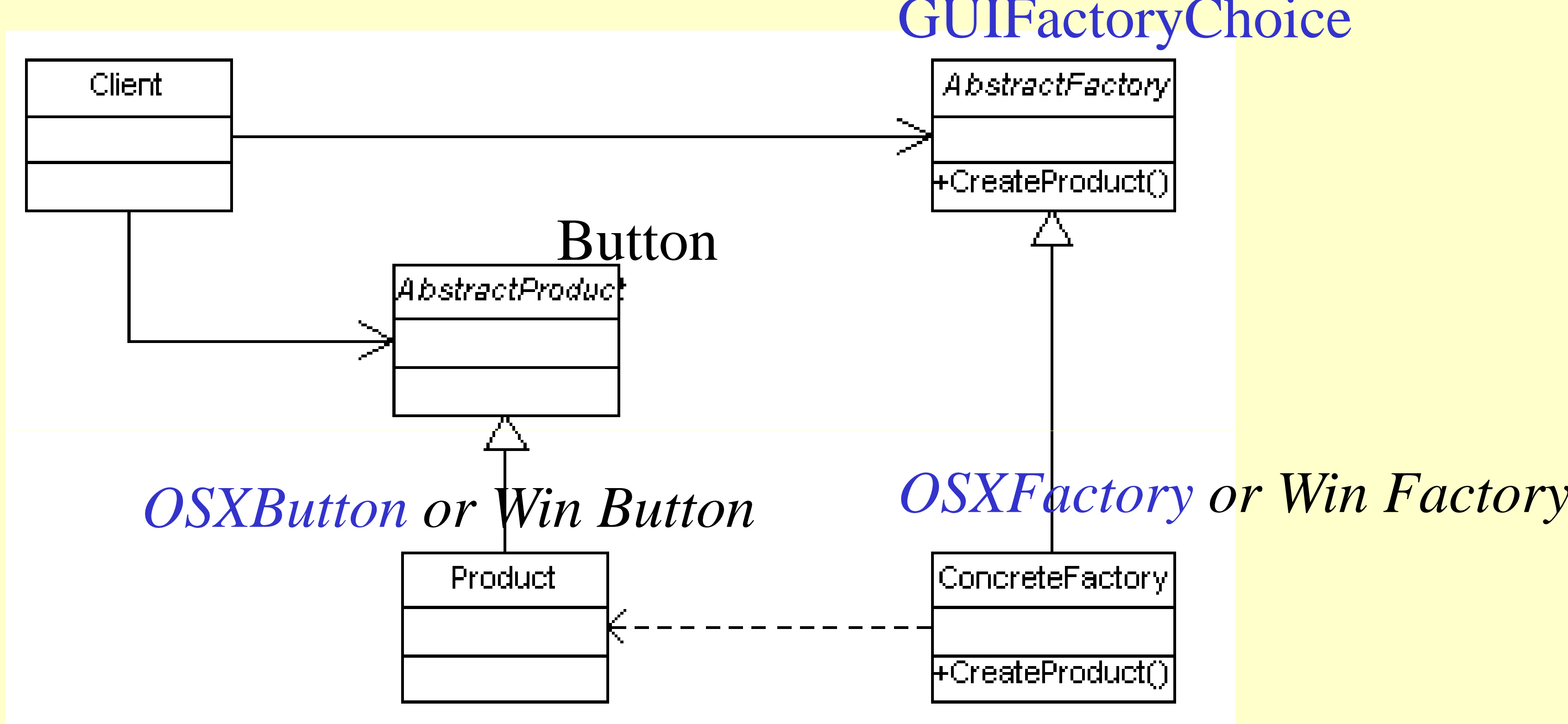
Factory - Windows GUI in Java

```java
abstract class Button

{

    private String caption;

    public abstract void paint();

    public String getCaption() {return caption;}


    public void setCaption(String caption){

        this.caption = caption;

    }

}

public class WindowsButton extends Button
{
    public void paint(){
        System.out.println("WindowsButton: "+ getCaption());
    }
}
```

Factory - Windows GUI in Java

```
abstract class GUIFactory{
    public static GUIFactory getFactory(){
        return WindowsFactory.getInstance();
    }
public abstract Button createButton();
}
class WindowsFactory extends GUIFactory{
private static WindowsFactory factory = new WindowsFactory();

    public static WindowsFactory getInstance () {return factory;};
    public Button createButton(){
        return(new WindowsButton());
    }
}
```

Factory « *UML* »: OSXorWindowsFactory



TO DO: Write code for OSXButton and OSXFactory

Factory OSX and Win GUI Buttons in Java

```
abstract class GUIFactoryChoice{
    public enum OS_Type {Win, OSX}

    protected static OS_Type readFromConfigFile(String param){
     if (Math.random() > 0.5) return OS_Type.Win;
        else return OS_Type.OSX;
    }

 public static GUIFactory getFactory(){
        OS_Type sys = readFromConfigFile("OS_TYPE");
        switch (sys) {
            case Win:
                return WindowsFactory.getInstance();
            case OSX:
                return  OSXFactory.getInstance();
        }
 throw new IllegalArgumentException("The OS type " + sys + " is not recognized.");
     }

    public abstract Button createButton();
}
```

Use this more complex factory in your test code

Factory OSX and Win GUI Buttons in Java

```
public class OSXorWindowsFactory {

 public static void main(String[] args){

        GUIFactory aFactory = GUIFactoryChoice.getFactory();
        System.out.println("Using factory "+ aFactory+" to construct aButton");
        Button aButton = aFactory.createButton();
        aButton.setCaption("Push a");
        aButton.paint();

        GUIFactory bFactory = GUIFactoryChoice.getFactory();
        System.out.println("Using factory "+ bFactory+" to construct bButton");
        Button bButton = bFactory.createButton();
        bButton.setCaption("Push b");
        bButton.paint();

        GUIFactory cFactory = GUIFactoryChoice.getFactory();
        System.out.println("Using factory "+ cFactory+" to construct cButton");
        Button cButton = cFactory.createButton();
        cButton.setCaption("Push c");
        cButton.paint();
    }

}
```

TO DO: Compile and execute this code

Factory OSX and Win GUI Buttons in Java

p_factory.OSXorWindowsFactory

Version:

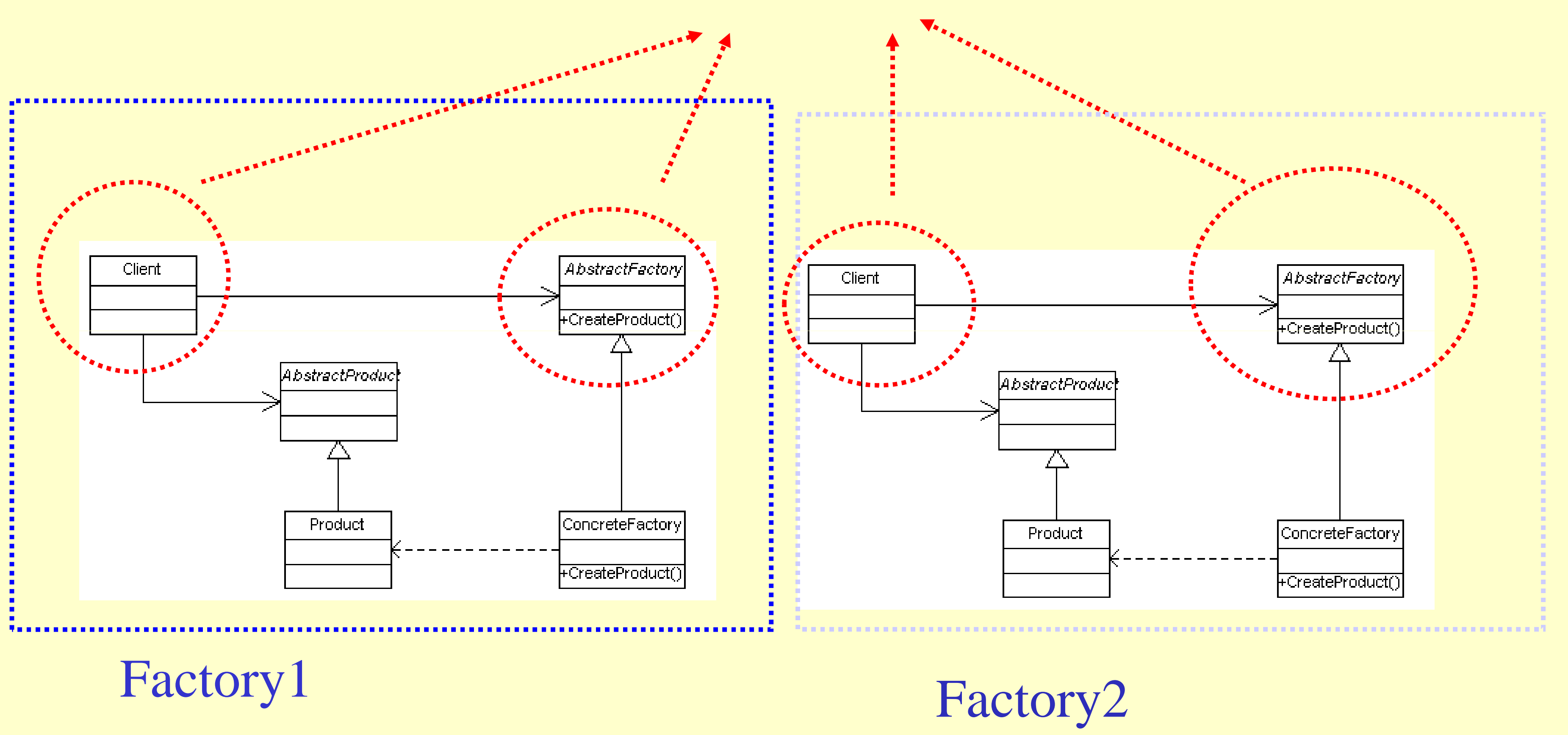
1 Check that different factories can be used but only 1 factory object of each type is ever created
EXPECTED (TYPICAL) OUTPUT

```
Using factory p_factory.WindowsFactory@1fb8ee3 to construct aButton
WindowsButton: Push a
Using factory p_factory.OSXFactory@14318bb to construct bButton
OSXButton: Push b
Using factory p_factory.WindowsFactory@1fb8ee3 to construct cButton
WindowsButton: Push c
```

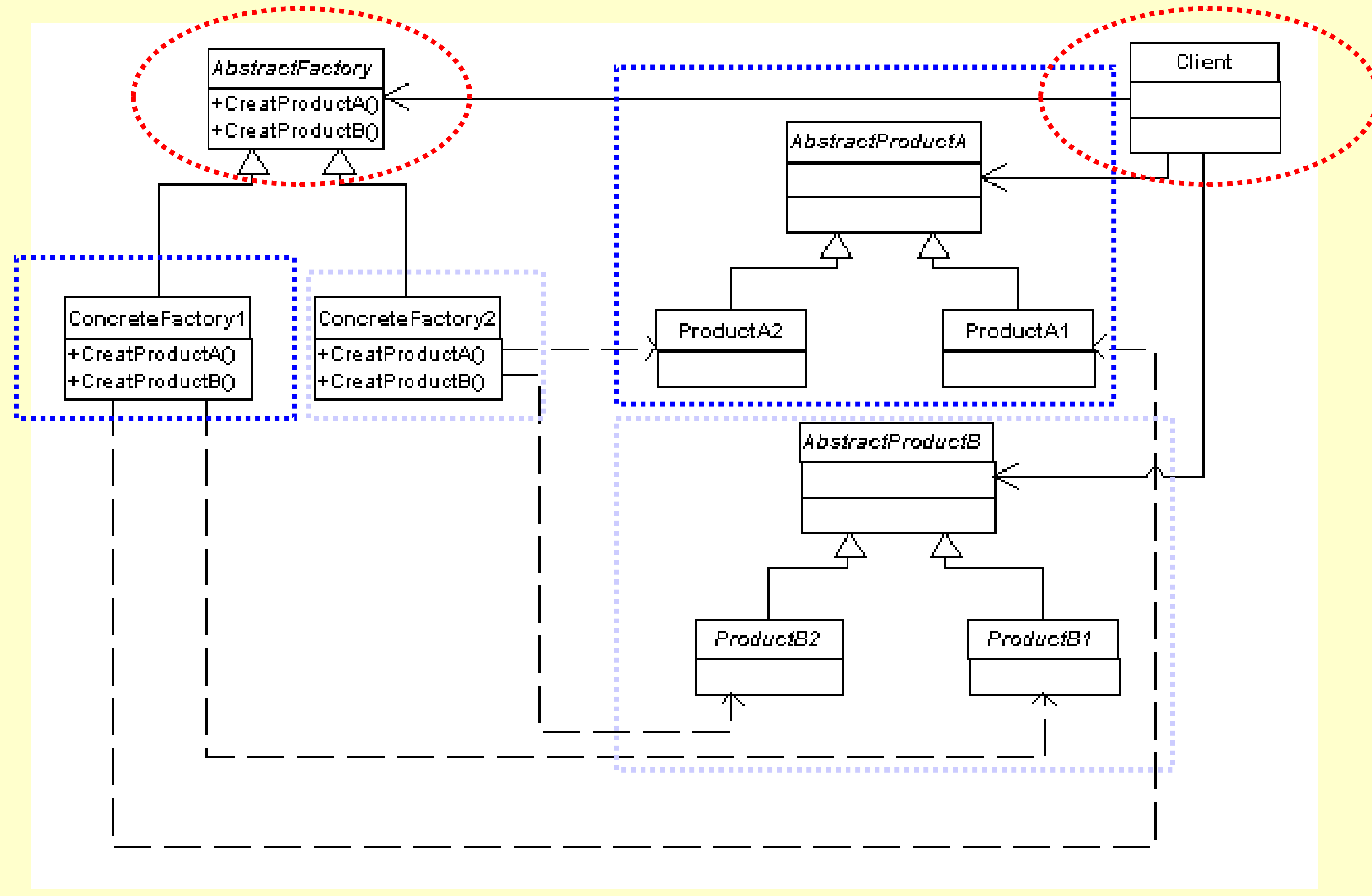
Author:

J Paul Gibson

Abstract Factory

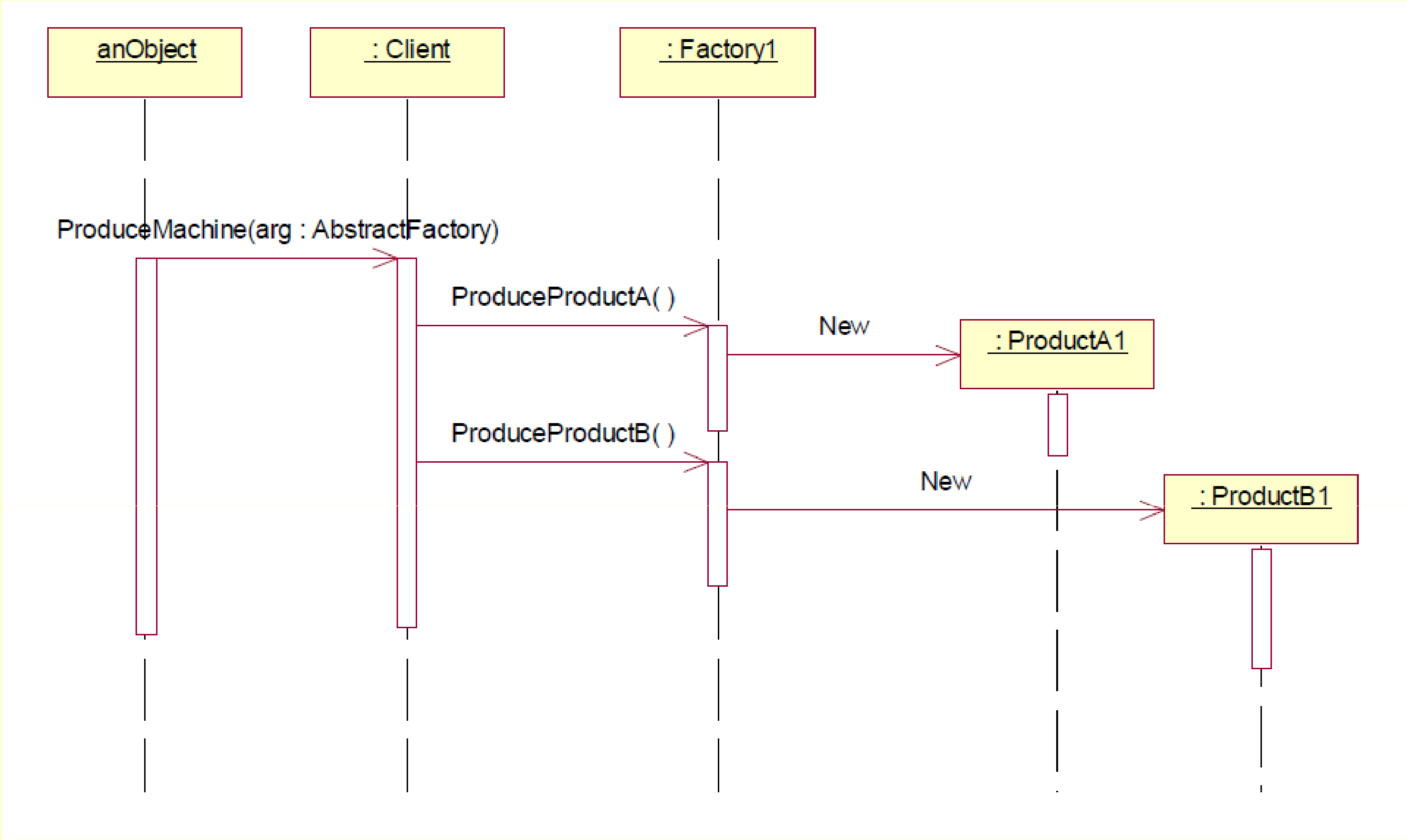


Abstract Factory: UML class diagram (2 products 2 factory types)



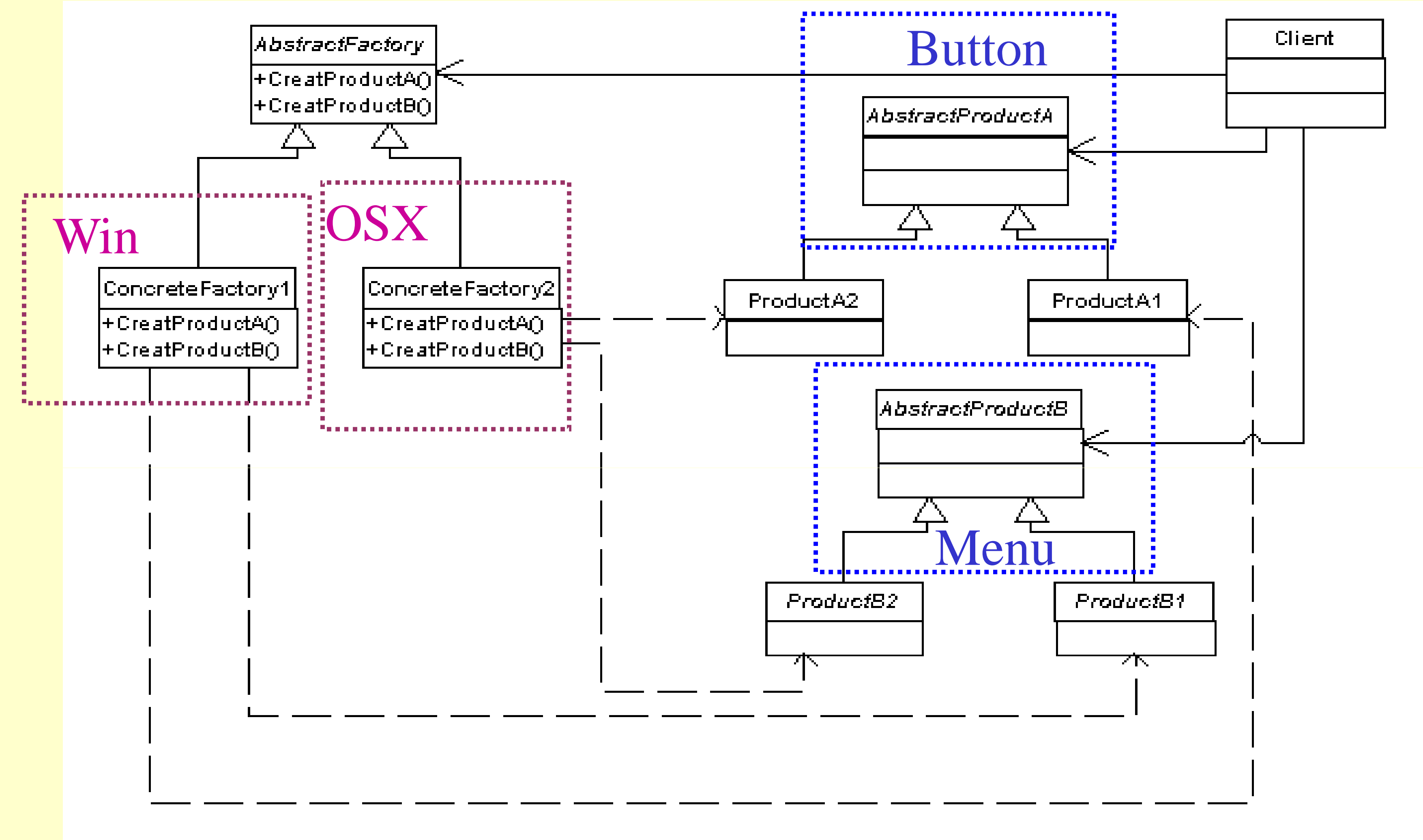
Can be generalised to multiple factories with multiple products

Abstract Factory: UML (sequence diagram)



Can be generalised to multiple factories with multiple products

Abstract Factory: UML - Buttons and Menus for Win and OSX



Abstract Factory – GUIFactoryChoice2

p_factory.OSXorWindowsFactory2

Version:

1 Check that different factories can be used but only 1 factory object of each type is ever created
Also check that we never mix component types (buttons and menus) in factories
EXPECTED (TYPICAL) OUTPUT

```
Using factory p_factory.OSXFactory2@1b67f74 to construct aButton
OSXButton: Push a
Using factory p_factory.OSXFactory2@1b67f74 to construct aMenu
OSXMenu: Menu a
Using factory p_factory.OSXFactory2@1b67f74 to construct bButton
OSXButton: Push b
Using factory p_factory.OSXFactory2@1b67f74 to construct bMenu
OSXMenu: Menu b
```

Author:

J Paul Gibson

TP - TO DO: Compile and execute this code in order to test it against expected behaviour

Abstract Factory – OSXorWindowsFactory2

```
public class OSXorWindowsFactory2 {
public static void main(String[] args){
        GUIFactory2 aFactory = GUIFactoryChoice2.getFactory();
        System.out.println("Using factory "+ aFactory+" to construct aButton");
        Button aButton = aFactory.createButton();
        aButton.setCaption("Push a");
        aButton.paint();
        System.out.println("Using factory "+ aFactory+" to construct aMenu");
        Menu aMenu = aFactory.createMenu();
        aMenu.setCaption("Menu a");
        aMenu.display();
        GUIFactory2 bFactory = GUIFactoryChoice2.getFactory();
        System.out.println("Using factory "+ bFactory+" to construct bButton");
        Button bButton = bFactory.createButton();
        bButton.setCaption("Push b");
        bButton.paint();
        System.out.println("Using factory "+ bFactory+" to construct bMenu");
        Menu bMenu = bFactory.createMenu();
        bMenu.setCaption("Menu b");
        bMenu.display();
    }
 }
```

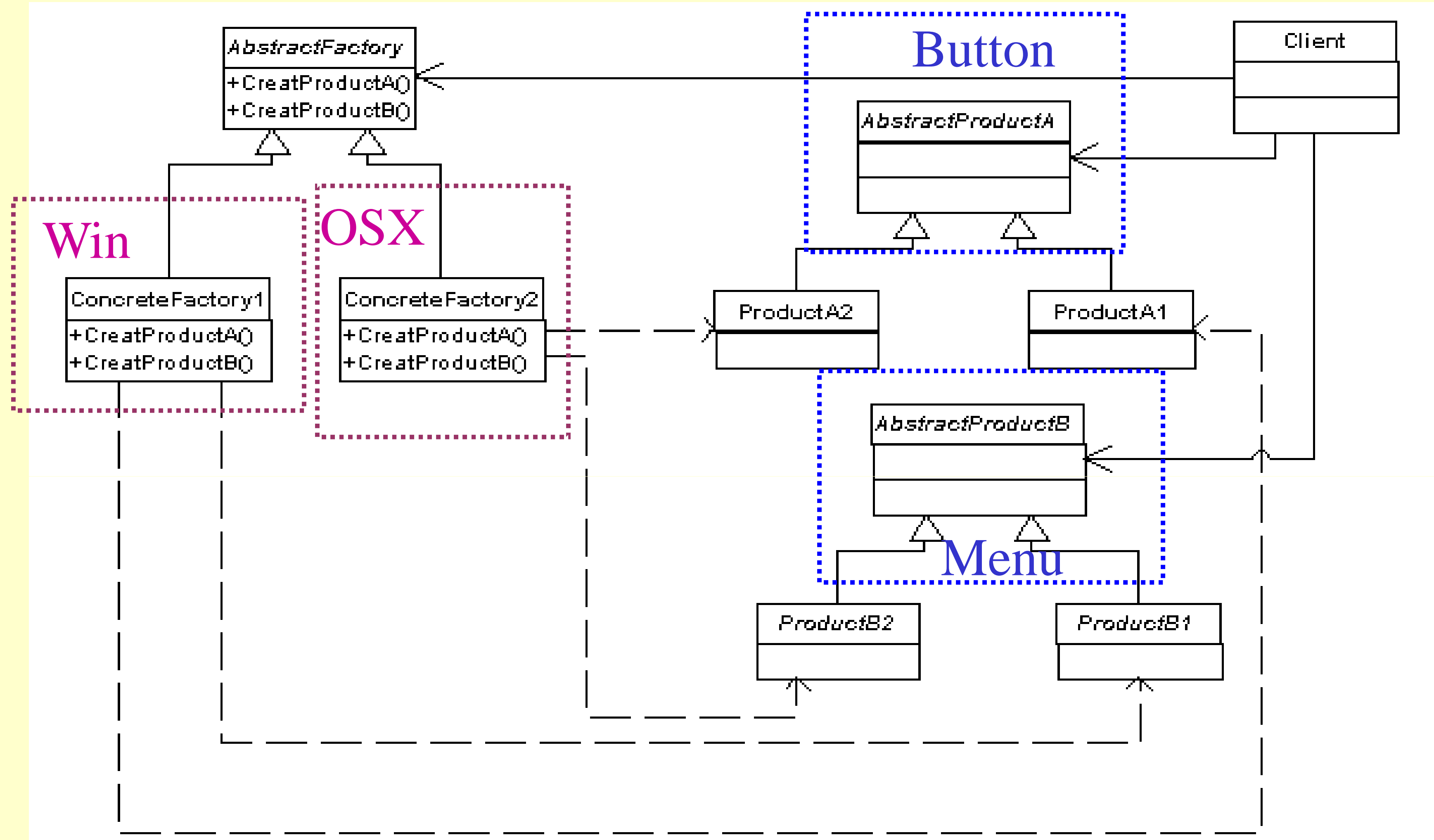
Abstract Factory – OSXorWindowsFactory2

Note that we had to extend the behaviour of classes in order to include buttons and menus (but we kept to the same design pattern):

```
public abstract class GUIFactory2 extends GUIFactory{
 public abstract Menu createMenu();
}

class WindowsFactory2 extends GUIFactory2 …

class OSXFactory2 extends GUIFactory2  …

class GUIFactoryChoice2 extends GUIFactoryChoice …
```

TO DO: Look at code and try to understand how it works

Problem – add an **OS (linux)** and a **Component (slider)**



Constuct a linux *product* with button and slider components: test the behaviour of your product (code)

Iterator

An **iterator** is an object that facilitates traversal of a container/collection of objects

Various types of iterators are often provided via a container's interface.

Though the interface and semantics of a given iterator are fixed, iterators are tightly coupled to the container implementation in order to enable the operational semantics of the iterator.

Note that an iterator performs traversal and also gives access to data elements in a container, but does not perform iteration.

QUESTION: Have you already seen this in Java?

Iterator Pattern

See - http://sourcemaking.com/design_patterns/iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
  - Promote to “full object status” the traversal of a collection.
  - Polymorphic traversal

- Problem

Need to “abstract” the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Iterator Pattern

See - http://sourcemaking.com/design_patterns/iterator

Relation to other patterns

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Iterator Pattern - Contrasting with indexing

Although indexing may also be used with some object-oriented containers, the use of iterators may have some advantages:

•Counting loops are not suitable to all data structures, in particular to data structures with no or slow random access

•Iterators make the code more readable, reusable, and less sensitive to a change in the data structure.

•An iterator can enforce additional restrictions on access, such as ensuring that elements can not be skipped or that a previously visited element can not be accessed a second time.

•An iterator may allow the container object to be modified without invalidating the iterator. For instance, once an iterator has advanced beyond the first element it may be possible to insert additional elements into the beginning of the container with predictable results. With indexing this is problematic since the index numbers must change.

Iterators and collections in Java

There are multiple ways to iterate a collection in Java

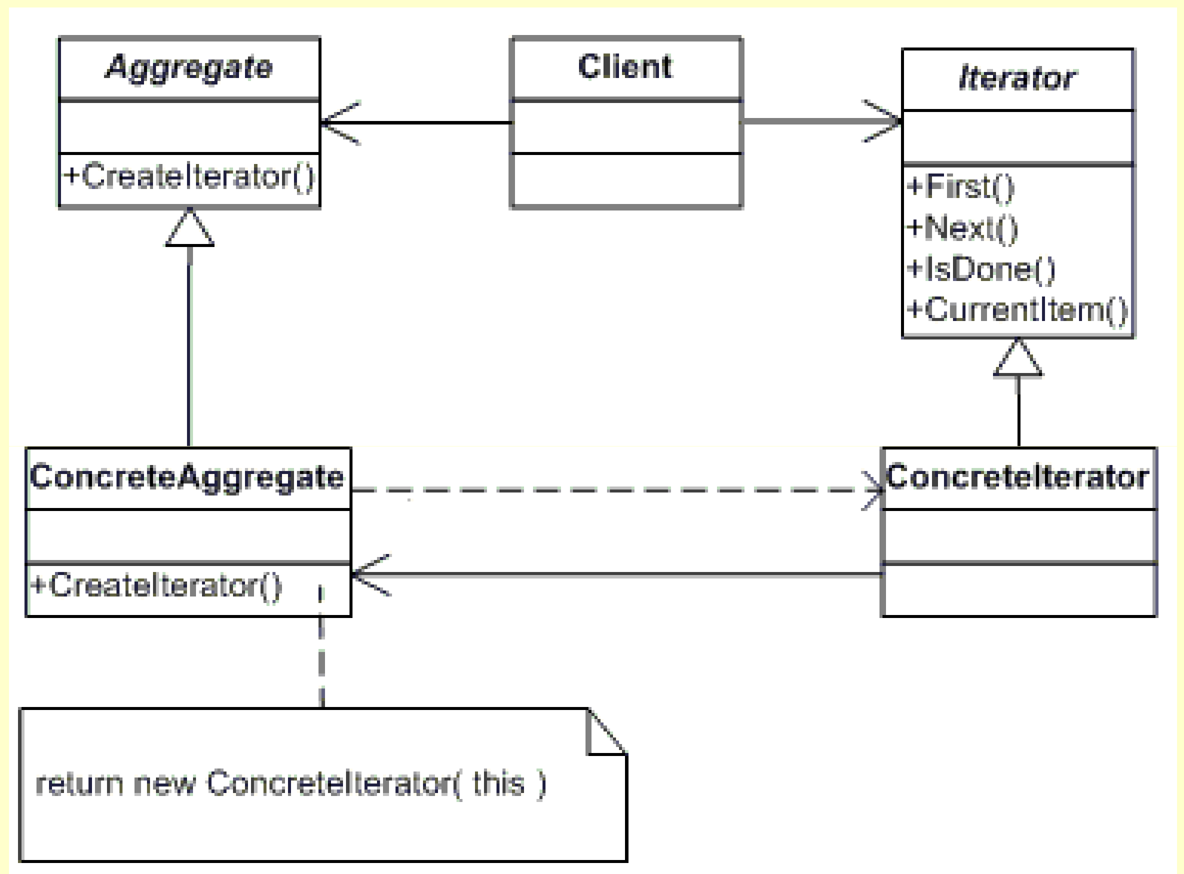
Example:

```
ArrayList persons = new ArrayList();

Person p= new Person("john", "mith");
persons.add(p);
...

Iterator i = persons.iterator();
```

```
while(i.hasNext())
{

    Person p= (Person)i.next();


    p.print();
}
```

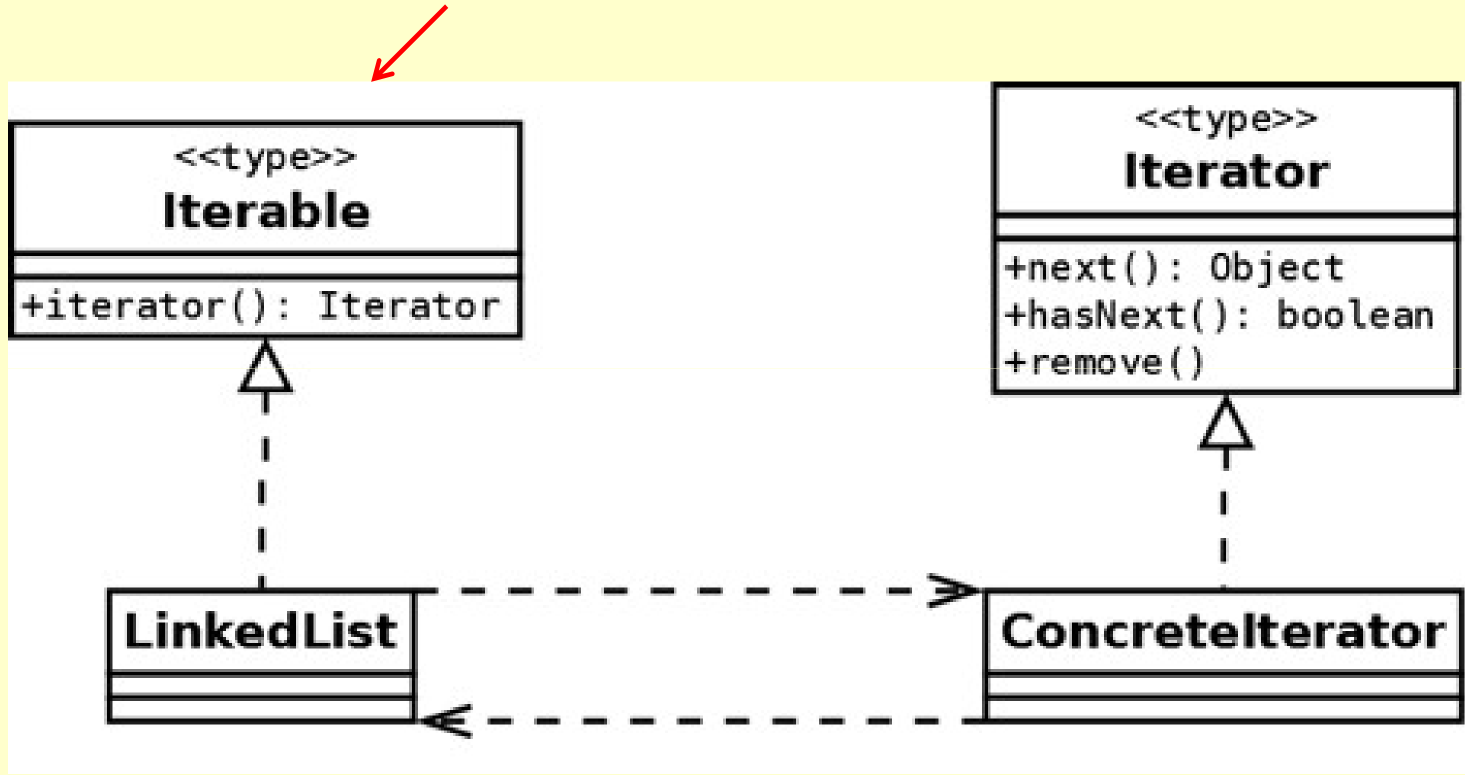
```
for(Person p : persons)
{
   p.print();
} // since java 1.5
```

Iterator in UML class diagram



Iterator in UML class diagram (linked list example in Java)

Notice that the **Aggregate** is an **Iterable** (in Java)



Java Iterable Example (in Patterns folder/package)

```
public class MyTableOfStrings implements
Iterable<String> {

protected String[] data;

public MyTableOfStrings(String [] data) {
this.data = data;
}

public int length(){return data.length;}

public Iterator<String> iterator() {
return new MyTableOfStrings_Iterator(this);
}

}
```

```
public class MyTableOfStrings_Iterator implements Iterator<String> {

private int index;
private MyTableOfStrings table;

public MyTableOfStrings_Iterator(MyTableOfStrings tab) {
index = tab.length()-1;
table = tab;
}

public String next() {
index--;
return table.data[index +1];
}

public boolean hasNext() {
return index >= 0;
}

public void remove() {
throw new UnsupportedOperationException();
}

}
```

```java
public class MyTableOfStrings_Test {

public static void main(String[] s) {

String [] data = {"one", "two", "three"};
MyTableOfStrings t = new MyTableOfStrings(data);

System.out.println("Iterate over original data array");
for (String value : data) {
System.out.println(" "+value);
}
System.out.println("\nIterate over same data in MyTableOfStrings");
for (String value : t) {
System.out.println(" "+value);
}
}

}
```

p_Iterator.MyTableOfStrings_Test

Version:

1 A simple test for the [MyTableOfStrings]

EXPECTED OUTPUT :

```
 Iterate over original data array
  one
  two
  three

 Iterate over same data in MyTableOfStrings
  three
  two
  one
```

Author:

J Paul Gibson

TO DO : Compile and execute the test class

Random Iteration: Reservoir Sampling

In the previous example we saw how the `Iterator` code decides the order in which to visit the elements.

By default Java iterates through arrays from the 1st to the last elements. In the example we iterate through `MyTableOfStrings` in reverse order.

TO DO:

Change the iterator code so that the elements are visited in random order. Do not do this by shuffling the elements as this may be expensive for a large number of elements.

A more complex data structure: an urn/ballot box of bulletins/votes

p_Iterator.Urn

Version:

1 An urn of votes, where each vote is a table of strings, eg:

```
[["gibson", "smith", "hughes"],
["jones", "bell"],
["raffy", "lallet"]]
```

represents three preferential votes with the first vote being -
first preference for gibson, second preference for smith and third preference for hughes

Author:

J Paul Gibson

TO DO: Your task is to iterate through the Strings in the Urn

Look at the `Urn_Test` Code and write the `Urn` and `Urn_Iterator` classes appropriately.

Check that the test, executed on your code, produces the expected results

```java
public class Urn_Test {

public static void main(String[] s) {

String [] preferences1 = {"gibson", "smyth", "hughes"};
MyTableOfStrings vote1 = new MyTableOfStrings( preferences1);

String [] preferences2 = {"jones", "bell"};
MyTableOfStrings vote2 = new MyTableOfStrings( preferences2);

String [] preferences3 = {"raffy", "lallet"};
MyTableOfStrings vote3 = new MyTableOfStrings( preferences3);

MyTableOfStrings [] votes = { vote1, vote2, vote3};

Urn urn = new Urn (votes);

System.out.println("\nIterate over strings on bulletins in Urn");
for (String value : urn) {System.out.println(" "+value);}
}
}
```

p_Iterator.Urn_Test

Version:

1 A simple test for the `Urn`

EXPECTED OUTPUT :

```
 Iterate over strings on bulletins in Urn
  hughes
  smyth
  gibson
  bell
  jones
  lallet
  raffy
```

Author:

J Paul Gibson

The Visitor Pattern

See - http://sourcemaking.com/design_patterns/visitor

- **Intent**
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
  - The classic technique for recovering lost type information.
  - Do the right thing based on the type of two objects.
  - Double dispatch

- **Problem**

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

The Visitor Pattern

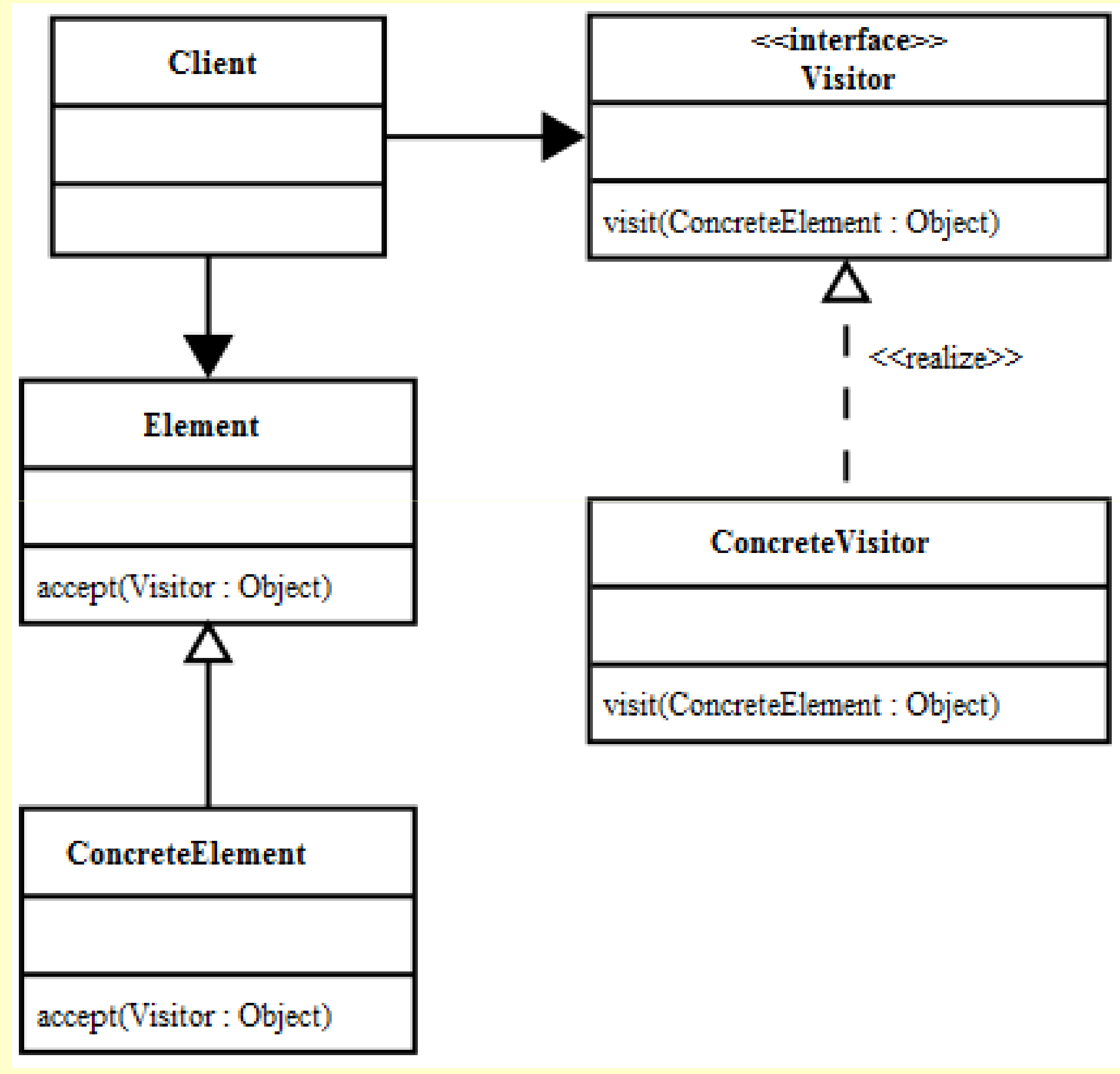
See - http://sourcemaking.com/design_patterns/visitor

Relation to other patterns

•The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
•Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
•The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.
•The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

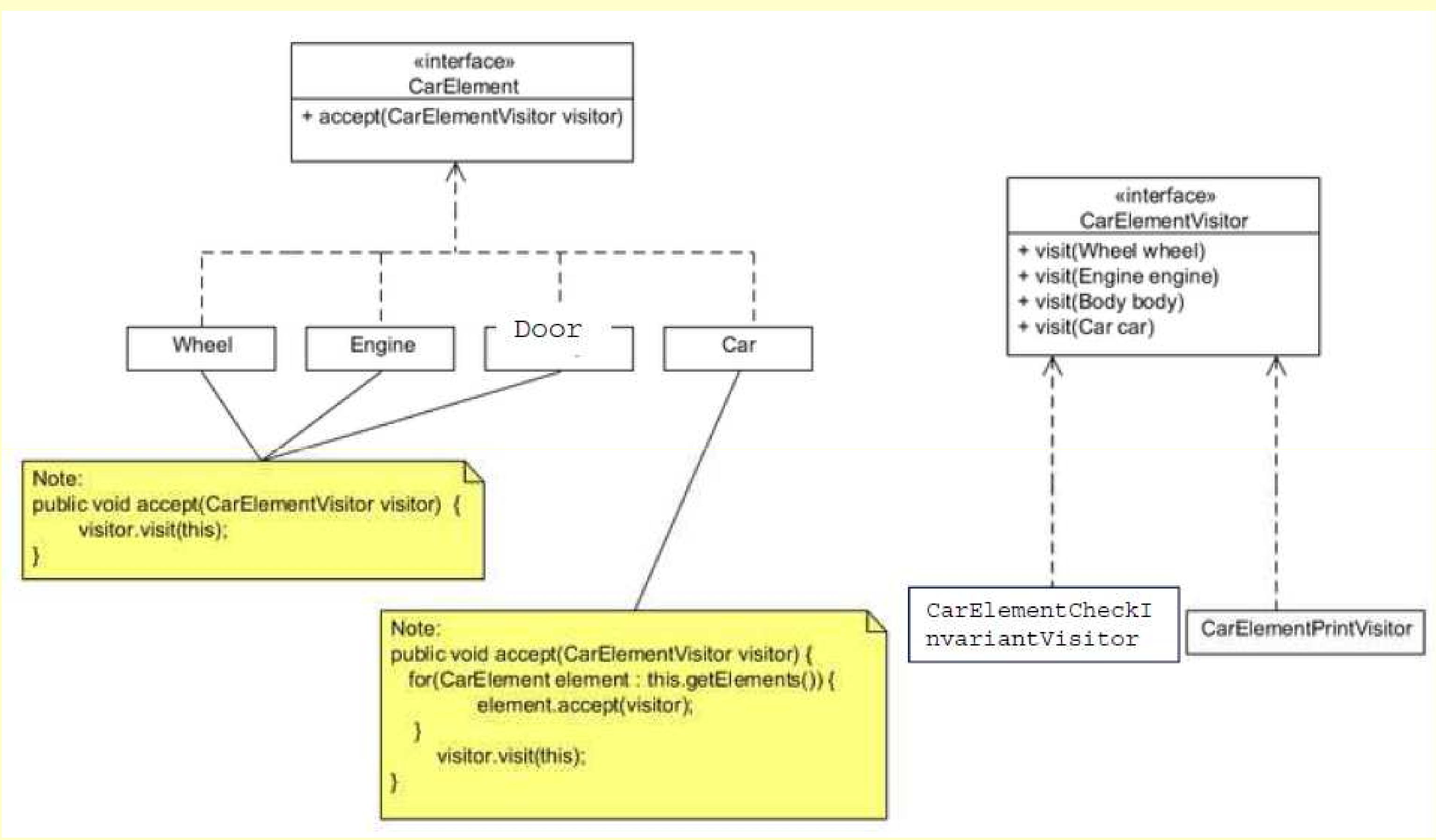
NOTE: Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended.

UML Class Diagram Visitor



QUESTION: why not define an abstract *visitable* class that can accept visitors?

UML Class Diagram Visitor Example - Car



An example *visitable*

```
class Car implements CarElement{
    CarElement[] elements;

    public CarElement[] getElements() {
        return elements.clone(); // Return a copy of the array of references.
    }

    public Car() {
        this.elements = new CarElement[]
          { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Doors(), new Engine(8) };
    }

    public String toString(){ return "\n  *** A Car *** \n"; }

    public boolean invariant (){ return (elements!=null && elements.length>0);}

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
        for(CarElement element : this.getElements()) {
            element.accept(visitor);
        }

    }
}
```

An example visitor

```
class CarElementPrintVisitor implements CarElementVisitor {

    public void visit(Wheel wheel) {
        System.out.println(wheel);
    }

    public void visit(Engine engine) {
        System.out.println(engine);
    }

    public void visit(Doors doors) {
        System.out.println(doors);
    }

    public void visit(Car car) {
        System.out.println(car);
    }
}
```

Testing the example

```
public class Visitor_Test {
    static public void main(String[] args){
        Car car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementCheckInvariantVisitor());
    }
}
```

```
*** A Car ***

front left is not turning
front right is not turning
back left is not turning
back right is not turning
LeftDoorLocked is true and RightDoorLocked is true
Engine speed is 0 / 8
```

QUESTION: what is **`CarElementCheckInvariantVisitor`** doing?

UML Class Diagram Visitor Example - Car

- p_visitor
  - Car.java
  - CarElement.java
    - CarElement
      - accept(CarElementVisitor) : void
  - CarElementCheckInvariantVisitor.java
  - CarElementPrintVisitor.java
  - CarElementVisitor.java
    - CarElementVisitor
      - visit(Car) : void
      - visit(Doors) : void
      - visit(Engine) : void
      - visit(Wheel) : void
  - Doors.java
    - Doors
  - Engine.java
  - Visitor_Test.java
  - Wheel.java

TO DO: Add a visitor (`breakInvariantVisitor`) which changes the state of each component of the car so that their invariants are broken.

Update the test class to check that this visitor is working as required