Design Patterns for Java

University of California, Irvine
University Extension

Barry Geipel

I&CSCI X460.26
Summer Quarter 2001

Welcome!

Design Patterns for Java

- Your Instructor:
  - Barry L. Geipel
- Perquisites
  - Familiarity with Microsoft Windows 95/98 or NT 4.0
  - One year programming with a Java or completion of Java Programming I (ICS X460.10)
  - Have access to Symantec Visual Café for Java v4.0 to complete the assignments

Welcome!

Course Structure

- 5 Meetings:
  - Thursdays, August 23-September 20, 5:30-8:30pm;
- Meeting Format:
  - First half of each class is lecture
  - Second half of each class is lab
    - Time devoted to specific exercises to demonstrate lecture topics
    - Time devoted to assignments
- Breaks:
  - A 15 minute break between lecture and lab

Welcome!

Course Outline

- Course introduction, Design Patterns overview, Java Review, UML overview
- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- Design Pattern Uses in Java

Welcome!

Syllabus

- Assignments, Labs and Final
  - Two assignments
  - A final exam
  - Labs are not graded, but are designed to help with understanding the material. There should be extra lab time to work on assignments and to ask me questions.
- Grading
  - Straight forward 90,80,70 etc...

Welcome!

Syllabus

- Textbooks
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Hamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Pub. Co., 395 Pages, ISBN 0-201-63361-2
  - *UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language* (Martin Fowler, Addison-Wesley Pub. Co., 185 Pages, ISBN: 0-201-65783X).
- Additional Recommended Reading:
  - *Java Design Patterns: a Tutorial* (James W. Cooper, Addison-Wesley Pub. Co., 329 Pages, ISBN 0-201-48539-7)

Welcome!

Syllabus

- Extra Credit
  - Various extra credit opportunities at my discretion
- Tools
  - In lab we will be using:
    - VisualCafe 4.0 for Java Development
    - Telelogic Tau UML Suite for UML diagrams
  - You should be able to do all of your UML in lab so you will not need another tool for home.
  - You will need time outside of class to complete your programming assignments so you will need a Java Development environment at home.

Welcome!

Syllabus

- Homework Assignments
  - You will have two homework assignments.
  - You must turn in:
    - A printout of all of your source code.
    - A printout of your UML Diagrams
      - Your UML diagrams may be in any printable form. You can use Tau UML in lab, any other drawing tool or even hand drawn diagrams.
    - An appropriately formatted floppy disk containing your Java source and compiled class files.

Welcome!

Expectations

- What I expect you will get from this class
  - A better understanding of Object Oriented Design using Reusable Design Patterns
  - A high level of enthusiasm for the art of software design.
- What I expect to get out of this class
  - Increase my expertise, communication and instructing skills
- Question escalation
  - Answer on-line/during class if possible
  - Answer off-line/after class if possible
  - Research or direct students to resource

Welcome!

Resources

- The web site for the class is:
  - http://www.GeipelNet.com/djp/
- My email address is:
  - barry@GeipelNet.com
- Other contact info:
  - Yahoo messenger: b_geipel
- Valuable web sites:
  - http://java.sun.com

Overview of Design Patterns

What are Design Patterns?

- Design Patterns “describe simple and elegant solutions to specific problems in object oriented design”
- The basic design patterns have grown out of years of mistakes and recoding. After a while, every developer begins to reuse patterns that they have developed.
- Upon close examination, there are a basic set of Design Patterns that are seen again and again. These form the basis of the Design Patterns Catalog.

Overview of Design Patterns

What are Design Patterns?

- Design Patterns are a GREAT way to study object oriented design.
- Reading the description of a pattern and understanding the relationships between classes will greatly increase your understanding of what it means to design an object oriented system.

Overview of Design Patterns

What Design Patterns are not

- Design Patterns are not functions, libraries or class libraries, although you may find some generic implementations of some of the Patterns.
- Design Patterns are not cast in stone. There are implementation specific improvements which you can and should use.
- Design Patterns are not language specific. The GOF book has examples in C++ and Small Talk, but they translate well into any object oriented language.

Overview of Design Patterns

The GOF Book

- The Design Patterns book was written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – collectively know as the “Gang of Four”
- You will often see the book referred to as “GOF”
- It is divided up into a brief introductory chapter, a design case study and then the Design Patterns Catalog.
- The catalog is broken up into three sections
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- We will talk about the catalog in more detail next week.

Java Programming Overview

Java is an Object Oriented Language

- Variables
  - Variables are data members of a class. Variables may be either primitive types or may be instances of classes (Objects)
  - Class Variables – variables that are static and shared between all instances of a class
  - Instance Variables – variables who's contents are unique to one instance of a class. Instance variables define the "state" of an object.
- Methods
  - Methods define the behavior of a class.
  - Methods contain the algorithms – code!
  - Similar to functions but methods operate only on an instance of a class.

Java Programming Overview

- Interface – A class' interface defines what methods and variables are exposed to other classes.
- Inheritance – A class can inherit the methods and variables of another class and use them as if they were all one class. In Java, a class can only inherit from one class (no multiple inheritance)
- Overriding – A class that has inherited methods from another class can implement it own version of any method from the parent class. This allows a class to have the same interfaces as the parent class but have different behavior

Java Programming Overview

- Overloading – Any method may have many implementations. In Java, the parameter list to a method is used to resolve which method to use.
- Polymorphism – many (poly-) forms (morph)
  - A class that inherits behavior from a parent class can be used as if it was the parent class. This allows an object oriented programmer to write code which can make use of classes that are not yet known.
  - Polymorphism is the basis for many features and design principles in the Java class hierarchy

Java Programming Overview

- Abstraction
  - A class is an abstraction of the object. The user of the object does not need to know anything about how the object is implemented
- Encapsulation
  - A class can hide data and behavior from the user by providing only key public methods to access data and behavior

Java Programming Overview

- Classes could represent:
  - A tangible grouping of data
  - A distinct component
  - A container
  - A representation of an abstract idea
  - A set of methods which have a common use

Interfaces

- An interface defines a collection of method declarations and constants
- Interfaces are used to establish a known set of methods without the need for inheritance.
- Method declarations only include the method signature. There is no code.
- May only use static variables
- Interfaces are not classes and may not be instantiated.
- Classes can implement interfaces and must implement all of the interfaces declared methods.

Interfaces

```
interface SetValueClass
{
    public void setValue(int i);
    public int getValue();
}
```

Interfaces

```
public class TestClass implements SetValueClass {
    private int value;

    TestClass() {
        this(10);
    }
    TestClass(int value) {
        this.value = value;
        AnotherClass ao = new AnotherClass(this);
        ao.run();
    }
    public void setValue(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public static void main(String args[]) {
             TestClasstc = new TestClass();
        System.out.println("Now the value is: "+tc.getValue());
    }
}
```

Interfaces

```
public class AnotherClass {
    SetValueClass svc;

    AnotherClass(SetValueClass svc) {
        this.svc = svc;
    }

    public void run() {
        int value = svc.getValue();
        svc.setValue(value * 2);
    }
}
```

Abstract Classes

- Abstract classes are similar to interfaces, but contain some implemented methods.
- Declare unimplemented methods as abstract and provide the signature.
- Declare the class abstract.
- Abstract classes cannot be instantiated. They must be subclassed.
- Classes that extend an Abstract class must implement all of the abstract methods.

Abstract Classes

```
public abstract class SetValueClass
{
    protected int value;

    public void printValue()
    {
        System.out.println("Value is: "+value);
    }

    public abstract void setValue(int i);
    public abstract int getValue();
}
```

Abstract Classes

```
public class AnotherClass {
    SetValueClass svc;

    AnotherClass(SetValueClass svc) {
        this.svc = svc;
    }

    public void run() {
        int value = svc.getValue();
        svc.setValue(value * 2);
    }
}
```

Abstract Classes

```
public class TestClass extends SetValueClass {
    TestClass() {
        this(10);
    }
    TestClass(int value) {
        this.value = value;
        AnotherClass ao = new AnotherClass(this);
        ao.run();
    }
    public void setValue(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public static void main(String args[]) {
        int value = Integer.parseInt(args[0]);
        TestClass tc = new TestClass(value);
        tc.printValue();
    }
}
```

UML Overview

UML Introduction

- UML – Unified Modeling Language
- UML is a industry accepted standard for presenting visual models of software systems.
- UML employs a number of diagrams to reflect different views of a software system.
- Each diagram type has its own distinct usage within a development cycle.
- UML gives developers, managers, designers and architects a unified way of drawing diagrams. If everyone on the team knows UML, then collaboration and understanding is increased.

UML Overview

UML Introduction

- UML builds upon a number of independent notation styles.
- Most of it lineage however comes from the old OMT style.
- The Design Patterns book uses OMT, but we will be using UML – sorry!
- UML designed by Grady Booch, James Rumbaugh and Ivar Jacobson – “The Three Amigos”
- UML is now under the Object Modeling Group (OMG) for standards maintenance.

UML Introduction

- For this class we will be focusing on only a few diagrams:
  - Class Diagram: used to show the static relationships between classes.
  - Sequence Diagram: used to show the sequence of events in instantiating and using objects. On one diagram, the lifeline of an object is shown with its interaction with other objects.

UML Tools

- There are a number of tools available for drawing UML diagrams.
  - Microsoft Visio
  - Rational Rose
  - Embarcadaro GDPro
  - Telelogic Tau UML Suite*
  - TogetherSoft TogetherJ
  - Standard Whiteboard

*We will be using Telelogic Tau UML Suite in this class

UML Tools

- Standard Whiteboard?
  - The whiteboard is my favorite tool because it allows for instant collaboration and explanation.
  - Most importantly, having people draw UML on whiteboards gives an instant filter to convey only the information required for the discussion at hand.
  - Preserve the UML in a tool after the discussion is over.
  - Persistent UML may or may not fit into your development process.

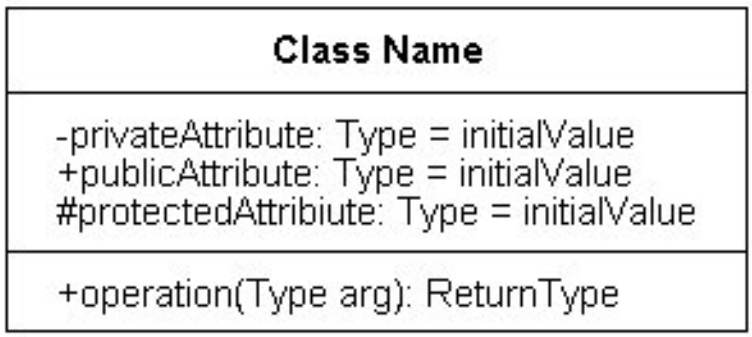
UML Overview

Class Diagram

- The class diagram is used to show static relationships between classes.
- Class diagrams do not show objects or show any state behavior. Methods (operations) may be stated, but generally do not convey any implementation details.
- At the heart of the class diagram is the class notation which shows the class name, attributes and operations (name, variables and methods).
- Adornments can be used to show things like visibility, abstract class and interfaces.

UML Overview

Class



UML Overview

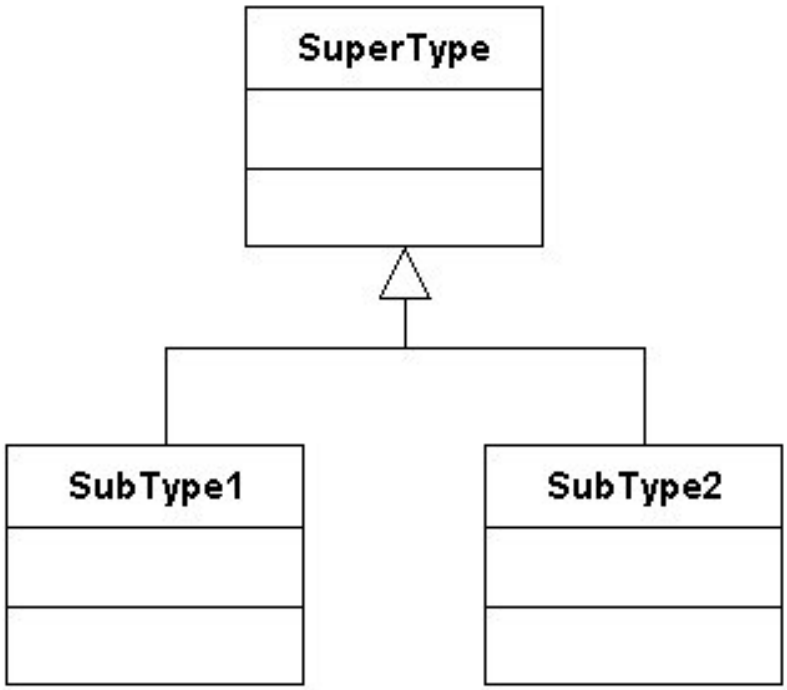
Relationships

- The relationships between classes have unique notations and adornments
- Relationships are:
  - Generalization
    - Inheritance
  - Association
    - Who knows about who
  - Dependency
    - Who is dependant on who

Generalization

- Generalization shows a “is a kind of” relationship.
- In Java, this is inheritance.

Generalization



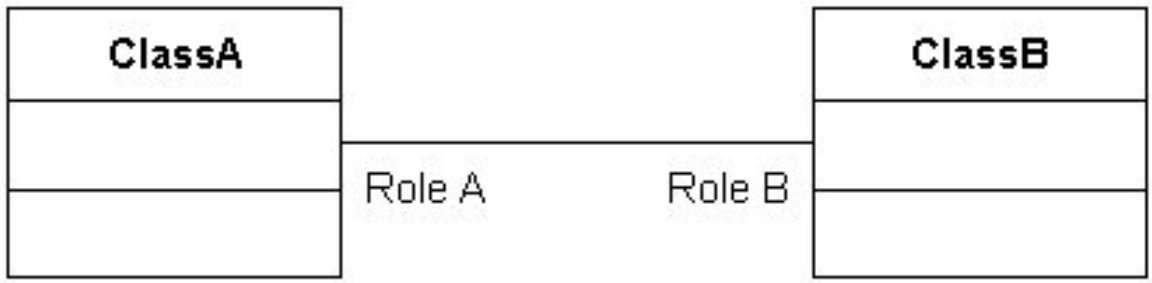
UML Overview

Association

- Association is a dependency relationship based on roles. “I know about you through this role”.
- In Java, roles tend to be instance variables containing a reference to an object of another class.
- Roles may or may not be displayed.
- Specific adornments on associations further define details about the association.

UML Overview

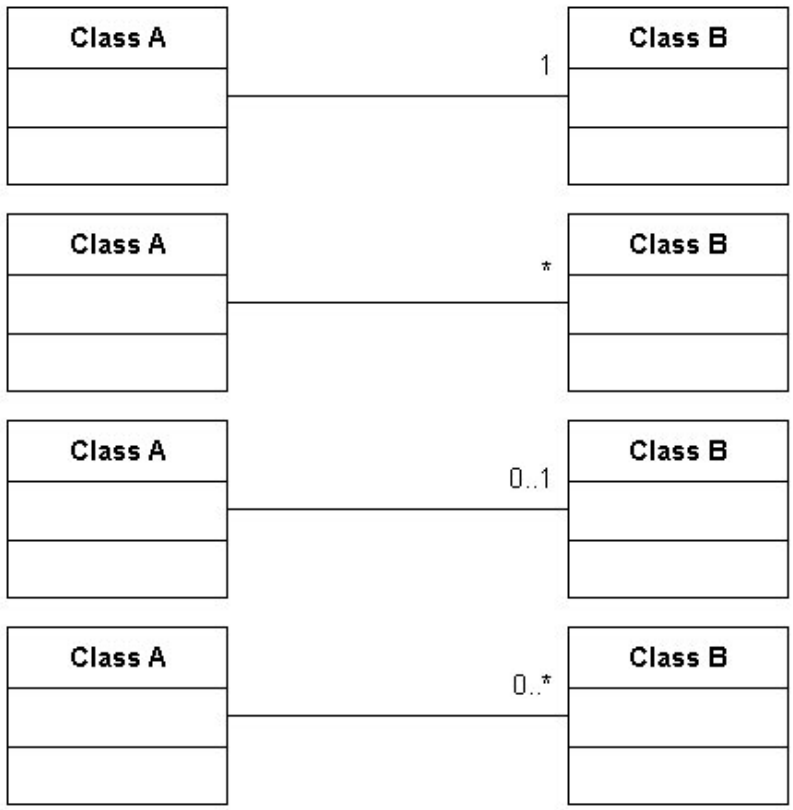
Association



UML Overview

- The multiplicity adornments give cardinality to an association.
  - If no multiplicity adornment is stated, it is assumed to be 1.
  - You can place a fixed number which says exactly how many instances of the class are involved in the association.
  - The “*” adornment means zero to many
  - The “0..1” means zero or one.
  - You can also specify a range using “m..n” for example, “0..*” means any number.

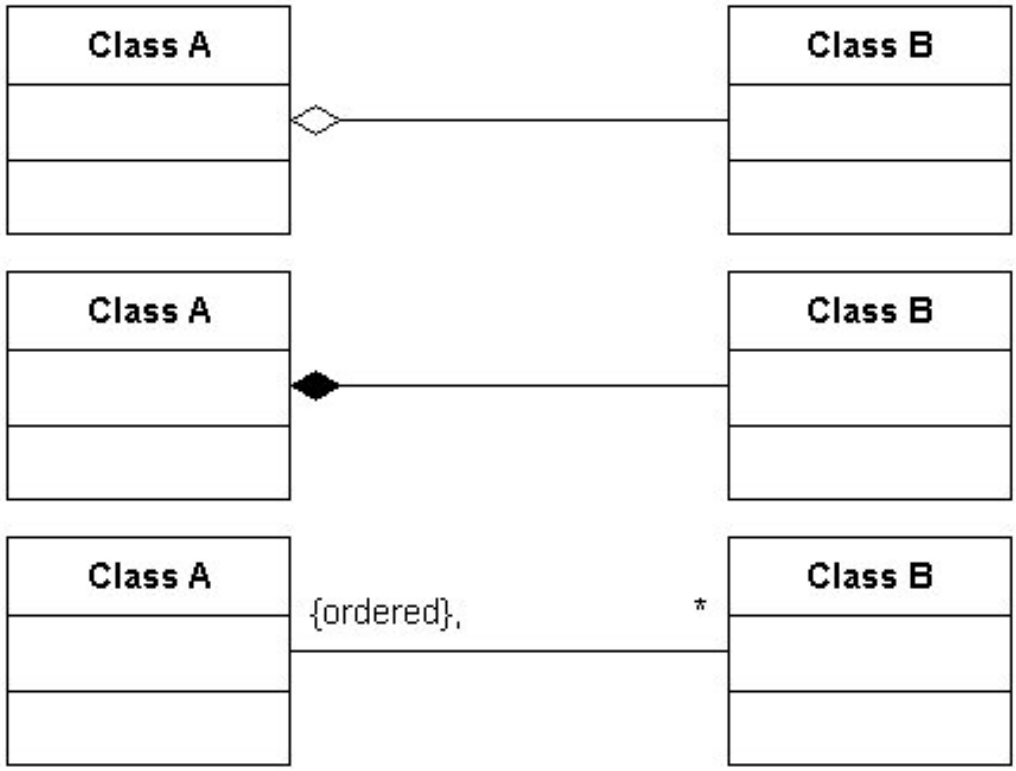
Multiplicity



UML Overview

- The composition adornments give details about coupling to the association.
  - Aggregation, specified by an open diamond, implies a loose coupling. In general it means that the class is a container of these objects, but the objects have a life outside of the class.
  - Composition, specified by a filled in diamond, implies a tight coupling. The classes is a container of objects and that class probably created and destroys the objects.
  - The {ordered} adornment shows that the association is stored with some kind of ordering.

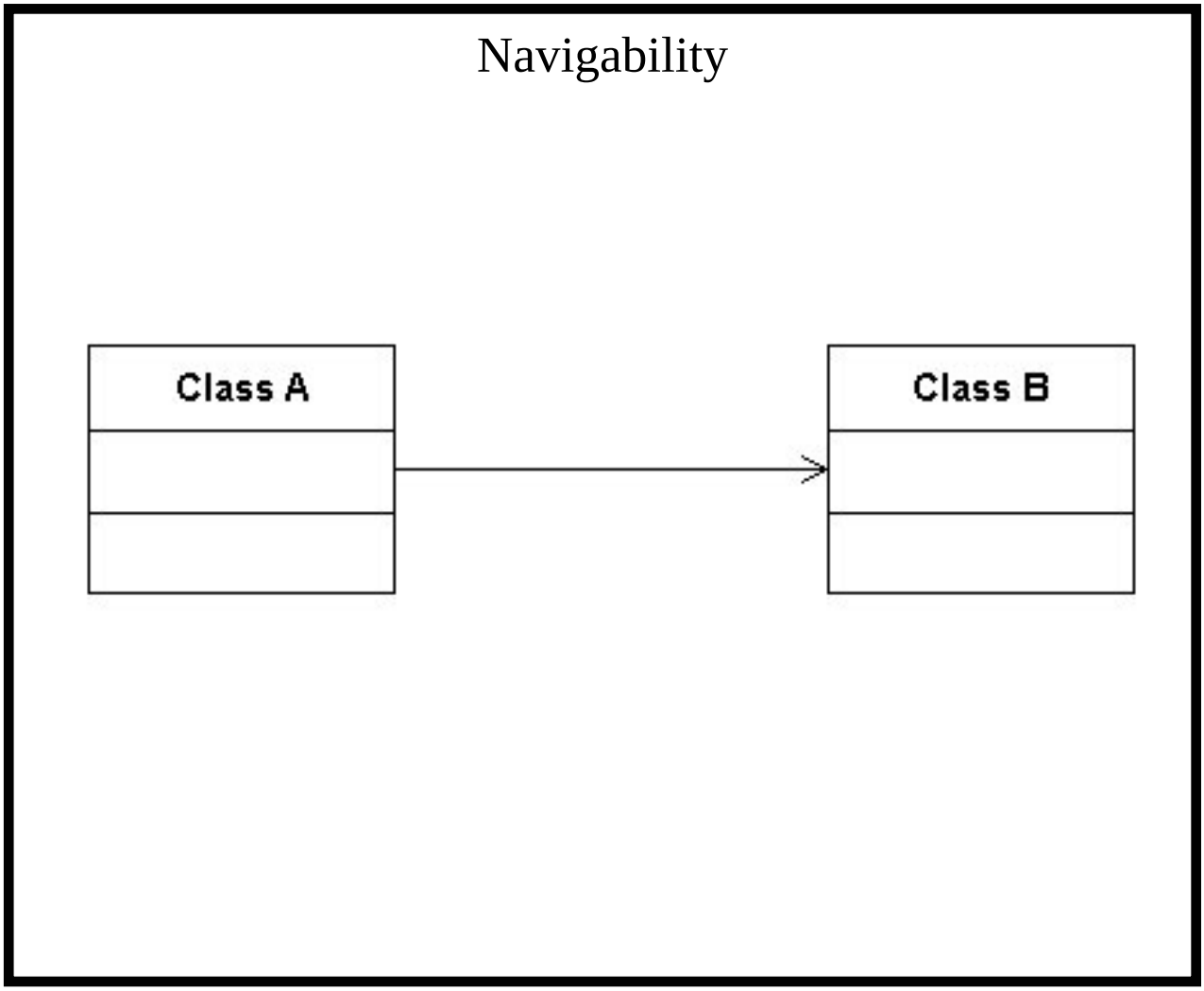
Composition



UML Overview

- The navigability adornment shows which class knows about the other class.
- The default is no adornment which means that the classes each now about each other.
- An arrow is used to show that one class directly knows about the other.

UML Overview



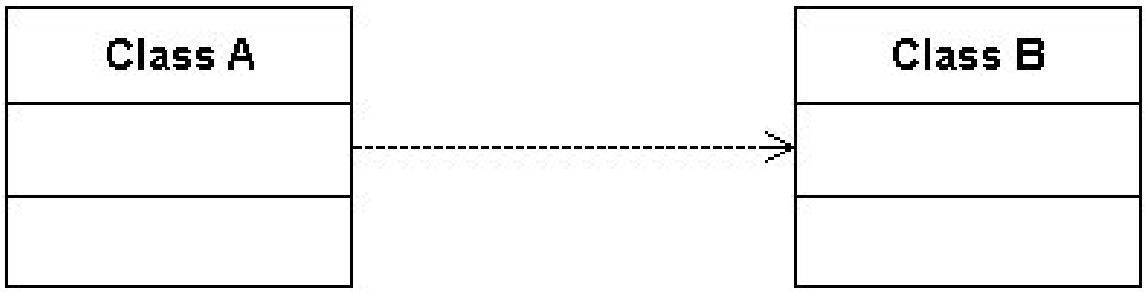
Page: 45

UML Overview

- A dependency relationship is show using a dashed line. This says that one class is dependant on another class to do some critical work for it.

UML Overview

Dependency

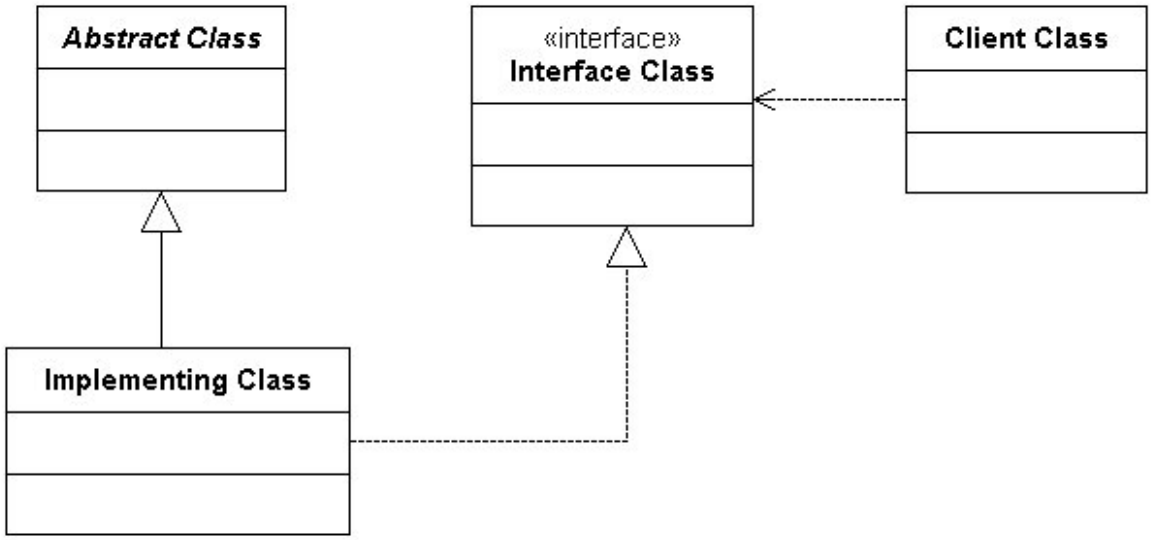


UML Overview

- A class that is an interface is shown using the <<interface>> adornment.
- An abstract class is shown by using italics in the name of the class.
- A class that implements an interface is shown with a dashed line with an open arrow pointing to the interface.
- A class that uses an interface has a dependency on the interface.
  - This is such a common relationship that a special adornment called a lollipop is used.
  - Mostly used when when dealing with a well known interface.

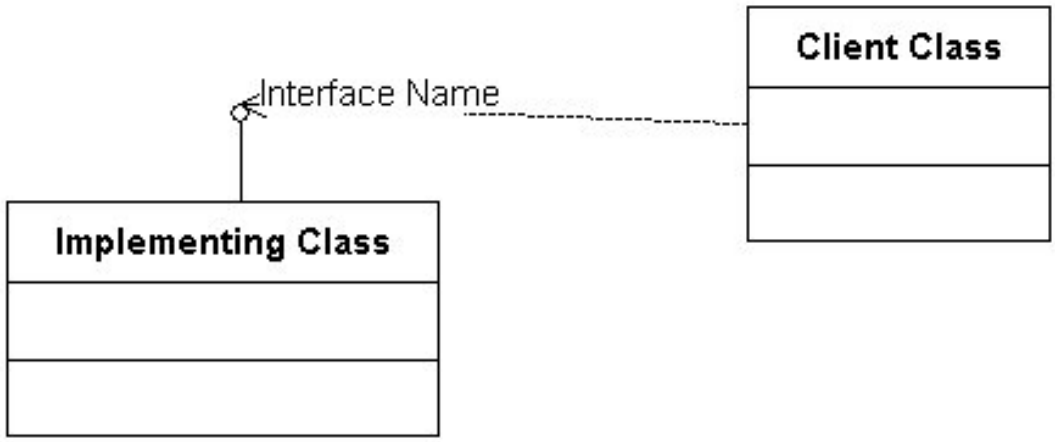
UML Overview

Interfaces



UML Overview

Interfaces



UML Sequence Diagrams

- A sequence diagram is used to show the sequence of events in instantiating and using objects. On the diagram, the lifeline of an object is shown with its interaction with other objects.
- Unlike the class diagram, sequence diagrams deal with objects and methods.
- Some of the relationships represented on a class diagram are realized on the sequence diagram.

UML Sequence Diagrams

- The key notations on a sequence diagram are:
- Objects
  - Shown as a box at the top of a vertical line
  - The vertical line is the object's lifeline which represents the object's life during the particular interaction.
  - An activation bar can exist along the lifeline to show when an object is actively being used. This generally means that a method is being called on that object.

UML Sequence Diagrams

- Messages
  - Messages are normally method calls.
  - There are a few kinds of messages
    - A message from on object to the object box of another is a create method. In Java, this is normally a call to a constructor using the **`new`** operator.
    - A message from one object's line to another object's lifeline is a normal method call where the calling object waits for the method to return.
    - A message may point back to the objects own lifeline. This is a case of a method calling another method in the same object.

UML Sequence Diagrams

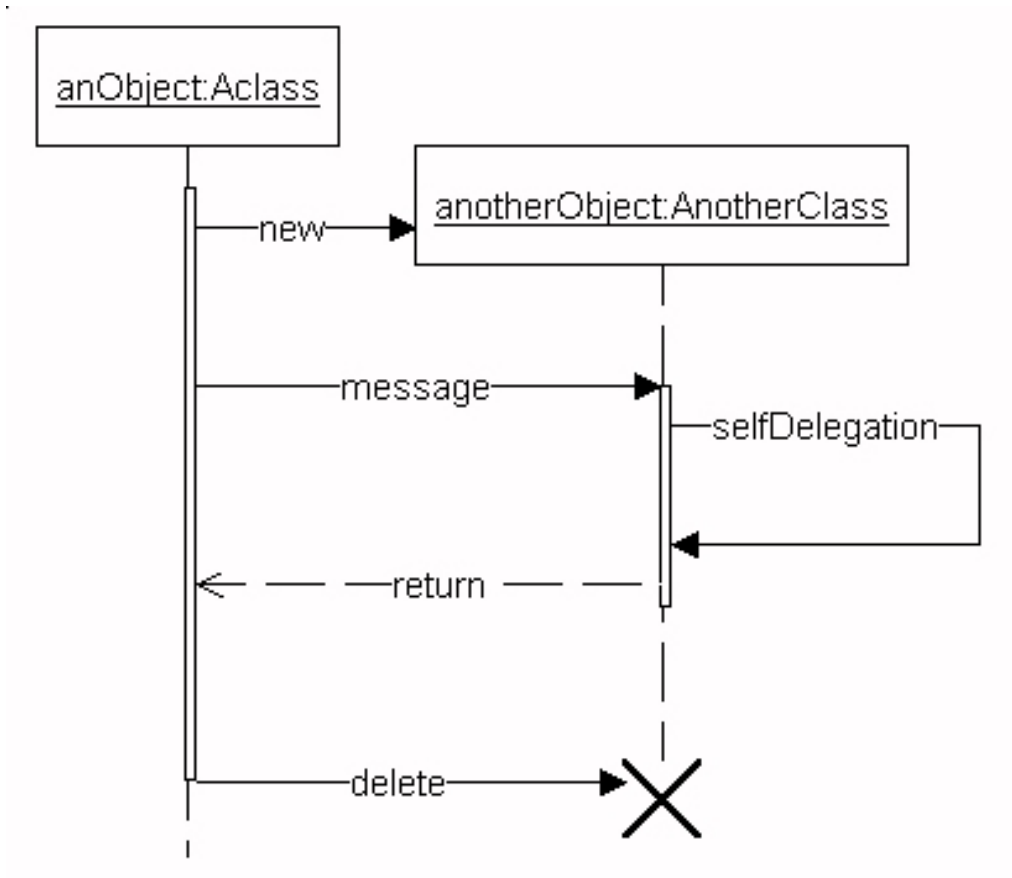
- Messages (continued)
  - If the message is represented with a half open arrow, then it is an asynchronous message. These messages allow the calling object to continue and not wait for the return result of the called method. In java these are generally threads.
  - Optionally you may use a dashed arrow pointing back to the calling class's lifeline to show a return. This is normally only done for clarity and the end of an activation bar represents the same thing.

UML Sequence Diagrams

- Messages (continued)
  - At the end of a lifeline you can have a big X which denotes the end of an objects existence. Any message pointing to this X is assumed to be a destructor.
  - In Java, the garage collector is in charge of calling `finalize` methods so I would say message pointing to the X probably represent explicit de-referencing by setting the reference to `null`. In some extreme cases, this may be an explicit call to the garbage collector.

UML Overview

UML Sequence Diagram



UML Sequence Diagrams

- More on Activation Bars
  - Activation Bars show when a particular class is currently being used. In general, this will show when a particular method is being called.
  - You may nest activation bars to show other methods getting called from with a method. This makes self delegation very clear.
  - Many tools will treat all messages originating from a given activation bar to be a part of a logical scope. This is nice since the messages will move with the activation bar as you use the tool for layout.

Design Patterns Catalog

- The Design Patterns Catalog consists of 23 design patterns.
- The catalog is divided up into three chapters
  - Creational Patterns
    - Deal with how objects are instantiated.
  - Structural Patterns
    - Deal with how classes and objects are composed to form larger structural concepts.
  - Behavioral Patterns
    - Deal with the algorithms and the assignment of responsibilities between objects.

Design Patterns Catalog

- Each Pattern is describe with some combination of the following:
- **Pattern Name and Classification**
  - A descriptive name
- **Intent**
  - A short description describing what the pattern does.
- **Also Know As**
  - Other names this pattern may have been referred to.
- **Motivation**
  - A scenario that describes the problem and how the pattern is used to solve the problem.

Design Patterns Catalog

- **Applicability**
  - The situation in which the pattern can be applied and what are some of the poor designs that the pattern can address.
- **Structure**
  - An OMT diagram giving a graphical representation of the pattern.
- **Participants**
  - The classes and/or objects participating in the pattern and their responsibilities. All of these should be shown in the OMT diagram.
- **Collaborations**
  - How the participants collaborate to carry out their responsibilities.

Design Patterns Catalog

- **Consequences**
  - How the pattern really work to support its objectives and the trade-offs and results of using the pattern.
- **Implementation**
  - Issues you should be aware of while implementing the pattern. Of particular importance here are variations on the pattern in which you can find ways which are better suited for Java
- **Sample Code**
  - Code fragments which further illustrate the pattern. Sorry folks, all of the examples are in either C++ or Smalltalk.

Design Patterns Catalog

- **Know Uses**
  - Real world examples of the use of the pattern.
- **Related Patterns**
  - Other patterns which are closely related.

Creational Patterns

- The simple and most common way for instantiating a class into an object is through the new operator.

```
MyClass myClass = new MyClass();
```

- So what's wrong with that?
  - Well, the calling class has a very tight coupling with the instantiated class.
  - This breaks our Object Oriented rules of abstraction and encapsulation.
  - Also, it is not very flexible. It is very difficult to swap in a different implementation of MyClass.

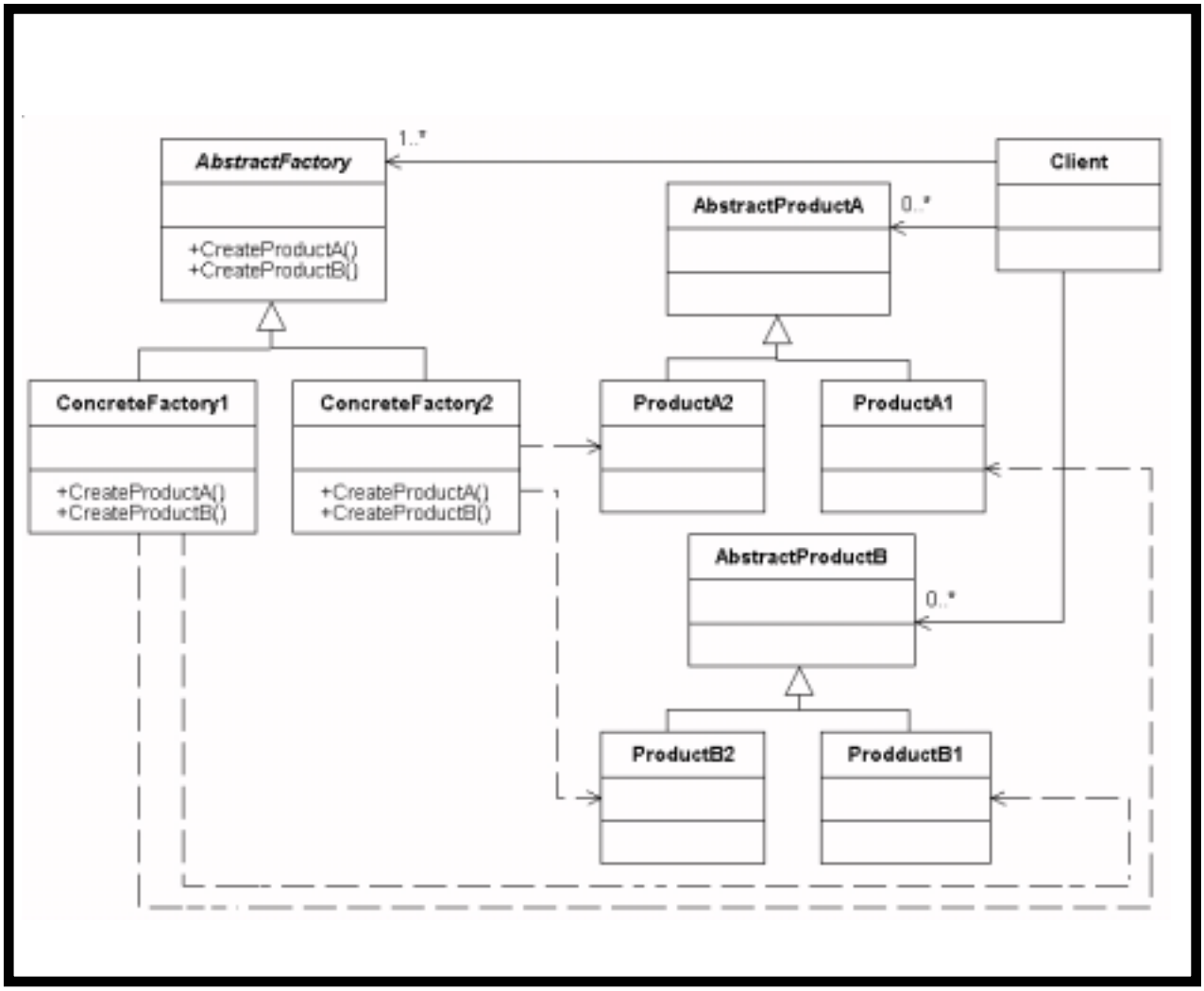
Creational Patterns

- The Creational Patterns take over the responsibility of instantiating objects. The result of a creational pattern is an instance of a class which can be used.
- Classes using the instantiated objects are coded to a known interface rather than an actual implementation.
- The knowledge of the concrete classes are fully encapsulated in the pattern and invisible to the user.
- How the concrete classes are instantiated and initialized is also fully encapsulated.

Abstract Factory

- The Abstract Factory Pattern is used to provide an interface for creating a family of related or dependent objects without exposing their concrete classes.
- For this, two sets of interfaces are need.
  - The Factory itself needs a known interface which exposes the create method signatures.
  - The objects which the Factory creates will also need to have defined interfaces.
- Someone will instantiate a concrete instance of the factory which will be passed to the client. The client will simply code to the Abstract Factory interface and to the Interfaces of products. The client does not know about any concrete classes.

Abstract Factory



Page: 66

Abstract Factory

Code Sample – AbstractFactory.java

```
public abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
```

Abstract Factory

Code Sample – ConcreteFactory1.java

```
public class ConcreteFactory1 extends AbstractFactory
{
    public AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}
```

Abstract Factory

Code Sample – ConcreteFactory2.java

```
public class ConcreteFactory2 extends AbstractFactory
{
    public AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }

    public AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}
```

Abstract Factory

Code Samples – AbstractProductA.java, ProductA1.java and ProductA2.java

```
public abstract class AbstractProductA {
    public abstract void run();
}

public  class ProductA1 extends AbstractProductA {
    public void run() {
        System.out.println("run() in ProductA1");
    }
}

public  class ProductA2 extends AbstractProductA {
    public void run() {
        System.out.println("run() in ProductA2");
    }
}
```

Abstract Factory

Code Sample – AbstractProductB.java, ProductB1.java and ProductB2.java

```
public abstract class AbstractProductB {
    public abstract void doIt();
}

public  class ProductB1 extends AbstractProductB {
    public void doIt() {
        System.out.println("doIt() in ProductB1");
    }
}

public  class ProductB2 extends AbstractProductB {
    public void doIt() {
        System.out.println("doIt() in ProductB2");
    }
}
```

Abstract Factory

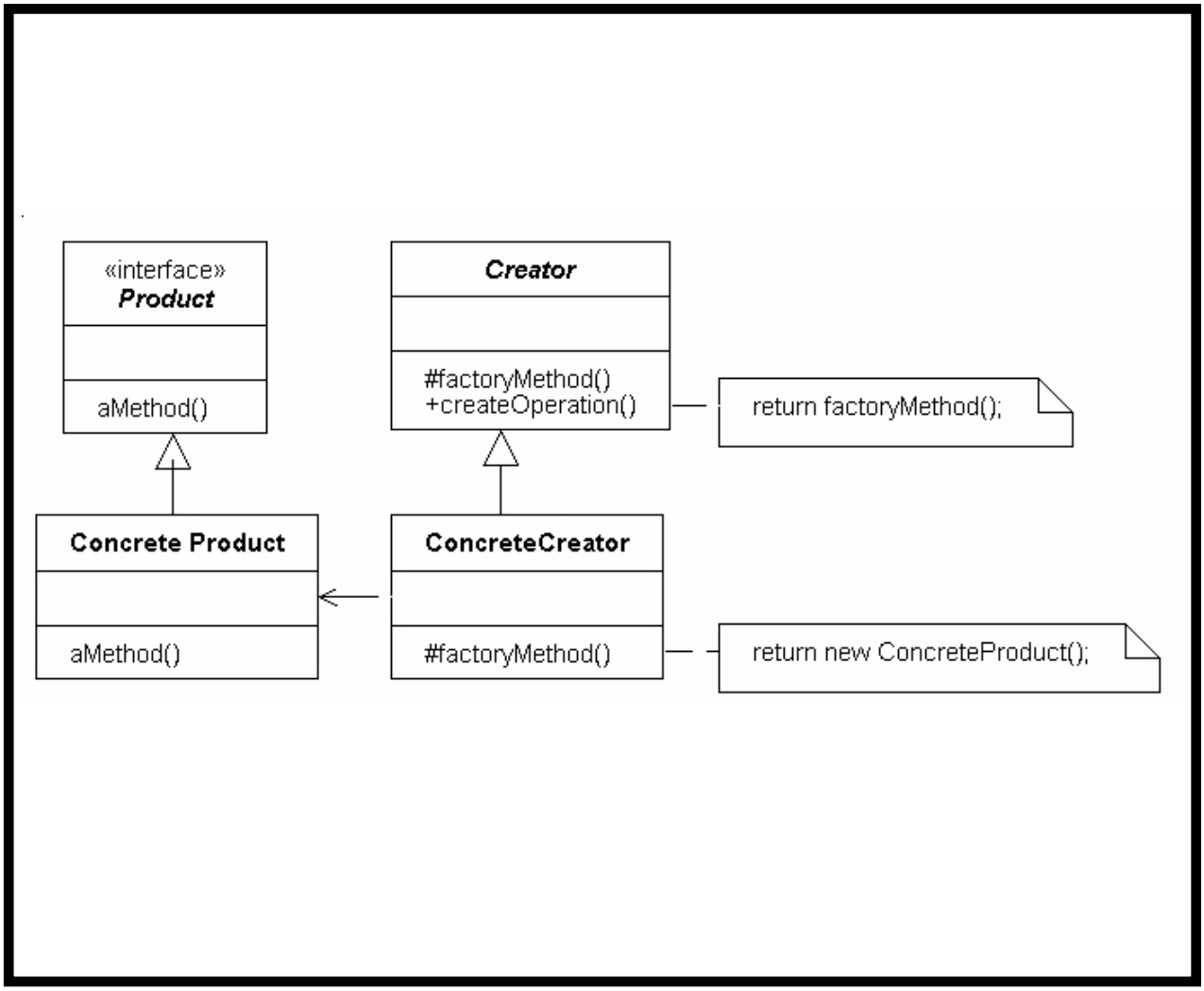
Code Sample – Client.java

```
public class Client
{
    Client(AbstractFactory factory)
    {
        AbstractProductA pa = factory.CreateProductA();
        AbstractProductB pb = factory.CreateProductB();
        pa.run();
        pb.doIt();
    }

    public static void main(String args[])
    {
        new Client(new ConcreteFactory1());
        new Client(new ConcreteFactory2());
    }
}
```

Factory Method

- The Factory Method Pattern defines an interface for creating an object but leave the actual instantiation up to a subclass.
- Very similar to an Abstract Factory except it is only really used to instantiate one object and defers the actual method to do the creation to a subclass.

Factory Method Pattern



Factory Method Pattern

Code Sample – Creator.java

```
public abstract class Creator {
    protected abstract Product factoryMethod();

    public Product createOperation(){
        return factoryMethod();
    }
}
```

Code Sample – ConcreteCreator.java

```
public class ConcreteCreator extends Creator {

    protected Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

Code Sample – AnotherConcreteCreator.java

```
public class AnotherConcreteCreator extends Creator {

    protected Product factoryMethod() {
        return new AnotherConcreteProduct();
    }
}
```

Factory Method Pattern

Code Sample - Product.java

```
public interface Product {
    public String aMethod();
}
```

Code Sample - ConcreteProduct.java

```
public class ConcreteProduct implements Product {
    public String aMethod() {
        return ("Hello, world");
    }
}
```

Code Sample - AnotherConcreteProduct.java

```
public class AnotherConcreteProduct implements Product
   {
    public String aMethod() {
        return ("Goodbye cruel world");
    }
}
```

Factory Method Pattern

Sample Code – FactoryMethodClient.java

```
public class FactoryMethodClient {

    public FactoryMethodClient(Creator creator) {
        Product product = creator.createOperation();
        System.out.println(product.aMethod());
    }
}
```

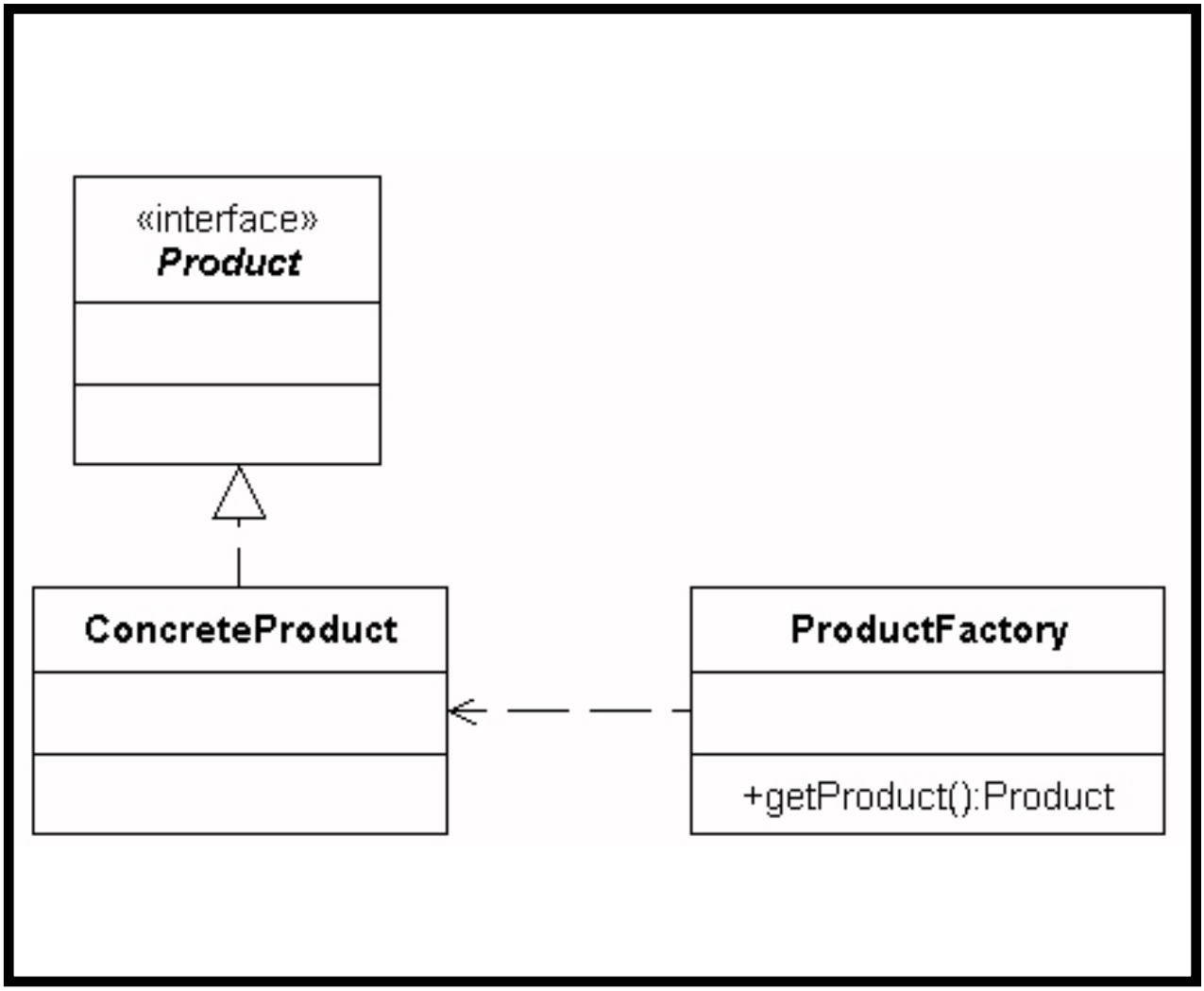
Sample Code – FactoryMethodExample.java

```
public class FactoryMethodExample {

    public FactoryMethodExample() {
        new FactoryMethodClient(new ConcreteCreator());
        new FactoryMethodClient(new AnotherConcreteCreator());
    }

    public static void main(String args[]) {
        new FactoryMethodExample();
    }
}
```

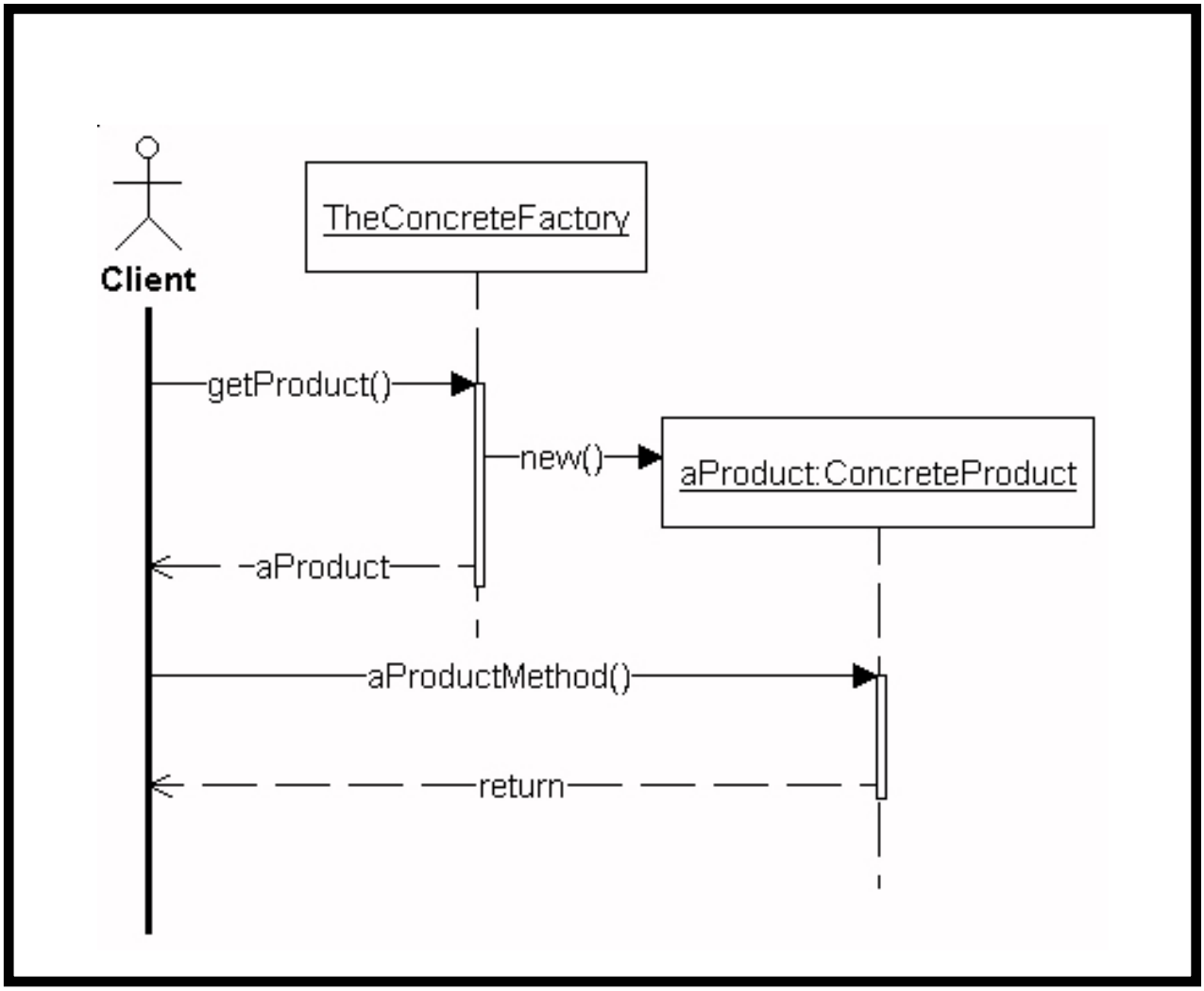
Factory Method Pattern

- One of the interesting variations on the Factory Method Pattern described in the Implementation section of the Factory Pattern Catalog entry is to use a Concrete Factory class which determines exactly which Concrete Product to construct.
- This totally encapsulates the creation process.
- However, you will still need some way of telling the Factory which class to instantiate. This can be done in a few ways:
  - Pass some parameters which can give clues.
  - Set up an environment option which contains the class name.
  - Query a naming service for the class name.
    - The last two are common in deployment environments.

Factory Method Pattern



Factory Method Pattern



Factory Method Pattern

Code Sample – ProductFactory.java

```
public class ProductFactory {
    private static Class productClass;

    public static Product getProduct()
        throws Exception {
      Product product = null;
      try {
        if ( productClass == null ) {
          String productClassName =
                 System.getProperty("ProductClass");
          if ( productClassName == null ){
            productClassName = "ConcreteProduct";
          }
          productClass = Class.forName(productClassName);
        }
        product = (Product) productClass.newInstance();
      }
      catch ( ClassNotFoundException  cnfe ) {
        throw new Exception(cnfe.getMessage());
      }
      catch ( InstantiationException ie ){
        throw new Exception(ie.getMessage());
      }
      catch ( IllegalAccessException iae ){
        throw new Exception(iae.getMessage());
      }
      return product;
    }
}
```

Factory Method Pattern

Code Sample – FactoryMethodClient.java

```
public class FactoryMethodClient {
    public FactoryMethodClient() {
      try {
        Product product = ProductFactory.getProduct();
            System.out.println(product.aMethod());
      }
      catch ( Exception e ) {
            System.out.println( e.getMessage() );
      }
    }
}
```

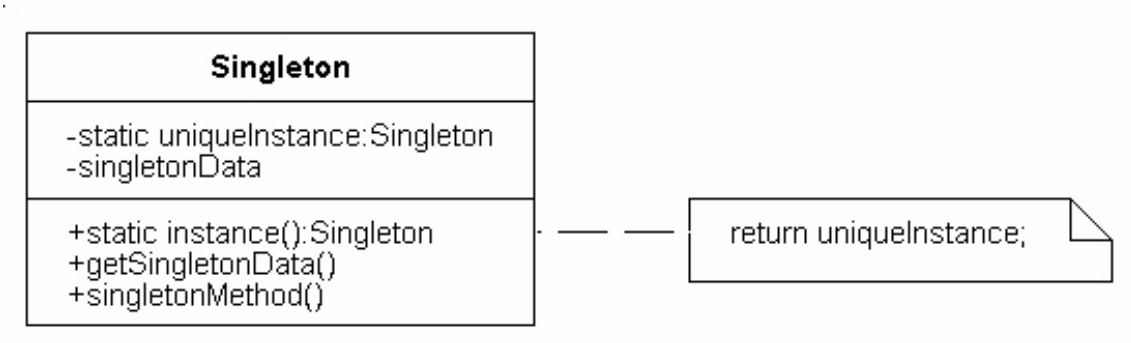
Code Sample – FactoryMethodExanple.java

```
public class FactoryMethodExample {

    public FactoryMethodExample() {
        new FactoryMethodClient();
    }

    public static void main(String args[]) {
        new FactoryMethodExample();
    }
}
```

Singleton Pattern

- Singletons are used whenever you want to enforce only one instance of a class can exist.
- Normally, this is done to present some form of global data – often a resource or a manager of resources.
  - Print Spools
  - Thread Pools
- One access point returns the unique instance.
- Constructors are made private to prevent anyone from access the data.
- You could simple make everything in a class static, but then it would be hard to use the class with other patterns that expect an object.

Singleton Pattern



Singleton Pattern

Code Sample – Singleton.java

```
public class Singleton
{
    private static Singleton uniqueInstance = null;
    private String singletonData = "Hello, world";

    private Singleton() {
    }
    public static Singleton instance() {
        if ( uniqueInstance == null ) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    public void setSingletonData(String message) {
        singletonData = message;
    }

    public String getSingletonData() {
        return singletonData;
    }
}
```

Singleton Pattern

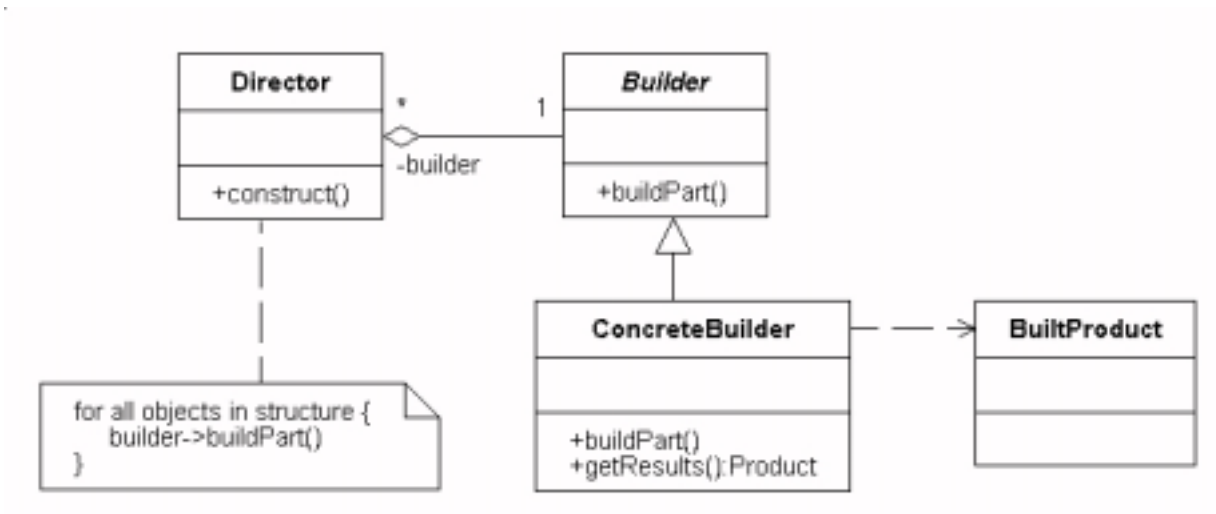
Code Sample – Singleton.java

```
public class SingletonExample
{
    public SingletonExample() {
        Singleton firstSingleton = Singleton.instance();
        Singleton secondSingleton = Singleton.instance();
        System.out.println(firstSingleton.getSingletonData());

    System.out.println(secondSingleton.getSingletonData());
        firstSingleton.setSingletonData("Goodby cruel world");
        System.out.println(firstSingleton.getSingletonData());

    System.out.println(secondSingleton.getSingletonData());
    }

    public static void main(String args[]) {
        new SingletonExample();
    }
}
```

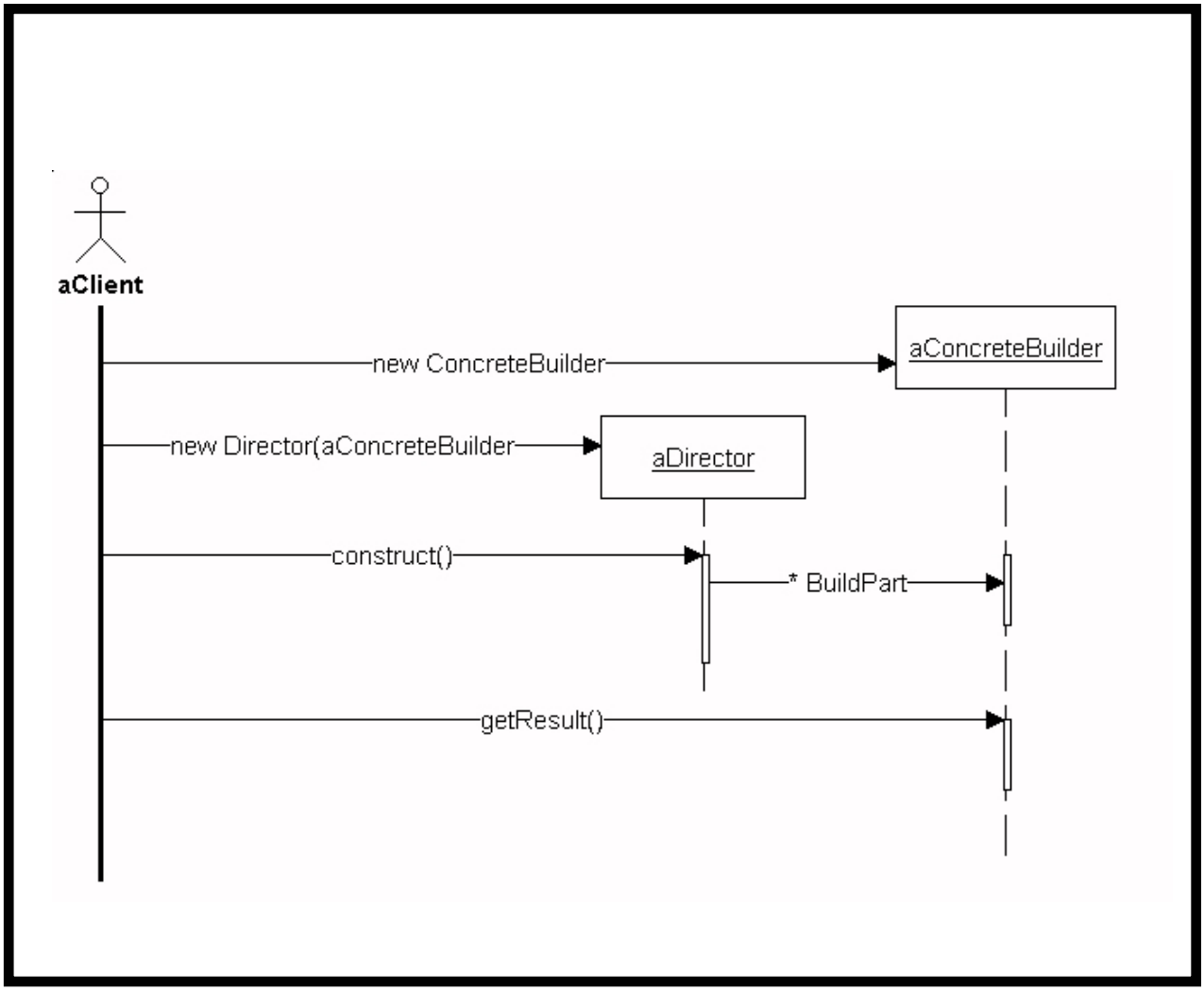
Builder pattern

- The builder pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different representations.
- The construction process is represented inside of a class generically called a Director. This class knows only the steps needed to construct an object by calling known methods of a Builder interface.
- The Builder interface defines methods which the director uses.
- ConcreteBuilders implement the Builders interface to actually construct a Product.
- The ConcreteBuilder will also have a method which will allow a client to get the instance of the product class.

Builder Pattern



Builder Pattern



Page: 92

Code Sample – DateBuilder.java

```
public abstract class DateBuilder
{
    String dateString;

    public abstract void buildDate(int month, int day, int year);

    public String getDateString()
    {
        return dateString;
    }
}
```

Code Sample – USDateBuilder.java

```
public class USDateBuilder extends DateBuilder
{
    public void buildDate(int month, int day, int year)
    {
        dateString = month+"/"+day+"/"+year;
    }
}
```

Code Sample – EuroDateBuilder.java

```
public class EuroDateBuilder extends DateBuilder
{
    public void buildDate(int month, int day, int year)
    {
        dateString = day+"/"+month+"/"+year;
    }
}
```

Code Sample – DateDirector.java

```
class DateDirector
{
    DateBuilder dateBuilder;

    DateDirector(DateBuilder dateBuilder)
    {
        this.dateBuilder = dateBuilder;
    }

    public void constructDate(int month, int date, int year)
    {
        dateBuilder.buildDate(month, date, year);
    }
}
```

Builder Pattern

Code Sample – Client.java

```
public class Client
{
    DateBuilder dateBuilder;
    DateDirector dateDirector;

    Client( DateBuilder dateBuilder)
    {
        dateDirector = new DateDirector(dateBuilder);
        this.dateBuilder = dateBuilder;
        this.dateDirector = dateDirector;
    }

    public String genDate(int month, int day, int year)
    {
        dateDirector.constructDate(month, day, year);
        return dateBuilder.getDateString();
    }

      // Continued on next page
```

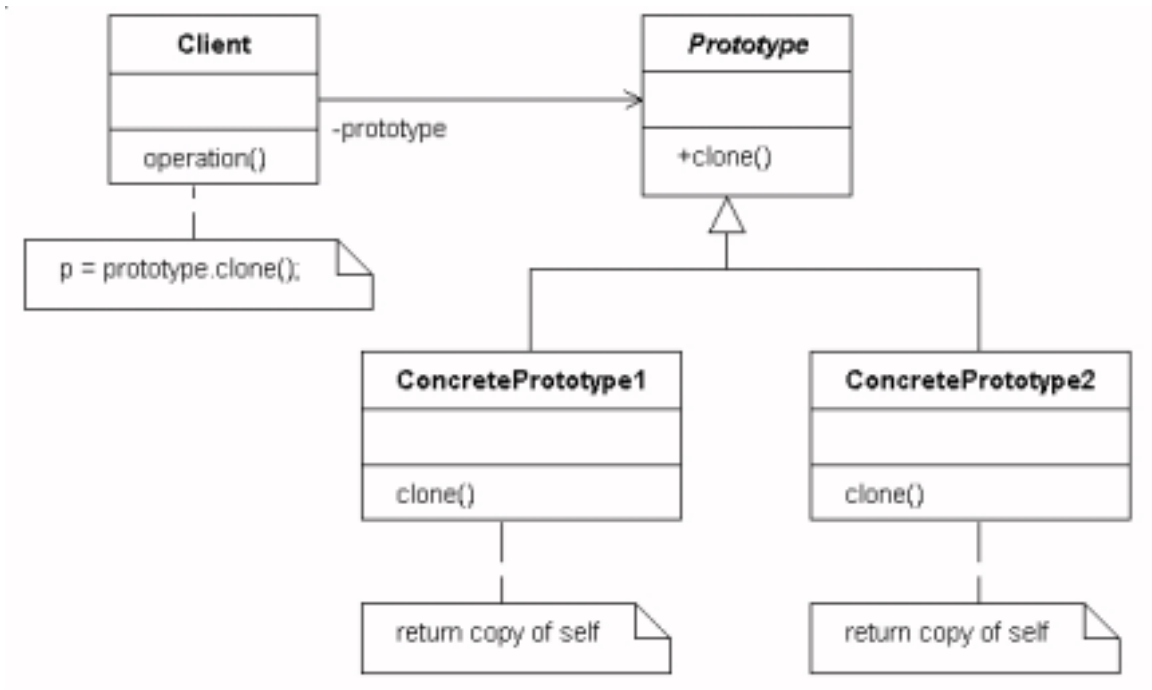
Code Sample – Client.java (cont.)

```
    public static void main(String args[])
    {
        DateBuilder dateBuilder = new USDateBuilder();
        Client client = new Client(dateBuilder);
        System.out.println("US Date: "+client.genDate(7,12,65));

        dateBuilder = new EuroDateBuilder();
        client = new Client(dateBuilder);
        System.out.println(
                "EURO Date: "+client.genDate(7,12,65));
    }
}
```

Prototype Pattern

- The Prototype Pattern uses an object which can create new instances of itself by cloning.
- This is generally useful if you need to deal with exact copies of an object or if new instances of the class need to be fully populated with the data from another.
- In Java, this is normally accomplished using a clone method. Additional flexibility can be achieved by providing an Abstract class or an interface to a set of similar Prototypes.

Prototype Pattern



Page: 100

Prototype Pattern

Code Sample – Phrase.java

```
// this is the Prototype
public abstract class Phrase
{
    String noun;

    public void setNoun(String noun)
    {
        this.noun = noun;
    }

    public void printPhrase()
    {
        System.out.println(getPhrase());
    }

    public abstract String getPhrase();

    public abstract Phrase cloneMe();
}
```

Prototype Pattern

Code Sample – HelloPhrase.java

```
public class HelloPhrase extends Phrase
{
    public String getPhrase()
    {
        return ("Hello, " + noun);
    }

    public Phrase cloneMe()
    {
        HelloPhrase hp = new HelloPhrase();
        hp.noun = this.noun;
        return hp;
    }
}
```

Prototype Pattern

Code Sample – GoodbyePhrase.java

```
public class GoodbyePhrase extends Phrase
{
    public String getPhrase()
    {
        return ("Goodbye, " + noun);
    }

    public Phrase cloneMe()
    {
        GoodbyePhrase gbp = new GoodbyePhrase();
        gbp.noun = this.noun;
        return gbp;
    }
}
```

Prototype Pattern

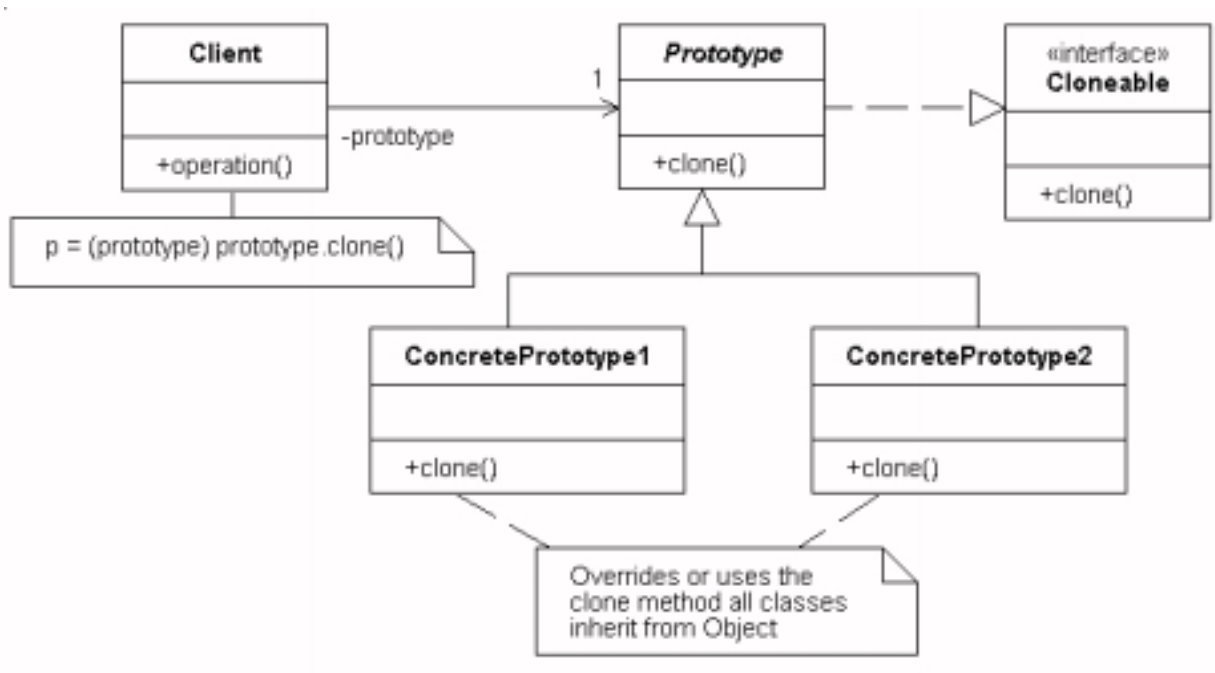
Code Sample – Client.java

```
public class Client
{
    public Client(Phrase protoPhrase) {
        protoPhrase.printPhrase();
        Phrase anotherPhrase = protoPhrase.cloneMe();
        anotherPhrase.printPhrase();
        anotherPhrase.setNoun("Barry");
        anotherPhrase.printPhrase();
    }

     public static void main(String args[]) {
        Phrase hp = new HelloPhrase();
        hp.setNoun("World");
        new Client(hp);
        Phrase gbp = new GoodbyePhrase();
        gbp.setNoun("World");
        new Client(gbp);
    }
}
```

Page: 104

Prototype Pattern

Prototype using the Clone method on the Java Object Class



Page: 105

Structural Patterns

- Structural patterns show how classes and objects are composed to form larger structures.
- Use inheritance and composition to define interfaces and allows the classes to be linked together.
- Defines ways to compose objects to realize new functionality without major code rewrite.
- Like the creational patterns, the structural patterns do have similarity to each other. The subtle differences will help you determine where each pattern should be used.

Adapter Pattern

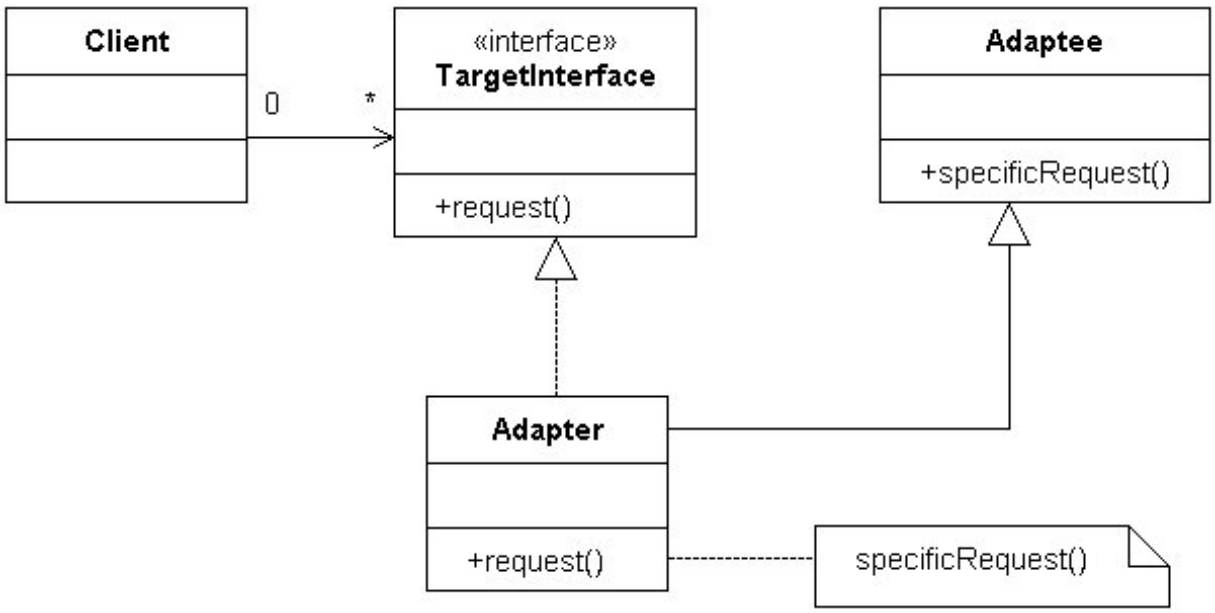
- The adapter pattern is used to convert the interface which one class may expect into another interface.
- This allows two classes to work together in cases where they normally could not due to incompatible interfaces.
- The GOF book shows two patterns to implement the adapter pattern. One of them uses multiple inheritance. Since Java does not have multiple inheritance, we will look at an alternative that uses single inheritance and an interface implementation.
  - This is a good example of how to realize the same functionality that multiple inheritance would provide in Java.

Adapter Pattern

- In the adapter via inheritance pattern, an Adapter class is defined that extends the implemented class to which we will adapt (the Adaptee) and we implement an interface which the client expects (the TargetInterface).
- The Adapter class will implement each of the methods defined in the Target interface.
- Each of those methods will call an appropriate method in the base class thus realizing the base classes interface.
- In the GOF pattern, the Adapter class extends both the Target and the Adaptee which makes it possible to use the pattern if there is no Interface for the Target.

Adapter Pattern

Adapter Using Inheritance



Page: 109

Adapter Pattern

- The adapter pattern can also be expressed using composition rather than inheritance.
- In this model, the Adapter extends the Target class and contains an instance of the Adaptee.
- The methods in the Target class ore overridden to call the appropriate method in the Adaptee.
- This model works much better in Java than the adapter pattern using Inheritance because it does not require that the Target have an interface.
  - Most legacy systems are not well designed using interfaces.

Adapter Pattern

Code Sample – Adaptee.java

```
public class Adaptee
{
    int value=0;

    public void setTheValue(int value)
    {
        this.value = value;
    }

    public int getTheValue()
    {
        return value;
    }
}
```

Page: 111

Code Sample – Target.java

```
public interface Target
{
    public void setValue(int value);
    public int getValue();
}
```

Adapter Pattern

Code Sample – Adapter,java

```
public class Adapter extends Adaptee implements
   Target
{
    public void setValue(int value)
    {
        setTheValue(value);
    }

    public int getValue()
    {
        return getTheValue();
    }
}
```

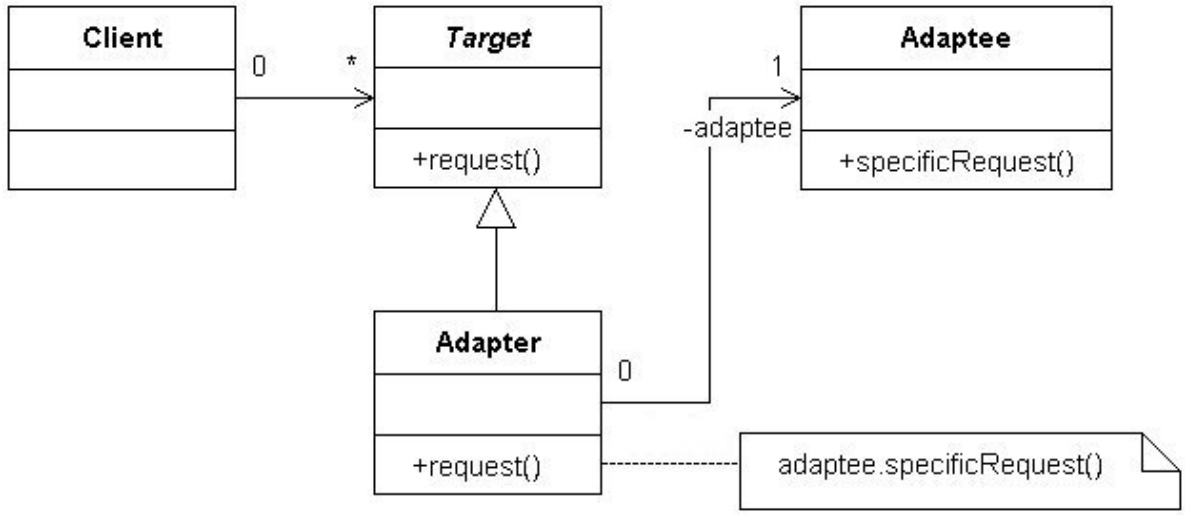
Adapter Pattern

Code Sample – Client.java

```
public class Client
{
    Client(Target theTarget)
    {
        theTarget.setValue(10);
        System.out.println("Value is:
   "+theTarget.getValue());
    }

    public static void main( String args[] )
    {
        Adapter theTarget = new Adapter();
        new Client(theTarget);
    }
}
```

Adapter Pattern

Adapter Using Composition



Code Sample – Adaptee.java

```
// Same as previous Adaptee
public class Adaptee
{
    int value=0;
    public void setTheValue(int value)
    {
        this.value = value;
    }
    public int getTheValue()
    {
        return value;
    }
}
```

Code Sample – Target.java

```
public abstract class Target
{
    abstract void setValue(int value);
    abstract int getValue();
}
```

Adapter Pattern

Code Sample – Adapter.java

```
public class Adapter extends Target {
    Adaptee adaptee;

    Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void setValue(int value) {
        adaptee.setTheValue(value);
    }

    public int getValue() {
        return adaptee.getTheValue();
    }
}
```

Page: 118

Adapter Pattern

Code Sample – Client.java

```
public class Client
{
    // Same as previous exampe
    Client(Target theTarget){
        theTarget.setValue(10);
        System.out.println("Value is:
   "+theTarget.getValue());
    }

    public static void main( String args[] ){
        Adapter theTarget = new Adapter(new
   Adaptee());
        new Client(theTarget);
    }
}
```

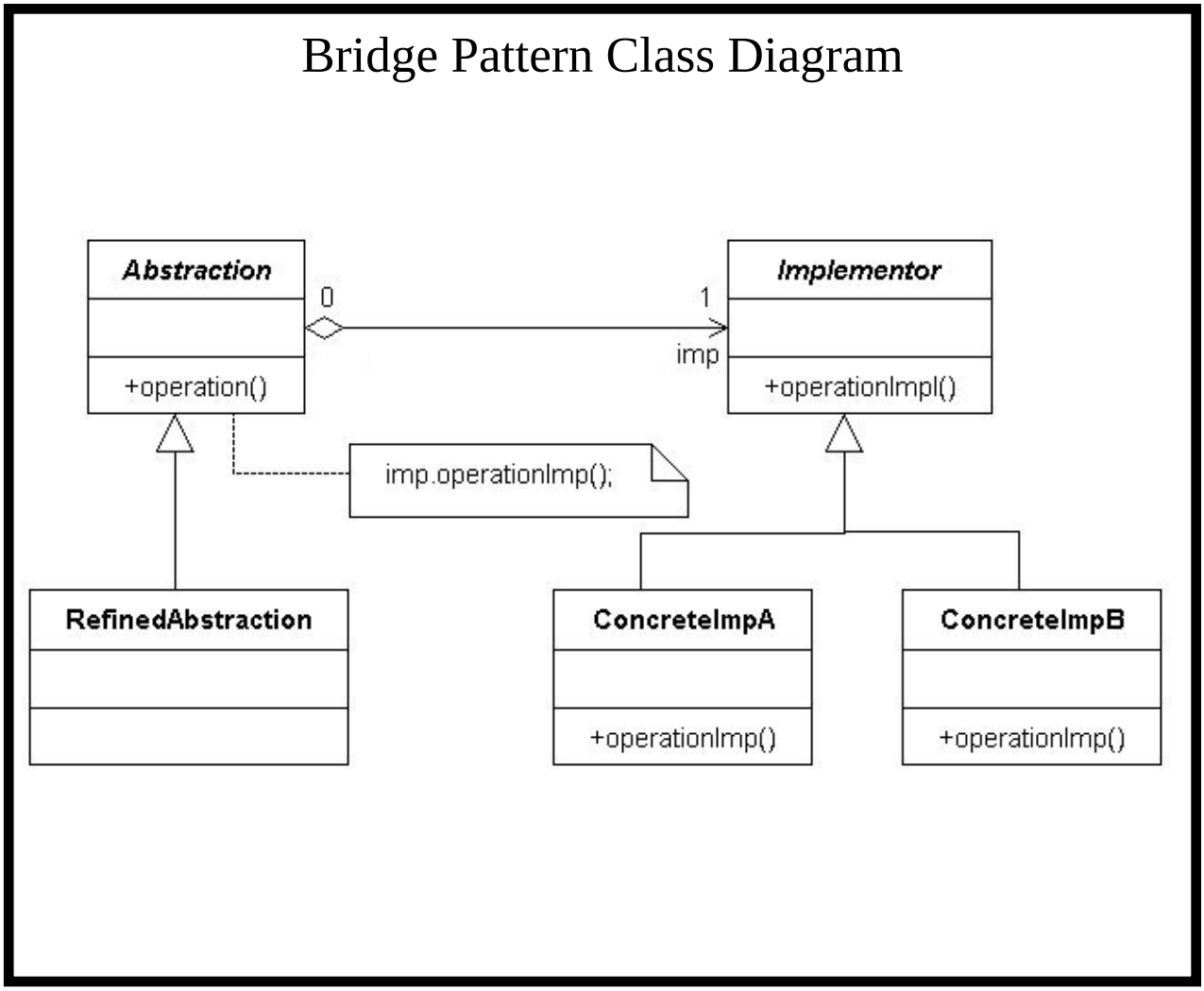
Bridge Pattern

- The bridge pattern is used to decouple an abstraction from its implementation so that the two can vary independently.
- This gives a greater level of flexibility to the implementation of the back end of a system but with some costs.
- The Abstraction class maintains a consistent interface which can be used by a client. Its methods know which methods can be called on the Implementor class.
- Subclasses of the Abstraction can be modified to encapsulate methods in the Abstraction.
- When new methods are added to the Implementor class, the Abstraction class will need to be updated. However, most of the changes can be confined to the Bridge (the Abstraction and the Implementor).

Bridge Pattern

- The Bridge Pattern is similar to the Adapter Pattern, but a Bridge assumes that the abstraction and Implementor are being designed and developed at the same time. The Adaptor pattern assumes that the Adaptee class is fixed and will not be changed.

Bridge Pattern



Bridge Pattern

Code Sample – Reporter.java

```
public abstract class Reporter
{
    String header = "";
    String trailer = "";
    String report = "";
    public abstract void addLine(String line);

    public void setHeader(String header) {
        this.header = header;
    }
    public void setTrailer(String trailer){
        this.trailer = trailer;
    }
    public String getReport(){
        return header+report+trailer;
    }
}
```

Code Sample – ASCIIReporter.java

```
public class ASCIIReporter extends Reporter
{
    public void addLine(String line)
    {
        report += line + "\n";
    }
}
```

Bridge Pattern

Code Sample – HTMLReporter.java

```
public class HTMLReporter extends Reporter
{
    public HTMLReporter()
    {
        setHeader("<HTML>\n<HEAD></HEAD>\n<BODY>\n");
        setTrailer("</BODY>\n</HTML>");
    }
    public void addLine(String line)
    {
        report += line + "<BR>\n";
    }
}
```

Code Sample – Report.java

```
import java.util.*;
public abstract class Report
{
    Reporter reporter;

    public Report(Reporter reporter){
        this.reporter = reporter;
    }

    public void addReportItem(Object item){
        reporter.addLine(item.toString());
    }

    public void addReportItems(List items){
        Iterator iterator = items.iterator();
        while ( iterator.hasNext() )
        {
                 reporter.addLine(
                         iterator.next().toString());
        }
    }

    public String report(){
        return reporter.getReport();
    }
}
```

Code Sample – StockListReport.java

```
import java.util.*;
public class StockListReport extends Report
{
    ArrayList stock=new ArrayList();

    public StockListReport(Reporter reporter)
    {
        super(reporter);
    }

    public void addStockItem(StockItem stockItem)
    {
        stock.add(stockItem);
        addReportItem(stockItem);
    }
}
```

Code Sample – StockItem.java

```
public class StockItem
{
    int count;
    String description;
    float price;

    public StockItem(int count, String description,
                     float price){
        this.count = count;
        this.description = description;
        this.price = price;
    }

    public String toString(){
        return ("Quantity: "+count+" Description: "+
                description + " Price: $"+price);
    }
}
```

Bridge Pattern

Code Sample – Client.java

```
public class Client
{
    Client(Report report)
    {
        System.out.println(report.report());
    }

    public static void main(String args[])
    {
        StockListReport report = new StockListReport(
                                 new HTMLReporter());
        report.addStockItem(new StockItem(
                    10,"Mountain Bike",999.99f));
        report.addStockItem(new StockItem(
                    15,"Helmet",29.99f));
        new Client(report);
        System.out.println();
        report = new StockListReport(new
   ASCIIReporter());
        report.addStockItem(new StockItem(
                    10,"Mountain Bike",999.99f));
        report.addStockItem(new StockItem(
                    15,"Helmet",29.99f));
        new Client(report);
    }
}
```

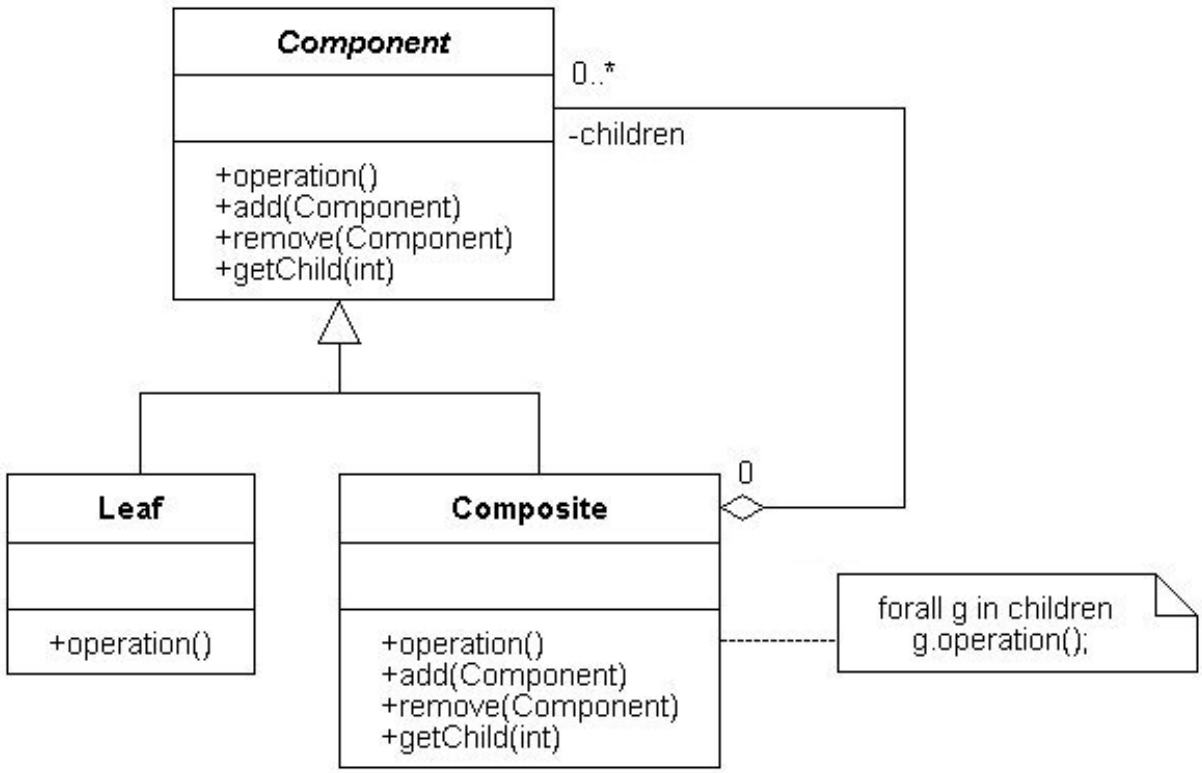
Page: 129

Composite Pattern

- The composite pattern is used to provide a uniform interface to components of a tree structure regardless of the components location in the tree.
- This allows one interface to be used to add a component to a container (Composite) or to add a container to a container.
- There are two fundamental variations on the pattern.
  - The first, as presented in the GOF book show the Composite class as a subclass of a Component with the Component defining the interface for the tree management methods.
  - The Composite class maintains a list of all of its components as well as implements the tree management methods.
  - Leaf Components are concrete components with no ability to contain other components.

Composite Pattern

Composite Class Diagram



Page: 131

Sample Code – Component.java

```
public abstract class Component
{
    public abstract void operation();
    public void add(Component component){};
    public void remove(Component component){};
}
```

Composite Pattern

Sample Code – Leaf.java

```
public class Leaf extends Component
{
    String name;

    public Leaf(String name)
    {
        this.name = name;
    }

    public void operation()
    {
        System.out.println(name);
    }
}
```

Sample Code – Component.java

```
import java.util.*;
public class Composite extends Component
{
    String name;
    ArrayList children = new ArrayList();
    public Composite(String name){
        this.name = name;
    }

    public void add(Component component){
        children.add(component);
    }

    public void remove(Component component){
        children.remove(component);
    }

    public void operation(){
        System.out.println(name);
        Iterator iterator = children.iterator();
        while(iterator.hasNext()){
            Component child =
                 (Component)iterator.next();
            child.operation();
        }
    }
}
```

Composite Pattern

Sample Code – Client.java

```
public class Client
{
    public static void main(String args[])
    {
        Composite otto = new Composite("Otto");
        Composite henry = new Composite("Henry");
        otto.add(henry);
        otto.add(new Leaf("Horst"));
        Composite barry = new Composite("Barry");
        Composite gary = new Composite("Gary");
        henry.add(gary);
        henry.add(barry);
        gary.add(new Leaf("Ethan"));
        barry.add(new Leaf("Patrick"));
        barry.add(new Leaf("Joseph"));
        otto.operation();
    }
}
```

Composite Pattern

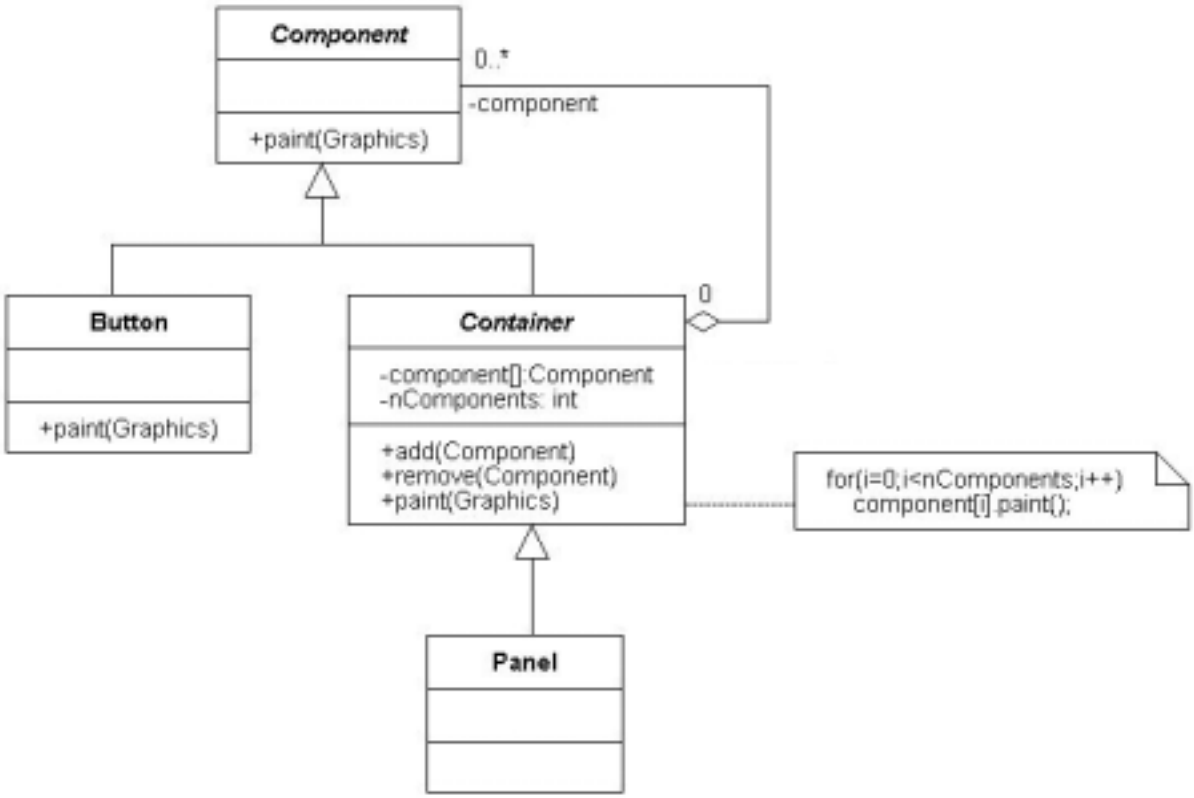
- The GOF pattern is very flexible, but does not provide much type safety. Compilers have no way of differentiating between Composites and Leafs.
- Adding a Leaf to a Leaf is clearly illegal, but the pattern would allow it at compile time since Leaf has inherited the methods from Component.
  - If the tree methods in Component are abstract, then the leaf will need to provide a dummy implementation that returns either some meaning full data or throws an exception.
  - The Component could provide a default implementation that does the same as above.

Composite Pattern

- An alternative is to move the definition of the tree methods down to the Composite class.
- This make the overall pattern a bit less generic, but provides a strong degree of type safety.
- All classes in the tree can be considered Components, but only classes that extend the Composite subclass can be containers.
- This is exactly how the Java awt is designed.
  - Buttons, text fields etc are derived off of Component.
  - Container derives off of component and has tree methods.
  - Panels derive off of Container.

Composite Pattern

java.awt.Container Class Diagram



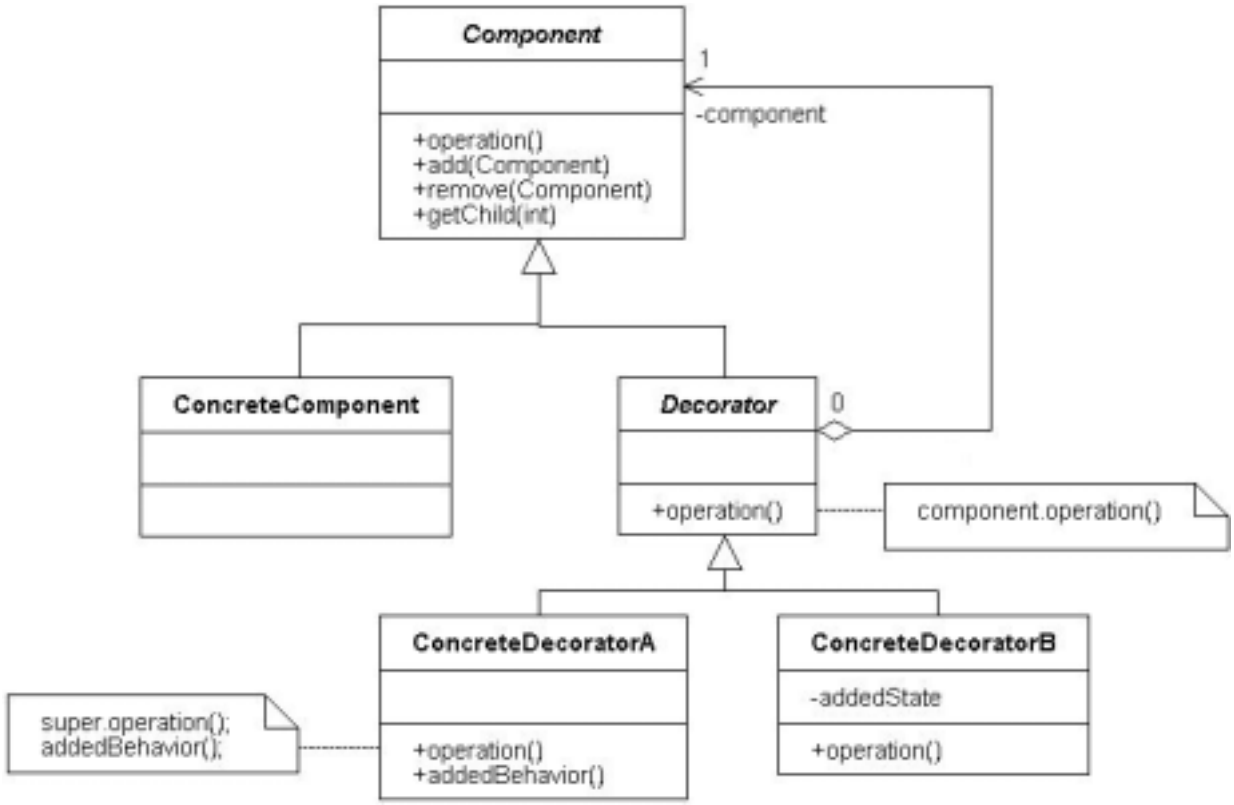
Page: 138

Decorator Pattern

- The decorator pattern attaches additional responsibility to an object dynamically.
- A Decorator is an abstract class which itself subclasses the base class of the component it is decorating.
  - Alternatively, it could implement the base interface.
- The Decorator will contain an instance of the component it is decorating all methods on the decorator will simply call the corresponding method on the contained object.
- Concrete subclasses of the decorator will now extend the behavior by overriding the needed methods. To maintain the existing behavior, the concrete class should still call the same the method on its parent class.

Decorator Pattern

Decorator Pattern Class Diagram



Page: 140

Java Sample – TextComponent.java

```
public abstract class TextComponent {
    protected String textValue;

    public void setText(String textValue) {
        this.textValue = textValue;
    }

    public String getText() {
        return this.textValue;
    }

    public abstract void displayText();
}
```

Java Sample – StdoutTextComponent.java

```
public class StdoutTextComponent extends TextComponent
{
    public void displayText() {
        System.out.println(textValue);
    }
}
```

Decorator Pattern

Java Sample –TextComponentDecorator.java

```
public abstract class TextComponentDecorator extends
    TextComponent {
    protected TextComponent textComponent;

    public void setTextComponent(TextComponent textComponent)
    {
        this.textComponent = textComponent;
    }

    public void displayText() {
        textComponent.displayText();
    }

    public String getText() {
        return textComponent.getText();
    }

    public void setText(String textValue) {
        textComponent.setText(textValue);
    }
}
```

Decorator Pattern

Java Sample –HTMLTextComponentDecorator.java

```
public class HTMLTextComponentDecorator extends
   TextComponentDecorator {

    public void displayText() {
        String oldText = getText();
        setText("<STRONG>"+ getText() + "</STRONG>");
        super.displayText();
        setText(oldText);
    }
}
```

Decorator Pattern

Java Sample –DecoratorExample.java

```
public class DecoratorExample {

    public DecoratorExample(TextComponent textComponent) {
        textComponent.displayText();
    }


    public static void main(String args[]) {
        StdoutTextComponent textComponent1 =
            new StdoutTextComponent();
        textComponent1.setText("Hello, world");
        new DecoratorExample(textComponent1);
        HTMLTextComponentDecorator htcd =
            new HTMLTextComponentDecorator();
        htcd.setTextComponent(textComponent1);
        new DecoratorExample(htcd);
    }
}
```

Page: 145

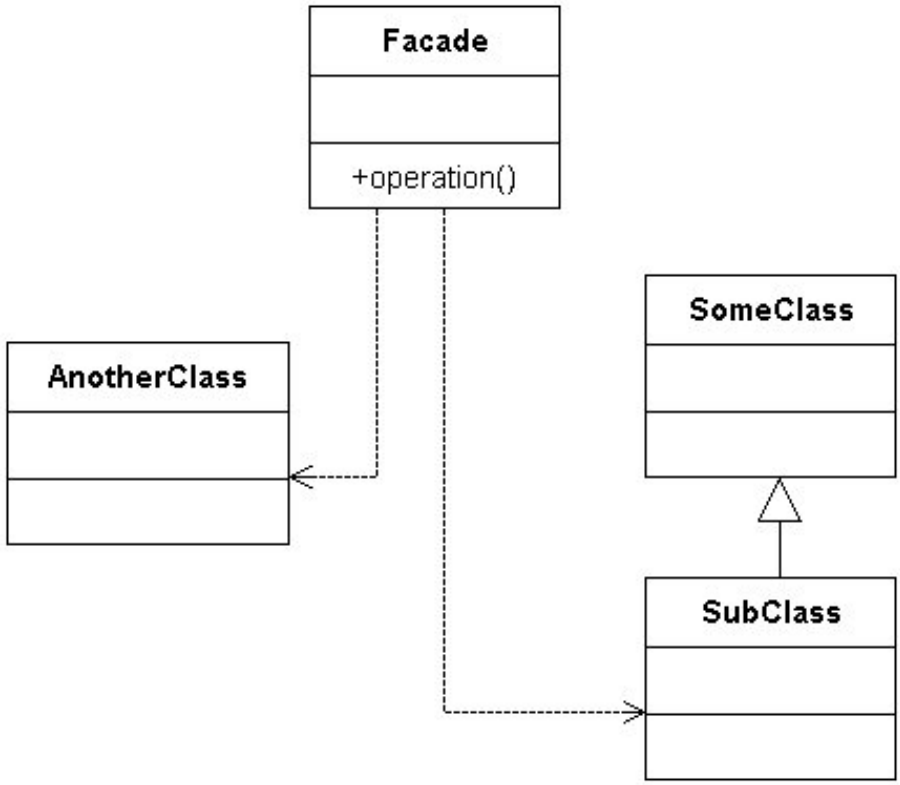
Facade Pattern

- The facade pattern allows you to provide an interface to a set of interfaces or classes.
- This is the most generic way of linking classes.
- The writer of the facade designs an interface and provides a class whose methods know how to call into the other classes.
- Clients written to use the façade have no knowledge of the classes behind the facade.
- This is a great way to provide an interface to a legacy system or to classes that have no public interface.
- A facade may be a container of the back end classes as well.

Page: 146

Facade Pattern

Facade Pattern Class Diagram



Page: 147

Facade Pattern

Code Sample – Asset.java

```
public interface Asset
{
    public String getAssetDescription();
    public int getAssetNumber();
}
```

Code Sample – Employee.java

```
public interface Employee
{
    public String getName();
    public float getSalary();
}
```

Facade Pattern

Code Sample – StaffEmployee.java

```
public class StaffEmployee implements Employee {
    private String name;
    private float salary;

    StaffEmployee(String name, float salary){
        this.name = name;
        this.salary = salary;
    }
    public String getName(){
        return name;
    }
    public float getSalary(){
        return salary;
    }
    public String toString(){
        return ("Name: "+getName()+
                " Salary: "+getSalary());
    }
}
```

Page: 149

Facade Pattern

Code Sample – CompanyAsset.java

```
public class CompanyAsset implements Asset{
    String description;
    int number;

    public CompanyAsset(String desc,int num){
        this.description = desc;
        this.number = num;
    }
    public String getAssetDescription(){
        return description;
    }
    public int getAssetNumber(){
        return number;
    }
    public String toString(){
        return ("Description: "+getAssetDescription()+
                " Number: "+getAssetNumber());
    }
}
```

Page: 150

Facade Pattern

Code Sample – AssetFactory.java

```
import java.util.*;

public class AssetFactory
{
    static HashMap assets = null;
    private AssetFactory() {}
    static Map getAssets(){
      if ( assets == null ) {
       assets = new HashMap();
       assets.put("Pencil",new CompanyAsset("Pencil",1));
       assets.put("Pen",new CompanyAsset("Pen",2));
       assets.put("Chair",new CompanyAsset("Chair",3));
      }
      return assets;
    }
}
```

Page: 151

Facade Pattern

Code Sample – EmployeeFactory.java

```
import java.util.*;
public class EmployeeFactory
{
  static Map employees = null;
  private EmployeeFactory() {}

  static Map getEmployees(){
    if ( employees == null ){
            employees = new HashMap();
      employees.put("Barry",
                      new StaffEmployee("Barry",20.0f));
      employees.put("Tom",
                      new StaffEmployee("Tom",25.0f));
      employees.put("Brian",
                      new StaffEmployee("Brian",30.0f));
    }
    return employees;
  }
}
```

Page: 152

Facade Pattern

Code Sample – CompanyFacade.java

```
import java.util.*;
public class CompanyFacade{
    Map assets;
    Map employees;

    public CompanyFacade()
    {
        assets = AssetFactory.getAssets();
        employees = EmployeeFactory.getEmployees();
    }

    public String companyReport(){
        StringBuffer sb = new StringBuffer();

        sb.append("Assets\n");
        Iterator iterator = assets.values().iterator();
        while (iterator.hasNext())
        {
            Asset asset = (Asset)iterator.next();
            sb.append(asset.toString()+"\n");
        }
        sb.append("Employees\n");
        iterator = employees.values().iterator();
        while (iterator.hasNext())
        {
            Employee emp = (Employee)iterator.next();
            sb.append(emp.toString()+"\n");
        }
        return sb.toString();
    }
   // continued on next slide
```

Page: 153

Code Sample – CompanyFacade.java (cont)

```
    public String getEmployeeInfo(String name)
    {
        Employee emp = (Employee) employees.get(name);
        return emp.toString();
    }
    public String getAssetInfo(String name)
    {
        Asset asset = (Asset) assets.get(name);
        return asset.toString();
    }
}
```

Code Sample – Client.java

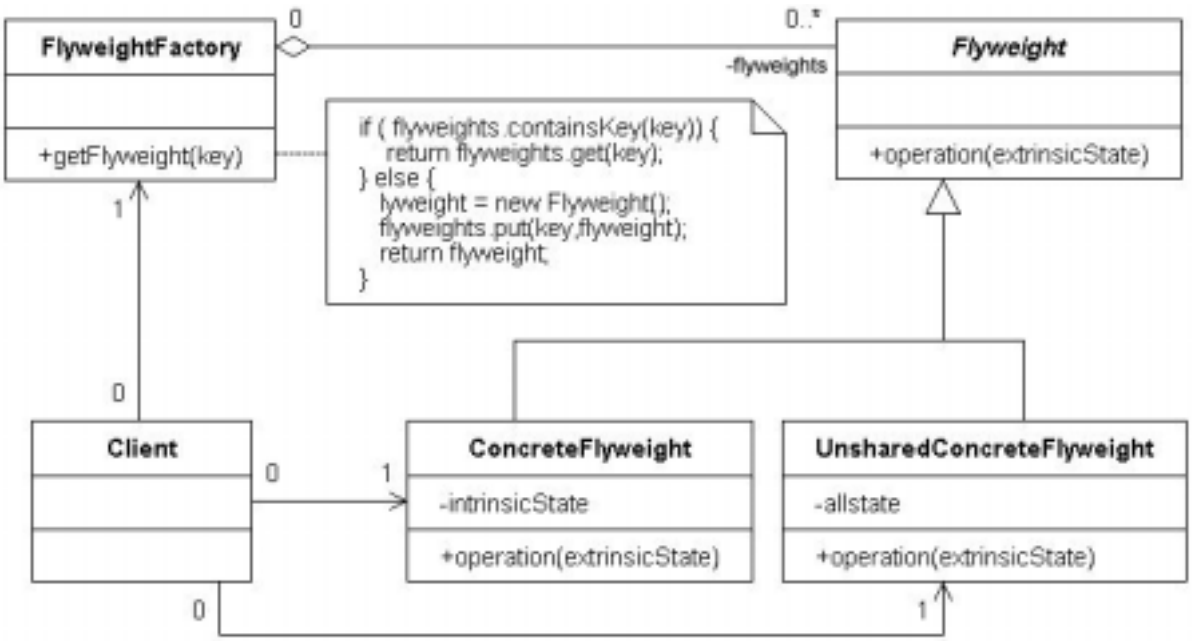
```
public class Client
{
    public static void main(String args[])
    {
        CompanyFacade company = new CompanyFacade();
        System.out.println(
                company.getEmployeeInfo("Barry"));
        System.out.println(
                company.getAssetInfo("Chair"));
        System.out.println(
                company.companyReport());
    }
}
```

Flyweight Pattern

- The flyweight pattern is used to maintain a large numbers of fine grade objects. Normally these objects are shared.
- Very useful for objects whose state does not change after being instantiated.
- Uses a FlyweightFactory to create and manage Flyweight objects.
- Flyweight objects are uniquely identified (using a key of some sort). Whenever the factory is asked to get an object with the associated key, it either creates a new one, or, if it already exists, uses one from the pool.

Flyweight Pattern

Flyweight Pattern Class Diagram



Page: 157

Code Sample – Number.java

```
public abstract class Number
{
    int number;

    public Number(int number)
    {
       this.number = number;
    }

    public abstract int multiply(int number);
    public int getNumber()
    {
        return number;
    }
}
```

Code Sample – ImmutableNumber.java

```
public class ImmutableNumber extends Number
{
    ImmutableNumber(int number)
    {
        super(number);
    }

    public int multiply(int number)
    {
        return(this.number * number);
    }
}
```

Code Sample – MutableNumber.java

```
public class MutableNumber extends Number
{
    MutableNumber(int number)
    {
        super(number);
    }

    public int multiply(int number)
    {
        this.number *= number;
        return(number);
    }
}
```

Page: 160

Flyweight Pattern

Code Sample – NumberFactory.java

```
import java.util.*;
public class NumberFactory
{
    Number numbers[] = new Number[100];
    public Number getImmutableNumber(int number)
    {
        if ( numbers[number] != null )
            return ( numbers[number] );
        Number aNumber = new ImmutableNumber(number);
        numbers[number] = aNumber;
        return aNumber;
    }
    public Number getMutableNumber(int number)
    {
        return new MutableNumber(number);
    }
}
```

Page: 161

Flyweight Pattern

Code Sample – Client.java

```
public class Client
{
    Client(Number number){
      System.out.println("Num: "+number.getNumber());
      System.out.println("Mult: "+number.multiply(10));
      System.out.println("Final:"+number.getNumber());
    }
    public static void main(String args[]){
        NumberFactory nf = new NumberFactory();
        Number im1 = nf.getImmutableNumber(1);
        Number im2 = nf.getImmutableNumber(2);
        Number im3 = nf.getImmutableNumber(2);
        if ( im2 == im3 )
         System.out.println("Hey, they are the same!");
        Number mn1 = nf.getMutableNumber(1);
        if ( mn1 != im1 )
          System.out.println("The same but different");
        new Client(im1);
        new Client(im2);
        new Client(mn1);
        new Client(mn1);
    }
}
```

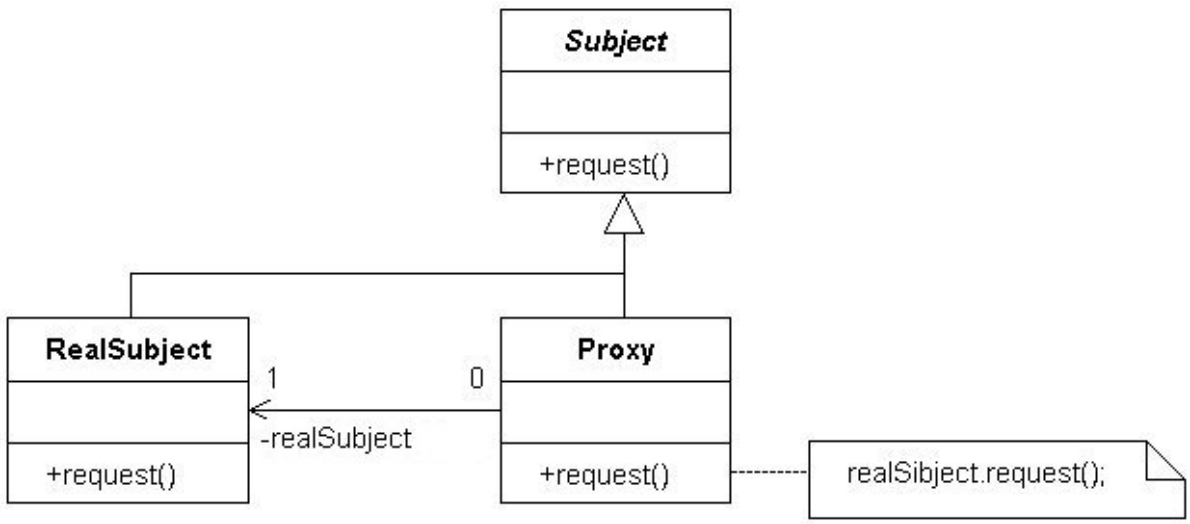
Page: 162

Proxy Pattern

- The proxy pattern is used to create a placeholder for another object to control access to it.
- The proxy class is a subclass of the same class as the object it is a placeholder for (RealSubject).
- The proxy maintains some reference to the RealSubject and will call methods on it.
- The methods in the proxy exactly matchthe methods in the RealSubject.
- Proxy's are often used in Java to provide the methods for remotely communicating with the realSubject
  - The RMI stub class is an example of a Proxy.

Proxy Pattern

Proxy Pattern Class Diagram



Page: 164

Sample Code – Employee.java

```
public interface Employee
{
    public String getName();
    public float getSalary();
}
```

Proxy Pattern

Sample Code – StaffEmployee.java

```
public class StaffEmployee implements Employee
{
    private String name;
    private float salary;

    StaffEmployee(String name, float salary){
        this.name = name;
        this.salary = salary;
    }

    public String getName(){
        return name;
    }

    public float getSalary(){
        return salary;
    }
}
```

Page: 166

Proxy Pattern

Sample Code – EmployeeProxy.java

```
public class EmployeeProxy implements Employee
{
    Employee realEmployee;
    boolean canReadSalary=false;

    EmployeeProxy(Employee realEmployee,
                  boolean canReadSalary){
        this.realEmployee = realEmployee;
        this.canReadSalary = canReadSalary;
    }
    public String getName(){
        return realEmployee.getName();
    }
    public float getSalary(){
        if ( canReadSalary )
            return realEmployee.getSalary();
        return 0.0f;
    }
}
```

Page: 167

Proxy Pattern

Sample Code – Client.java

```
public class Client
{
  Client(Employee emp){
     System.out.println("Name: "+ emp.getName());
     System.out.println("Salary: "+emp.getSalary());
  }

  public static void main(String args[]){
     Employee employee = new StaffEmployee(
                         "Barry L. Geipel",
                         20.00f);
     Employee ep = new EmployeeProxy(employee,true);
     new Client(ep);
     ep = new EmployeeProxy(employee,false);
     new Client(ep);
    }
}
```

Page: 168

Behavioral Patterns

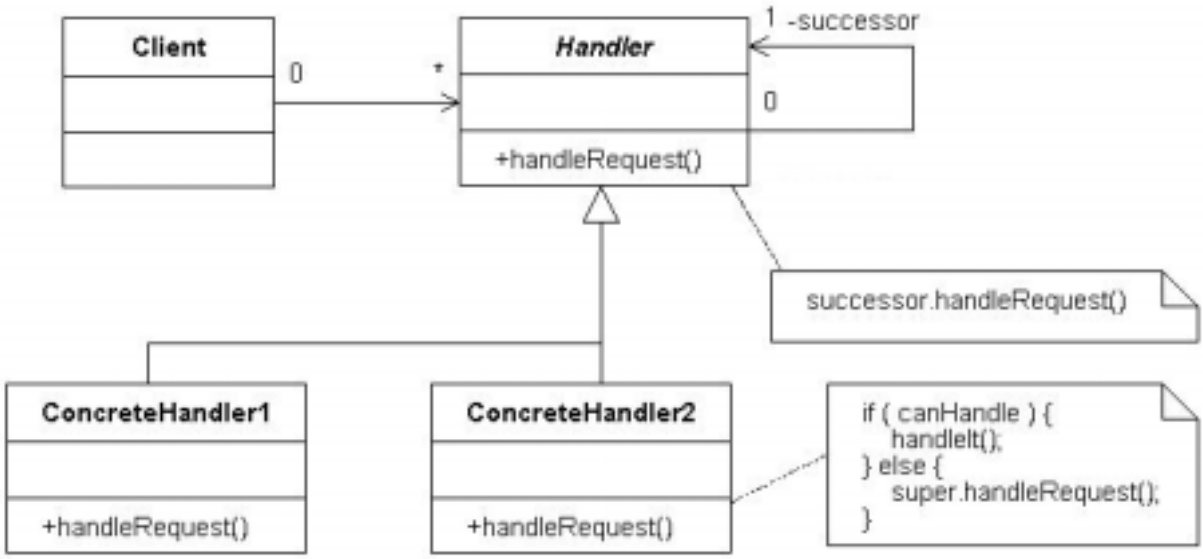
- Behavioral Patterns are used to define the communications between objects.
- The patterns themselves show the algorithms and the assignment of responsibilities.
- You will see the use of inheritance and composition in the behavioral patterns.
- We want to minimize the coupling between classes. All of the behavioral patterns are used to decouple classes.

Chain of Responsibility Pattern

- The Chain of Responsibility Pattern allow you to avoid coupling a sender of a request to is receiver by allowing intermediate objects a chance to handle the request.
- The sender has an instance of a concrete handler object which may or may not have the ability to handle a request.
- The handler will check to see if it know how to handle the request. If it does not, then it will defer handling to its super class.
- This process continues until the request is handled.

Chain of Responsibility Pattern

Chain of Responsibility Pattern Class Diagram

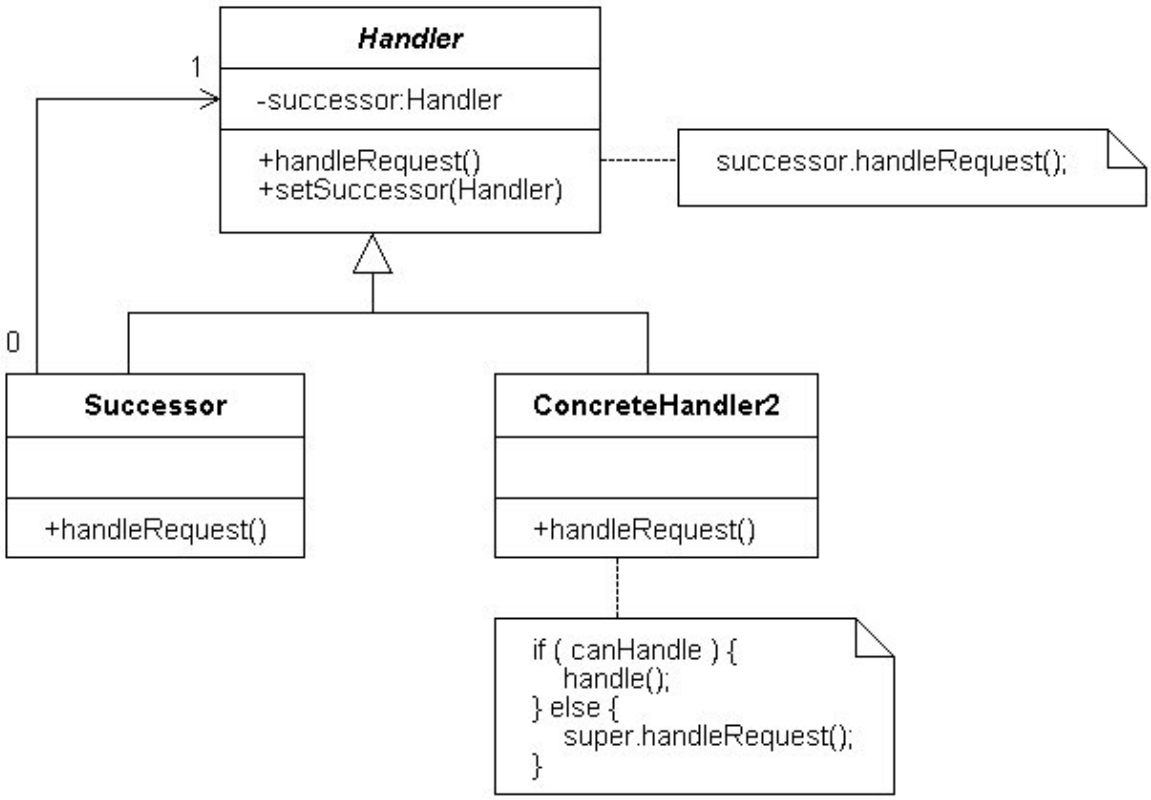


Page: 171

Chain of Responsibility Pattern

- A Client could register itself as a handler with the base class to ensure that the request is somehow handled.
- Actually, any Handler can register itself as a handler as seen in the following diagram.

Alternate Chain of Responsibility Pattern Class Diagram



Page: 173

Chain of Responsibility Pattern

Code Sample – HelpHandler.java

```
public abstract class HelpHandler
{
    HelpHandler handler=null;
    public void handleHelp()
    {
        if ( handler != null )
        {
            handler.handleHelp();
        }
    }
    public void setHandler(HelpHandler handler)
    {
        this.handler = handler;
    }
}
```

Chain of Responsibility Pattern

Code Sample – HalpHandler.java

```
public class Widget extends HelpHandler
{
    String help;
    String name;
    public Widget(String name){
        this.name = name;
    }
    public void setHelp(String help){
        this.help = help;
    }
    public void handleHelp(){
        if ( help != null ){
            System.out.println(name + ": "+help);
        }
        else{
            super.handleHelp();
        }
    }
}
```

Chain of Responsibility Pattern

Code Sample – Client.java

```
public class Client extends HelpHandler
{
    private Widget widget1;
    private Widget widget2;
    private Widget widget3;
    public Client(){
        widget1 = new Widget("Widget 1");
        widget2 = new Widget("Widget 2");
        widget1.setHandler(widget2);
        widget3 = new Widget("Widget 3");
        widget2.setHandler(widget3);
        widget3.setHandler(this);
    }
    public void handleHelp(){
        System.out.println("No help available");
    }
    public static void main(String args[]){
        Client client = new Client();
        client.widget3.setHelp("Read the manual");
        client.widget1.handleHelp();
        client = new Client();
        client.widget1.handleHelp();
    }
}
```

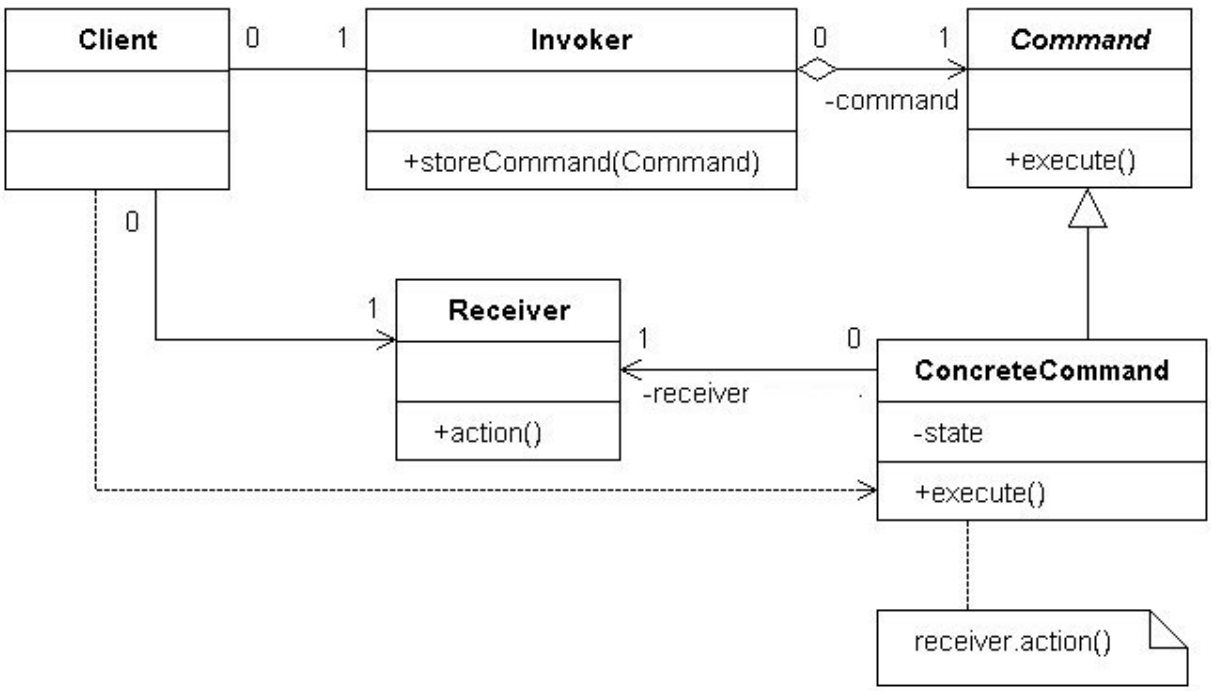
Page: 176

Command Pattern

- The Command Pattern encapsulates a request in an object which can be registered with another class.
- The coupling between the client, the request (Command) and the invoker is very loose.
- The client creates a command and registers it with the invoker.
- When the invoker needs to use the command, it calls the appropriate method.
- In the pattern, a Receiver object contains the bulk of the real code. The ConcreteCommand mearly acts as a facade to the Receiver.
- Similar to Observer but operates on only one Command object.

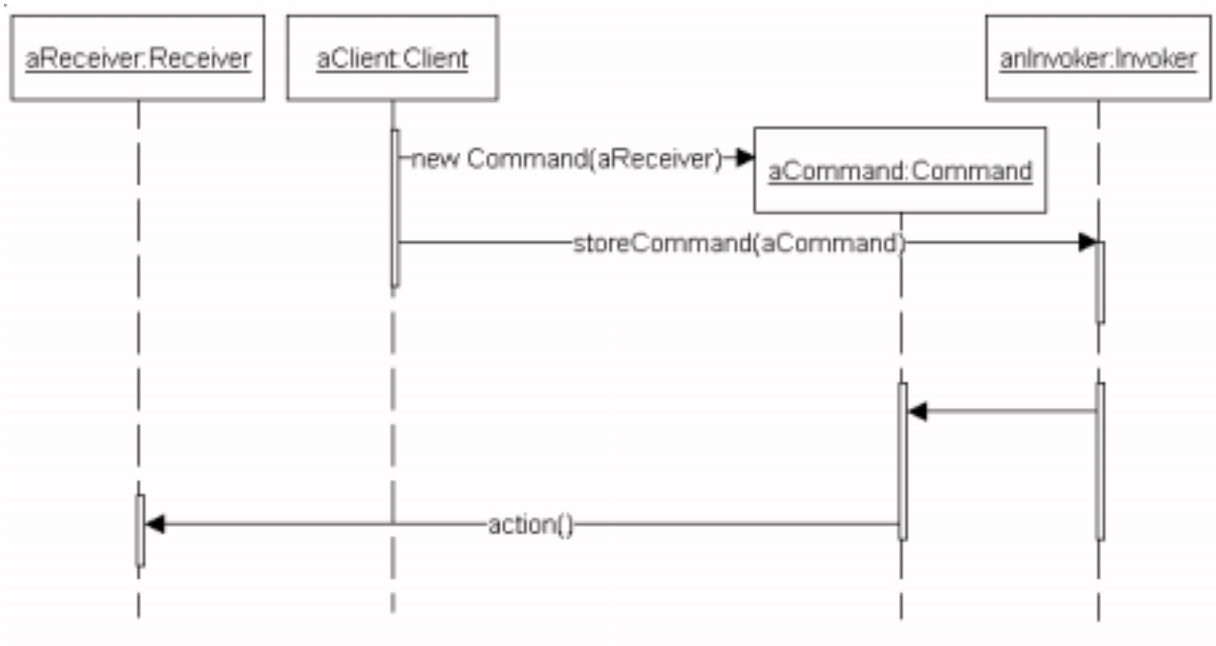
Command Pattern

Command Pattern Class Diagram



Command Pattern

Command Pattern Sequence Diagram



Command Pattern

Code Sample – Invoker.java

```
public class Invoker
{
    Command command;

    public void storeCommand(Command command)
    {
        this.command = command;
    }

    public void operation()
    {
        command.execute();
    }
}
```

Command Pattern

Code Sample – Command.java

```
public abstract class Command
{
    public abstract void execute();
}
```

Code Sample – ConcreteCommand.java

```
public class ConcreteCommand extends Command
{
    Receiver receiver;

    ConcreteCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    public void execute()
    {
        receiver.action();
    }
}
```

Page: 181

Command Pattern

Code Sample – Receiver.java

```
public interface Receiver
{
    public void action();
}
```

Code Sample – ConcreteReceiver.java

```
public class ConcreteReceiver implements Receiver
{
    private String name;
    public ConcreteReceiver(String name) {
        this.name = name;
    }
    public void action() {
        System.out.println(name+
                 " is handling this action");
    }
}
```

Command Pattern

Code Sample – Client.java

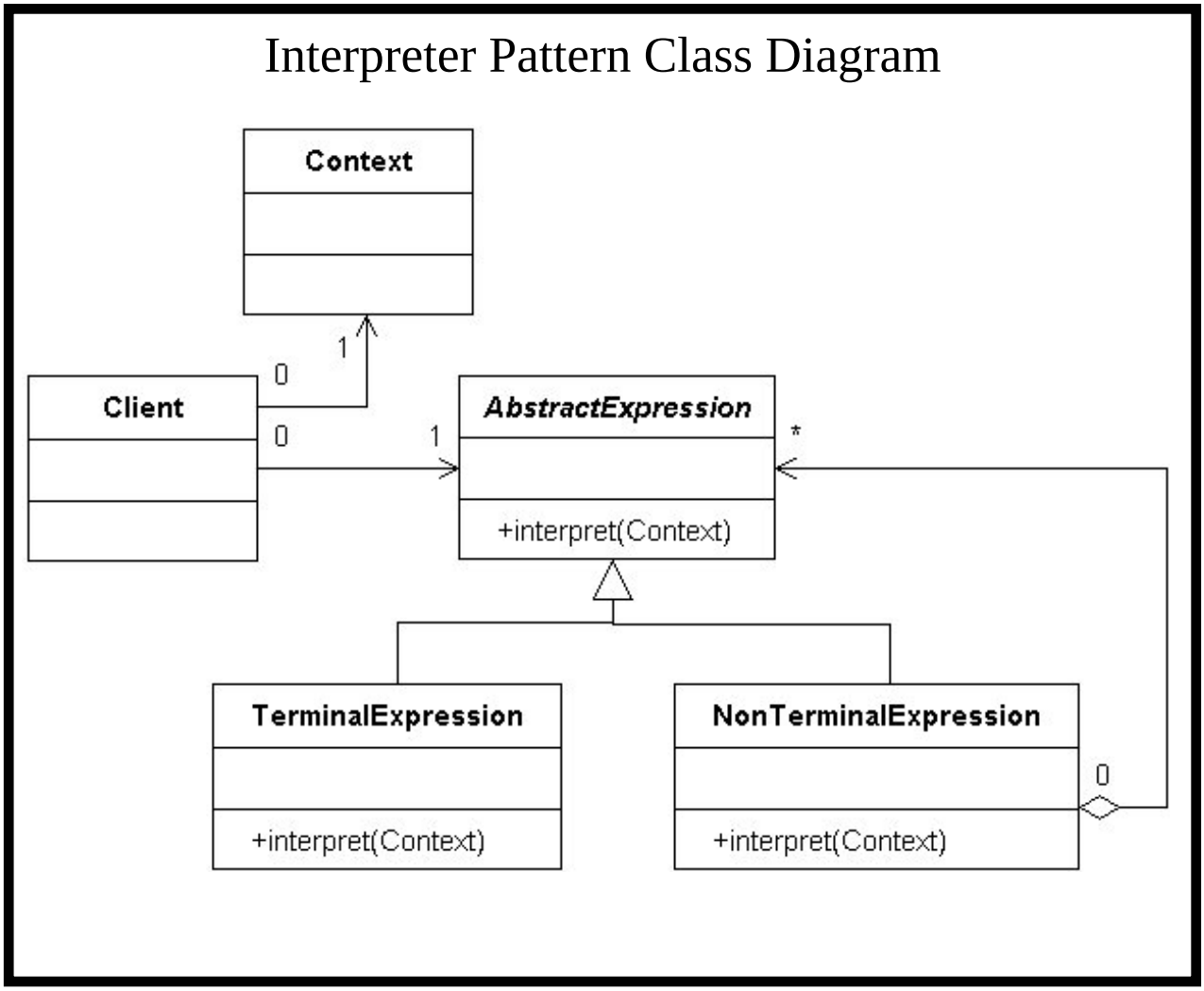
```
public class Client
{
    public Client(Invoker invoker)
    {
        invoker.operation();
    }

    public static void main(String args[])
    {
        Invoker invoker = new Invoker();
        Receiver receiver = new ConcreteReceiver(
                            "Barry");
        Command command = new ConcreteCommand(
                            receiver);
        invoker.storeCommand(command);
        new Client(invoker);
    }
}
```

Page: 183

Interpreter Pattern

- The interpreter pattern is a representation of language expressions.
- Languages are formal definitions of some kind of syntax. For example:
  - Command line input
  - Mathematical Expressions
  - Report Generators
- The expressions are parsed and instantiated as either a TerminalExpression (one that needs no further processing) or a NonterminalExpression (one which needs further processing.
- Conceptually, TerminalExpressions are leafs and NonterminalExpresions are Nodes.
- NonterminalExpressions are containers of other AbstractExpressions.

Interpreter Pattern



Page: 185

Interpreter Pattern

- For the following code sample, we will build a tree based on the following BNF for interpreting boolean expressions:

BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
‘(‘ BooleanExp ‘)’

AndExp ::= BooleanExp ‘and’ BooleanExp

OrExp ::= BooleanExp ‘or’ BooleanExp

NotExp ::= ‘not’ BooleanExp

Constant :: ‘true’ | ‘false’

VariableExp ::= ‘A’ | ‘B’ | … | ‘X’ | ‘Y’ | ‘Z’

- BooleanExp is the AbstractExpression
- AndExp, OrExp and NotExp are all NonTerminalExpressions
- Constant and VariableExp are TerminalExpresssions

Code Sample – BooleanExp.java

```
public abstract class BooleanExp
{
    public abstract boolean evaluate(
                        Context context);
}
```

Code Sample – Context.java

```
import java.util.HashMap;

public class Context
{
    private HashMap context = new HashMap();

    public boolean lookup(String var){
        Boolean tmp = (Boolean) context.get(var);
        return ( tmp.booleanValue());
    }

    public void assign(VariableExp varExp,
                       boolean value){
        context.put(varExp.getName(),
                    new Boolean(value));
    }
}
```

Page: 188

Interpreter Pattern

Code Sample – AndExp.java

```
public class AndExp extends BooleanExp
{
    private BooleanExp operand1;
    private BooleanExp operand2;

    public AndExp(BooleanExp operand1,
                  BooleanExp operand2)
    {
        this.operand1 = operand1;
        this.operand2 = operand2;
    }
    public boolean evaluate(Context context)
    {
        return (operand1.evaluate(context) &&
                operand2.evaluate(context));
    }
}
```

Page: 189

Code Sample – NotExp.java

```
public class NotExp extends BooleanExp
{
    private BooleanExp operand;

    public NotExp(BooleanExp operand)
    {
        this.operand = operand;
    }

    public boolean evaluate(Context context)
    {
        return !operand.evaluate(context);
    }
}
```

Page: 190

Interpreter Pattern

Code Sample – OrExp.java

```
public class OrExp extends BooleanExp
{
    private BooleanExp operand1;
    private BooleanExp operand2;

    public OrExp(BooleanExp operand1,
                 BooleanExp operand2)
    {
        this.operand1 = operand1;
        this.operand2 = operand2;
    }
    public boolean evaluate(Context context)
    {
        return (operand1.evaluate(context) ||
                operand2.evaluate(context));
    }
}
```

Page: 191

Interpreter Pattern

Code Sample – VariableExp.java

```
public  class VariableExp extends BooleanExp
{
    private String name;

    public VariableExp(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
    public boolean evaluate(Context context)
    {
        return context.lookup(name);
    }
}
```

Page: 192

Code Sample – Constant.java

```
public class Constant extends BooleanExp
{
    private boolean constVal;

    public Constant(boolean constVal)
    {
        this.constVal = constVal;
    }

    public boolean evaluate(Context context)
    {
        return constVal;
    }
}
```

Page: 193

Interpreter Pattern

Code Sample – Client.java

```
public class Client
{
    public static void main(String args[]){
        // (true and x) or (y and (not x))
        BooleanExp expression;
        Context context = new Context();
        VariableExp x = new VariableExp("X");
        VariableExp y = new VariableExp("Y");

        expression = new OrExp(
            new AndExp(new Constant(true), x),
            new AndExp(y, new NotExp(x)));

        context.assign(x, false);
        context.assign(y, true);
        System.out.println("Result “+
          expression.evaluate(context));
   }
}
```

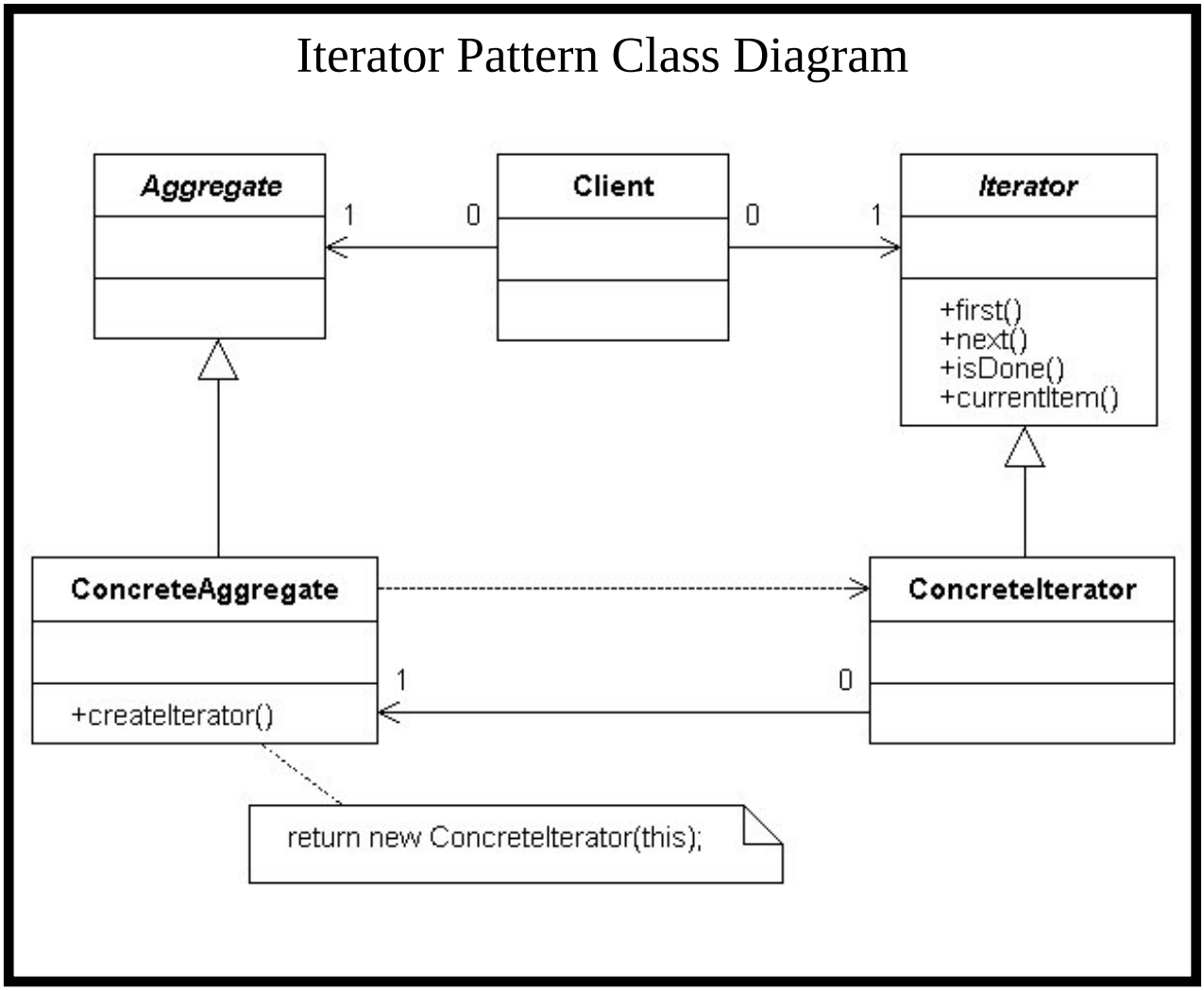
Page: 194

Iterator Pattern

- The Iterator Pattern provides a way to access the elements of an aggregate object in some sequence without exposing any details about the aggregates underlying representation.
- A client can add objects to the aggregate.
- The client then asks the aggregate for its Iterator Object.
- Using the iterator object, the client can then get members of the aggregate.
- The iterator can define the order that objects are retrieved.
- Since the iterator and the aggregate are tightly coupled, the iterator can impose security rules based on modification of the underlying aggregate.

Page: 195

Iterator Pattern



Page: 196

Code Sample – Aggregate.java

```
public abstract class Aggregate {
    public abstract Iterator createIterator();
    public abstract void add(Object o);
}
```

Code Sample – ConcreteAggregate.java

```
public class ConcreteAggregate extends Aggregate {
    java.util.Vector storage;

    public ConcreteAggregate() {
        storage = new java.util.Vector();
    }

    public Iterator createIterator() {
        return new ConcreteIterator(this);
    }

    public void add(Object o) {
        storage.add(o);
    }
}
```

Code Sample – Iterator.java

```
public abstract class Iterator {
    public abstract Object first();
    public abstract Object next();
    public abstract boolean isDone();
    public abstract Object currentItem();
}
```

Iterator Pattern

Code Sample – ConcreteIterator.java

```
public class ConcreteIterator extends Iterator {
    private ConcreteAggregate ca;
    private int currentIndex = -1;

    ConcreteIterator(ConcreteAggregate ca) {
        this.ca = ca;
    }
    public Object first() {
        return ca.storage.elementAt(0);
    }
    public Object next() {
        return ca.storage.elementAt(++currentIndex);
    }
    public boolean isDone() {
        return ( currentIndex >= ca.storage.size() -1 );
    }
    public Object currentItem() {
        return ca.storage.elementAt(currentIndex);
    }
}
```

Page: 200

Code Sample – IteratorClient.java

```
public class IteratorClient {

    public IteratorClient(Aggregate aggregate) {
       Iterator iterator = aggregate.createIterator();
       while ( !iterator.isDone() ) {
        System.out.print((String)iterator.next());
       }
       System.out.println();
    }

    public static void main(String args[]) {
       ConcreteAggregate ca = new ConcreteAggregate();
       ca.add(new String("Hello, "));
       ca.add(new String("world"));
       new IteratorClient(ca);
    }
}
```

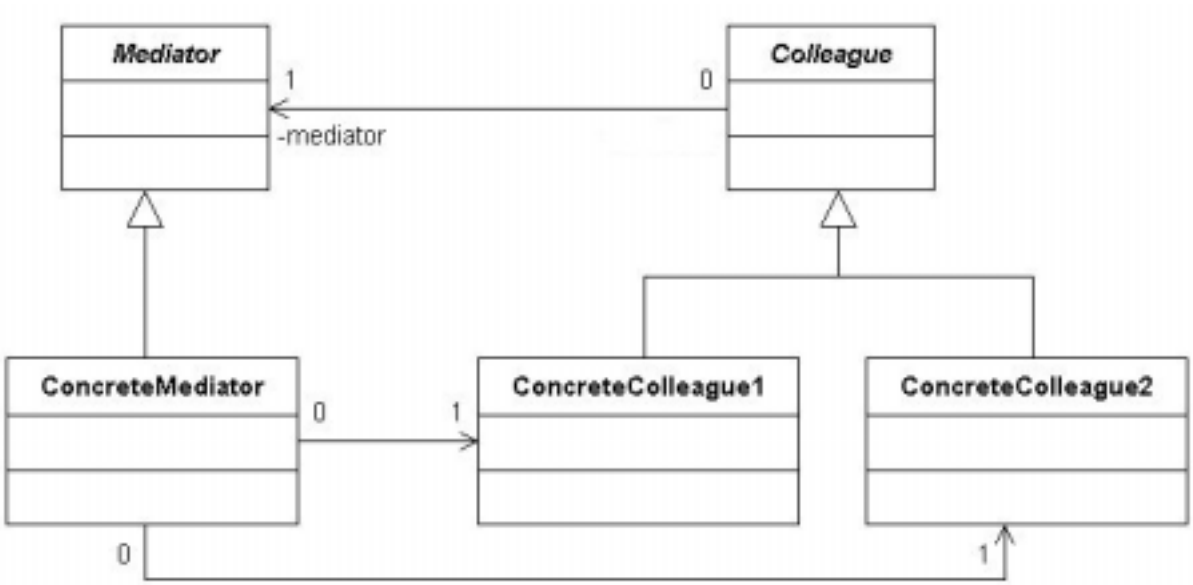
Mediator Pattern

- The Mediator Pattern is used to define an object which will encapsulate how a set of objects interact.
- When many objects interact with many other objects, then the coupling becomes very tight.
- To avoid this, the Mediator Pattern allows you to encapsulate the code for the interactions into a mediator class.
- The interfaces to the mediator remain constant. But the implementation inside of the mediator is open to change.
- Mediators are a common technique in implementing GUI's to keep the functionality and the rendering separate.

Page: 202

Mediator Pattern

Mediator Pattern Class Diagram



Page: 203

Code Sample – Widget.java

```
public abstract class Widget
{
    WidgetDirector director;

    public void setDirector(WidgetDirector director)
    {
        this.director = director;
    }

    public void changed()
    {
        director.widgetChanged(this);
    }
}
```

Code Sample – DisplayWidget.java

```
public class DisplayWidget extends Widget
{
    private String text;

    public void setText(String text)
    {
        System.out.println("Display> "+text);
        this.text = text;
    }
}
```

Mediator Pattern

Code Sample – EntryWidget.java

```
import java.io.*;
public class EntryWidget extends Widget implements Runnable {
    private String value;

    public EntryWidget(){
        Thread thread = new Thread(this);
        thread.start();
    }

    public void run(){
        try {
            BufferedReader br = new BufferedReader(
                                new InputStreamReader(
                                System.in));
            while ( true ) {
                System.out.print("Enter Data: ");
                value = br.readLine();
                changed();
            }
        }
        catch (IOException ioe){
            System.out.println(ioe.getMessage());
        }
    }

    public String getValue(){
        return value;
    }
}
```

Page: 206

Code Sample – WidgetDirector.java

```
public abstract class WidgetDirector
{
    public abstract void widgetChanged(Widget widget);
}
```

Mediator Pattern

Code Sample – IOWidgetDirector.java

```
public class IOWidgetDirector extends WidgetDirector
{
    private DisplayWidget dw;
    private EntryWidget ew;

    public IOWidgetDirector(DisplayWidget dw,
                            EntryWidget ew)
    {
        this.dw = dw;
        this.ew = ew;
    }

    public void widgetChanged(Widget widget)
    {
        if ( widget == ew )
        {
            String value = ew.getValue();
            dw.setText(value);
        }
    }
}
```

Page: 208

Mediator Pattern

Code Sample – Client.java

```
public class Client
{
    public static void main(String args[])
    {
        EntryWidget ew = new EntryWidget();
        DisplayWidget dw = new DisplayWidget();
        IOWidgetDirector wd = new
                            IOWidgetDirector(dw,ew);
        ew.setDirector(wd);
    }
}
```

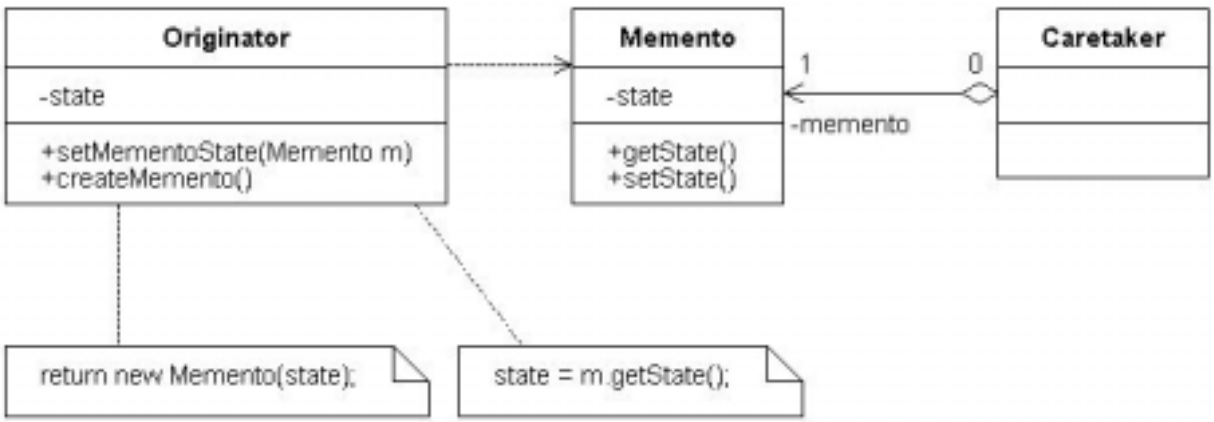
Memento Pattern

- The memento pattern is used to capture the internal state of an object so that it can be restored at a later time.
- The Originator is the object whose internal state we want to save.
- A Caretaker object becomes responsible for defining how the memento gets stored.
- Java supports serializable data, so the entire state of an object could be saved.
- But, if you are only interested in saving part of the state, you could create a memento which just represents the data you are interested in.
- In Java, you will need to be careful in how the data gets exposed so that the memento can build itself. Ideally, the internal data should be declared as package protected (the default) and the memento should be a class in the same package as the Originator.

Page: 210

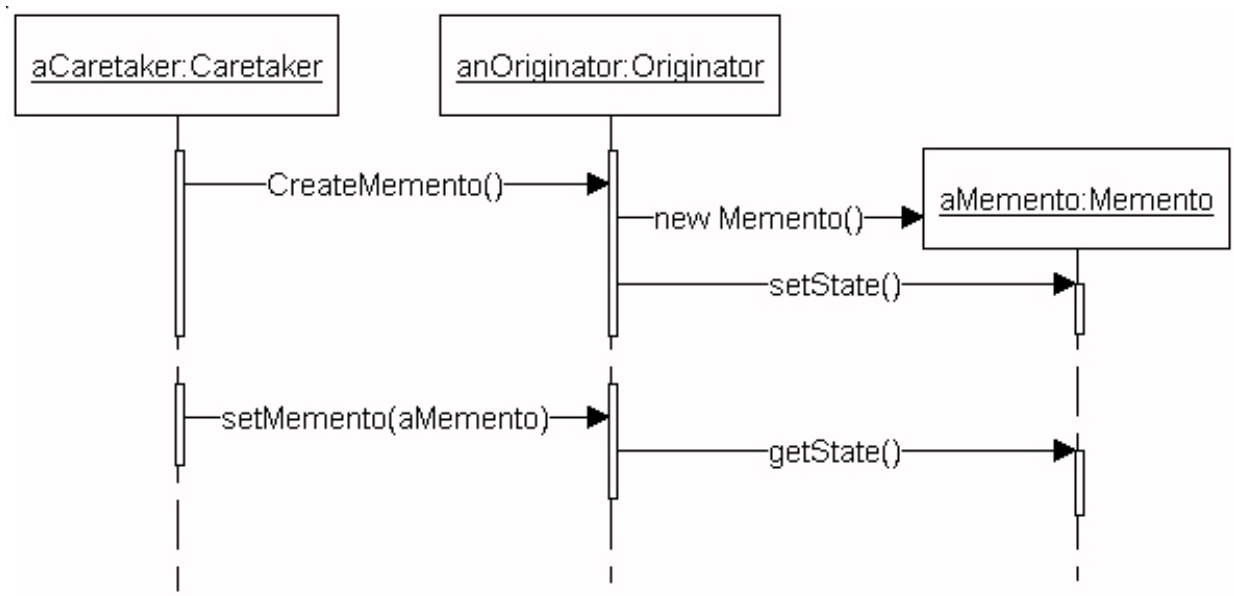
Memento Pattern

Memento Pattern Class Diagram



Memento Pattern

Memento Pattern Sequence Diagram



Page: 212

Memento Pattern

Code Sample – Originator.java

```
public class Originator
{
    private String name;
    private int value;

    public void setName(String name)
    {
        this.name = name;
    }
    public void setValue(int value)
    {
        this.value = value;
    }
    public String toString()
    {
        return name+": "+value;
    }

    // continued on next slide
```

Code Sample – Originator.java (cont.)

```
    public void setMemento(Memento m)
    {
        name = m.getName();
        value = m.getValue();
    }

    public Memento createMemento()
    {
        return new Memento(name,value);
    }
}
```

Memento Pattern

Code Sample – Memento.java

```
public class Memento
{
    String name;
    int value;

    public Memento(String name, int value)
    {
        this.name = name;
        this.value = value;
    }
    public String getName()
    {
        return name;
    }
    public int getValue()
    {
        return value;
    }
}
```

Page: 215

Memento Pattern

Code Sample – Client.java

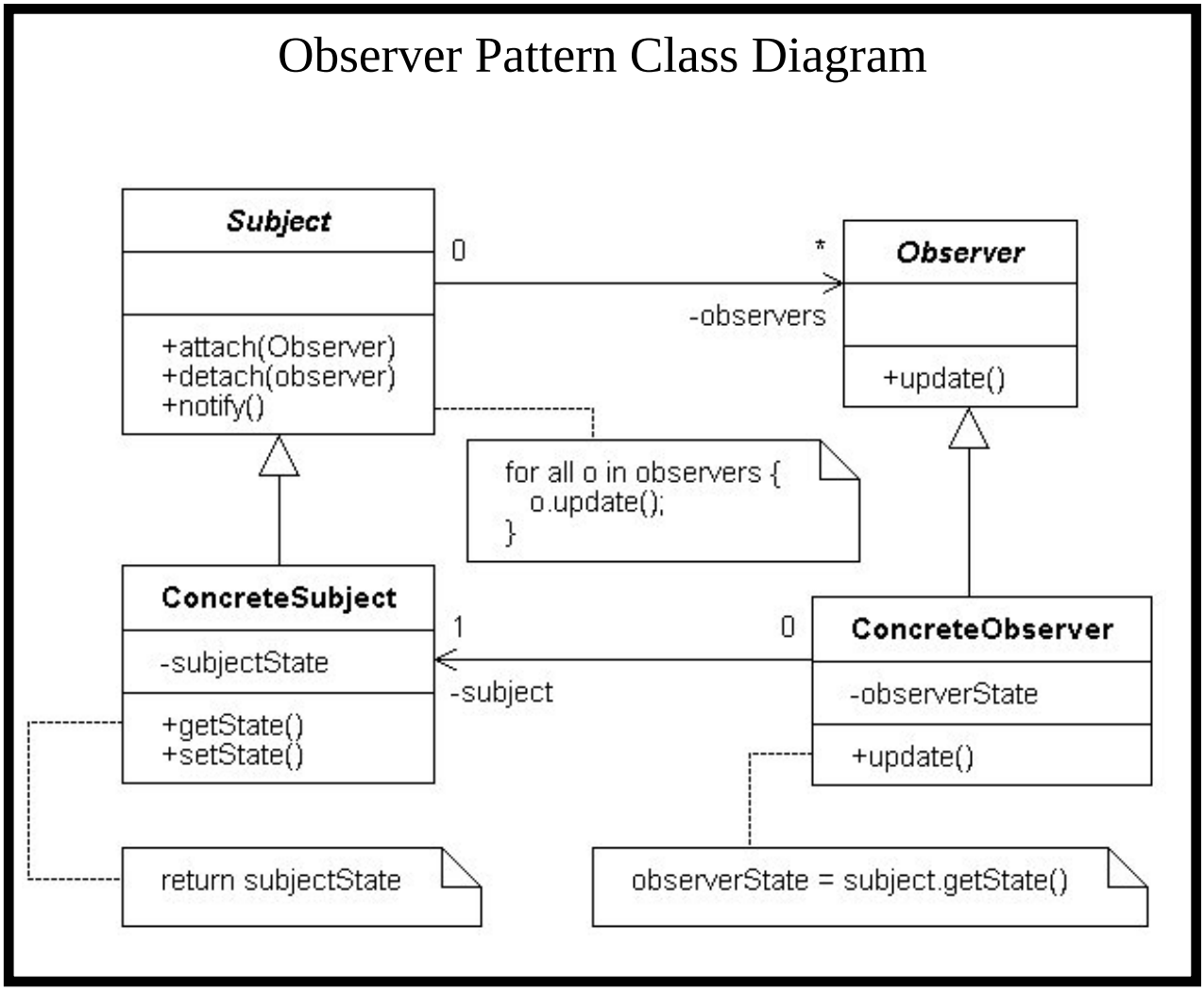
```
import java.util.*;
public class Client // acting as Caretaker
{
    public static void main(String args[])
    {
        ArrayList caretaker = new ArrayList();
        Originator o = new Originator();
        o.setName("Barry");
        o.setValue(10);
        caretaker.add(o.createMemento());
        System.out.println(o.toString());
        o.setValue(20);
        System.out.println(o.toString());
        o.setMemento((Memento)caretaker.get(
                          caretaker.size()-1));
        System.out.println(o.toString());
    }
}
```

Observer Pattern

- The Observer Pattern defines a one-to-many relationship between objects.
- One object, the Subject, keeps track of a collection of objects, Observers, and notifies the Observers whenever there is a change in the Subject's state.
- Java has a number of implementations of this Pattern.
  - java.util.Observer and java.util.Observable
  - Java.beans.PropertyChangeListener
  - More on this next week.

Page: 217

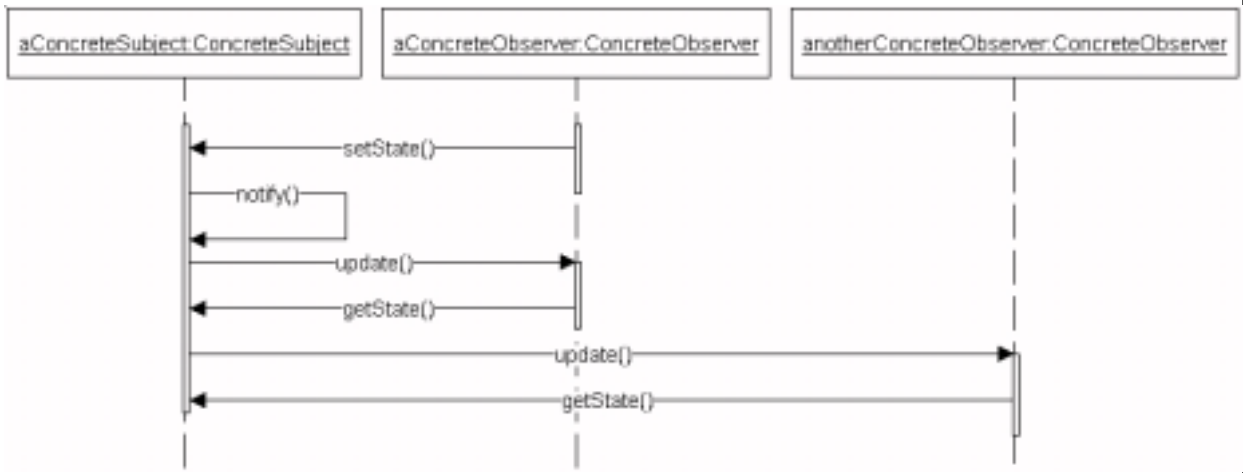
Observer Pattern



Page: 218

Observer Pattern

Observer Pattern Sequence Diagram



Observer Pattern

Code Sample – Subject.java

```
import java.util.*;

public abstract class Subject {
    ArrayList observers = new ArrayList();

    public void attach(Observer observer) {
        observers.add(observer);
    }
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while ( iterator.hasNext() ) {
            Observer observer = (Observer)iterator.next();
            observer.update();
        }
    }
}
```

Page: 220

Code Sample – ConcreteSubject.java

```
public class ConcreteSubject extends Subject {
    String state="";

    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}
```

Code Sample – Observer.java

```
public abstract class Observer {
    public abstract void update();
}
```

Code Sample – ConcreteObserver1.java

```
public class ConcreteObserver1 extends Observer {
    private String observerState = "";
    private ConcreteSubject subject;

    public ConcreteObserver1(ConcreteSubject subject){
        this.subject = subject;
    }
    public void update() {
        observerState = subject.getState();
        System.out.println(observerState);
    }
}
```

Code Sample – ConcreteObserver2.java

```
public class ConcreteObserver2 extends Observer {
    private ConcreteSubject subject;

    public ConcreteObserver2(ConcreteSubject subject){
        this.subject = subject;
    }
    public void update() {
        System.out.println(subject.getState());
    }
}
```

Observer Pattern

Code Sample – ObserverClient.java

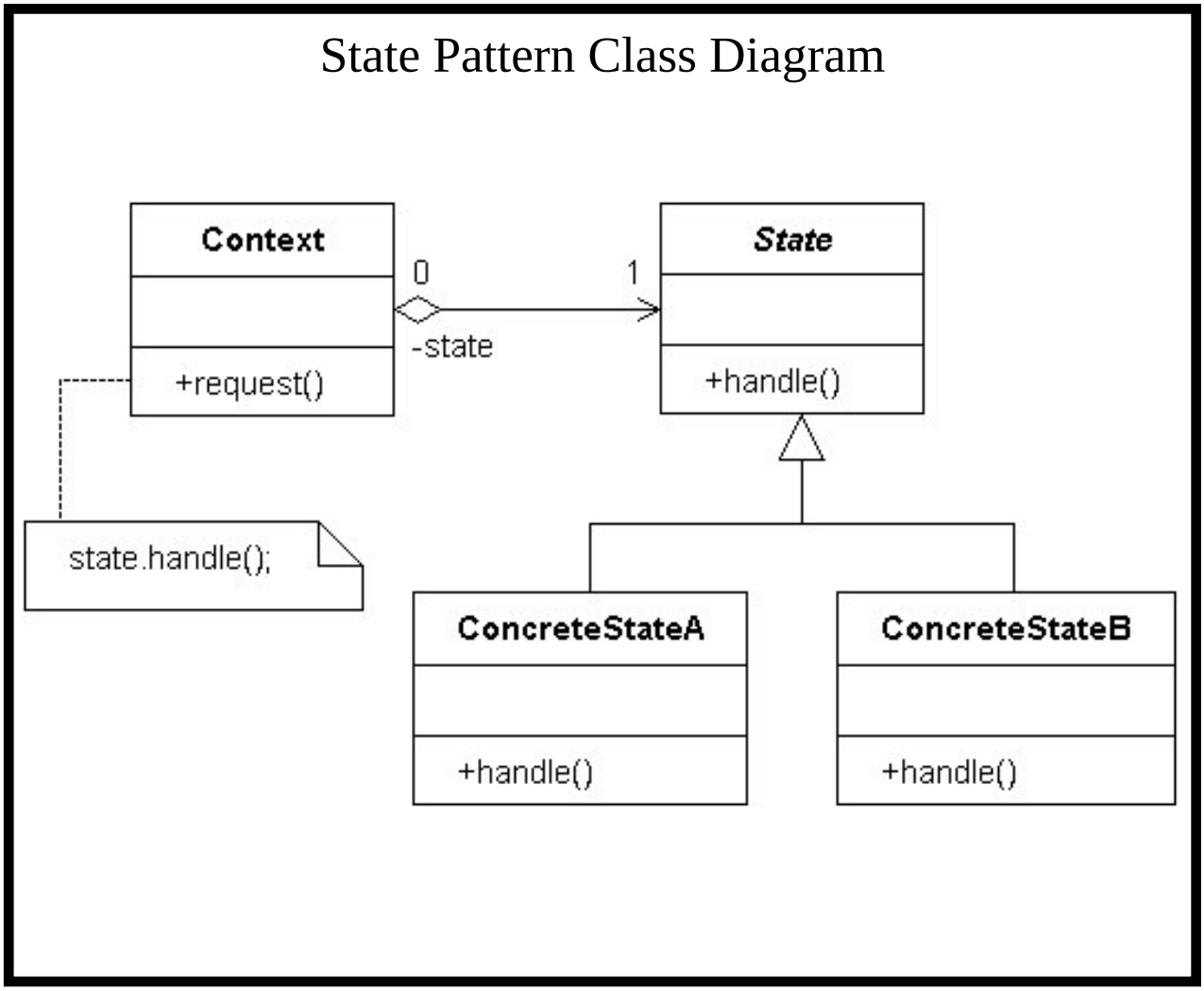
```
public class ObserverClient {

    public ObserverClient() {
        ConcreteSubject cs = new ConcreteSubject();
        ConcreteObserver1 co1 = new ConcreteObserver1(cs);
        ConcreteObserver2 co2 = new ConcreteObserver2(cs);
        cs.attach(co1);
        cs.attach(co2);
        cs.setState("Hello, world");
        cs.detach(co2);
        cs.setState("Goodbye cruel world");
    }

    public static void main(String args[]){
        new ObserverClient();
    }
}
```

Page: 225

State Pattern

- The State Pattern allows an object to change some aspect of its behavior based on changes in its internal state.
- The behavior which changes is isolated called through the State interface.
- Various ConcreteStates contain implementations of the State interface.
- The object whose internal behavior changes (the Context) keeps an instance of its current State and must also have the ability to instantiate alternate states.
- Within the Context class, state relevant calls all use the internal instance of the current State.

State Pattern



Page: 227

State Pattern

Code Sample – State.java

```
public interface State{
    public String handle();
}
```

Code Sample – State1.java

```
public class State1 implements State{
    public String handle(){
        return ("I am in state 1");
    }
}
```

Code Sample – State2.java

```
public class State2 implements State{
    public String handle(){
        return ("I am in state 2");
    }
}
```

Code Sample – ErrorState.java

```
public class ErrorState implements State{
    public String handle(){
        return ("Unknown state");
    }
}
```

Page: 228

Code Sample – Context.java

```
public class Context
{
    private State state1 = new State1();
    private State state2 = new State2();
    private State errorState = new ErrorState();
    private int state;

    public void setState(int state)
    {
        this.state = state;
    }

    public String request()
    {
        String message="";
        switch(state)
        {
            case 1:
                message = state1.handle();
                break;
            case 2:
                message = state2.handle();
                break;
            default:
                message = errorState.handle();
        }
        return message;
    }
}
```

Page: 229

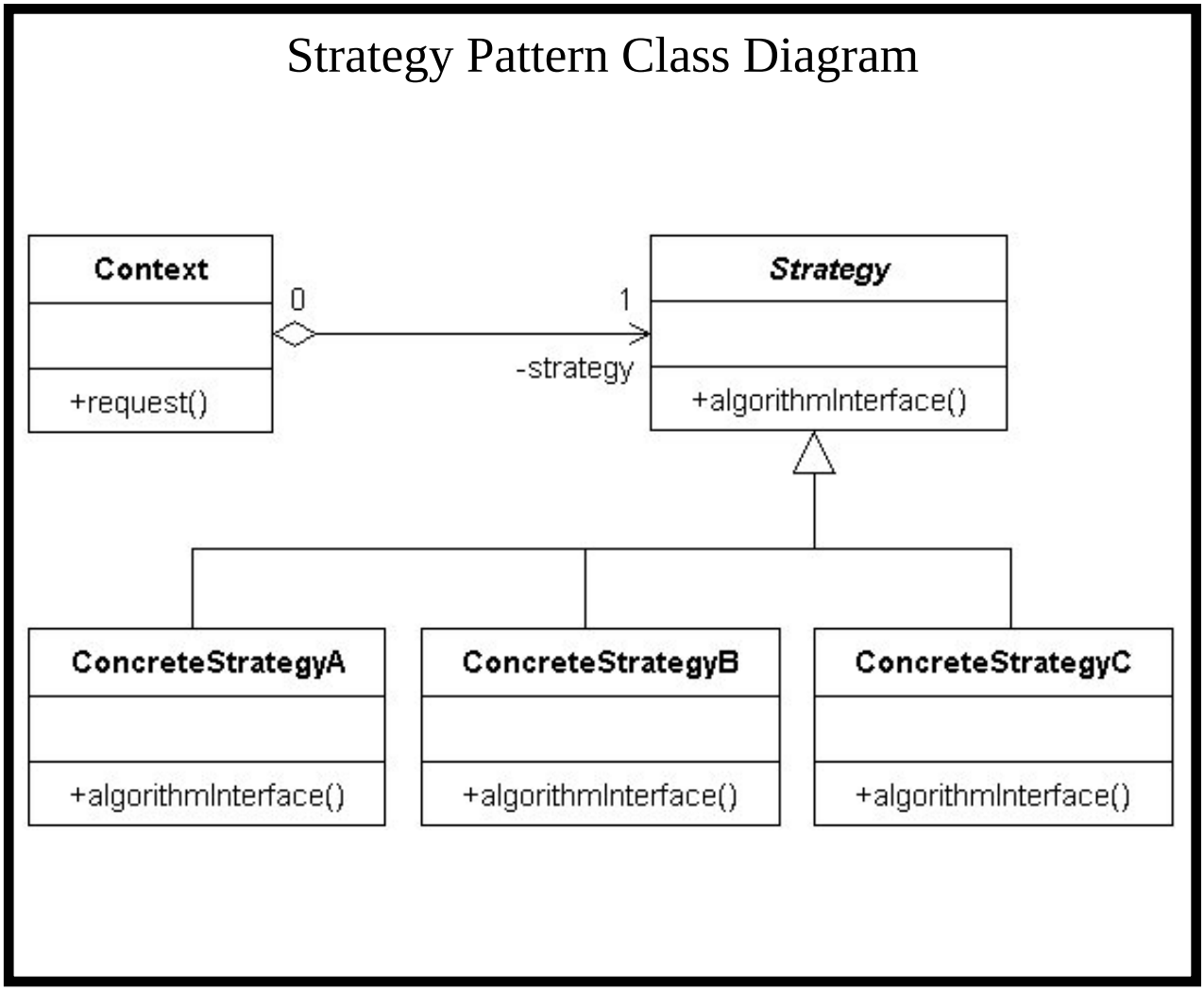
Code Sample – Client.java

```
public class Client
{
    public static void main(String args[])
    {
        Context context = new Context();
        System.out.println(context.request());
        context.setState(1);
        System.out.println(context.request());
        context.setState(2);
        System.out.println(context.request());
    }
}
```

Strategy Pattern

- The Strategy Pattern encapsulates an algorithm in a separate class therefore decoupling the algorithm from its use.
- This pattern is very similar to the State pattern in design, but its intent is quite different.
  - The State pattern is intended to change the way the Context class behaves based on an internal state.
  - The Strategy pattern is intended allow loading of a different algorithm which the normal behavior of the context expects.

Strategy Pattern



Page: 232

Code Sample – Strategy.java

```
public interface Strategy
{
    public String algorithmInterface();
}
```

Code Sample – ConcreteStrategyA.java

```
public class ConcreteStrategyA implements Strategy
{
    public String algorithmInterface()
    {
        return ("I am in Strategy A");
    }
}
```

Code Sample – ConcreteStrategyb.java

```
public class ConcreteStrategyB implements Strategy
{
    public String algorithmInterface()
    {
        return ("I am in Strategy N");
    }
}
```

Page: 233

Code Sample – DefaultStrategy.java

```
public class DefaultStrategy implements Strategy
{
    public String algorithmInterface()
    {
        return ("I am in the Default Strategy");
    }
}
```

Strategy Pattern

Code Sample – Context.java

```
public class Context
{
    private Strategy strategy;
    private Strategy defaultStratagy;
    public Context()
    {
        strategy = new DefaultStrategy();
    }

    public void setStrategy(Strategy strategy)
    {
        this.strategy = strategy;
    }

    public String contextInterface()
    {
        return strategy.algorithmInterface();
    }
}
```

Page: 234

Strategy Pattern

Code Sample – Client.java

```
public class Client
{
  public static void main(String args[])
  {
   Context context = new Context();

   System.out.println(context.contextInterface());
   context.setStrategy(new ConcreteStrategyA());

   System.out.println(context.contextInterface());
   context.setStrategy(new ConcreteStrategyB());

   System.out.println(context.contextInterface());
  }
}
```

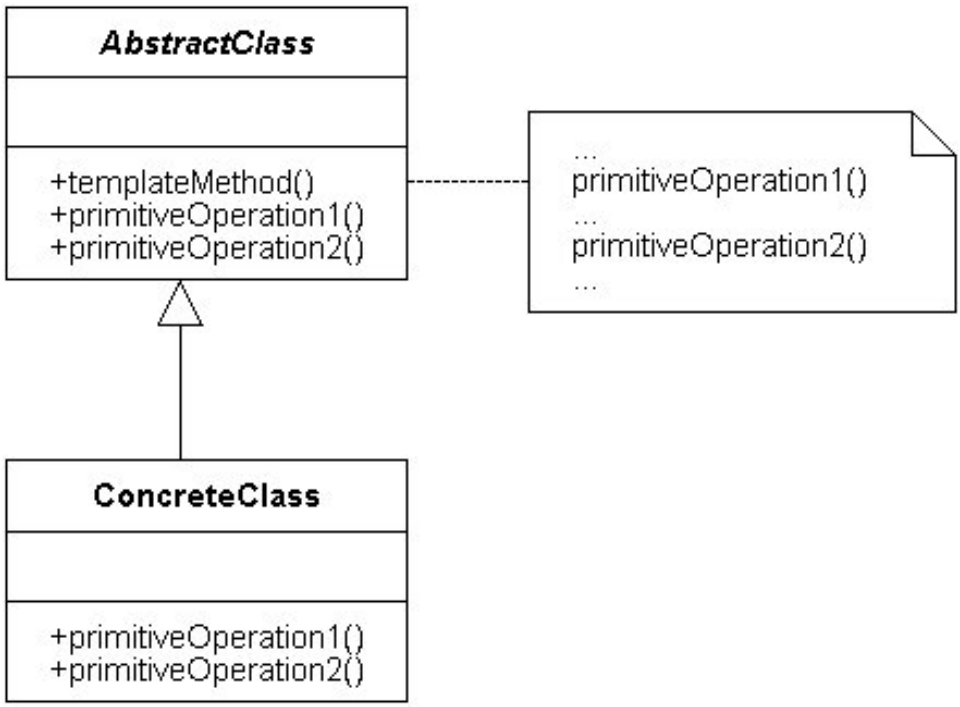
Template Method Pattern

- The Template Method Pattern is used to defer specific pieces of the implementation of a method to a sub class.
- Your base class has a templateMethod which contains a skeleton of implemented code and calls to unimplemented “primitiveOperations”.
- Subclasses will contain the completed primitiveOperations.
- It may be useful to turn this pattern upside down and have the primitiveOperations in the base class and defer the skeleton to the subclass.
- Another variation is combining the Template Method pattern with the Strategy pattern where the strategy contains the primitiveOperations.

Page: 236

Template Method Pattern

Template Method Pattern Class Diagram



Page: 237

Template Method Pattern

Code Sample – FormattedMessage.java

```
public abstract class FormattedMessage
{
    private String message;

    public void setMessage(String message)
    {
        this.message = message;
    }
    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        sb.append(header());
        sb.append(message);
        sb.append(footer());
        return sb.toString();
    }
    protected abstract String header();
    protected abstract String footer();
}
```

Page: 238

Template Method Pattern

Code Sample – QuotedFormattedMessage.java

```
public class QuotedFormattedMessage extends
   FormattedMessage
{
    protected String header()
    {
        return ("\"");
    }

    protected String footer()
    {
        return ("\"");
    }
}
```

Template Method Pattern

Code Sample – HTMLFormattedMessage.java

```
public class HTMLFormattedMessage extends
   FormattedMessage
{
    protected String header()
    {

   return("<HTML>\n<HEAD>\n</HEAD>\n<BODY>\n");
    }

    protected String footer()
    {
        return("\n</BODY>\n</HTML>\n");
    }
}
```

Page: 240

Template Method Pattern

Code Sample – Client.java

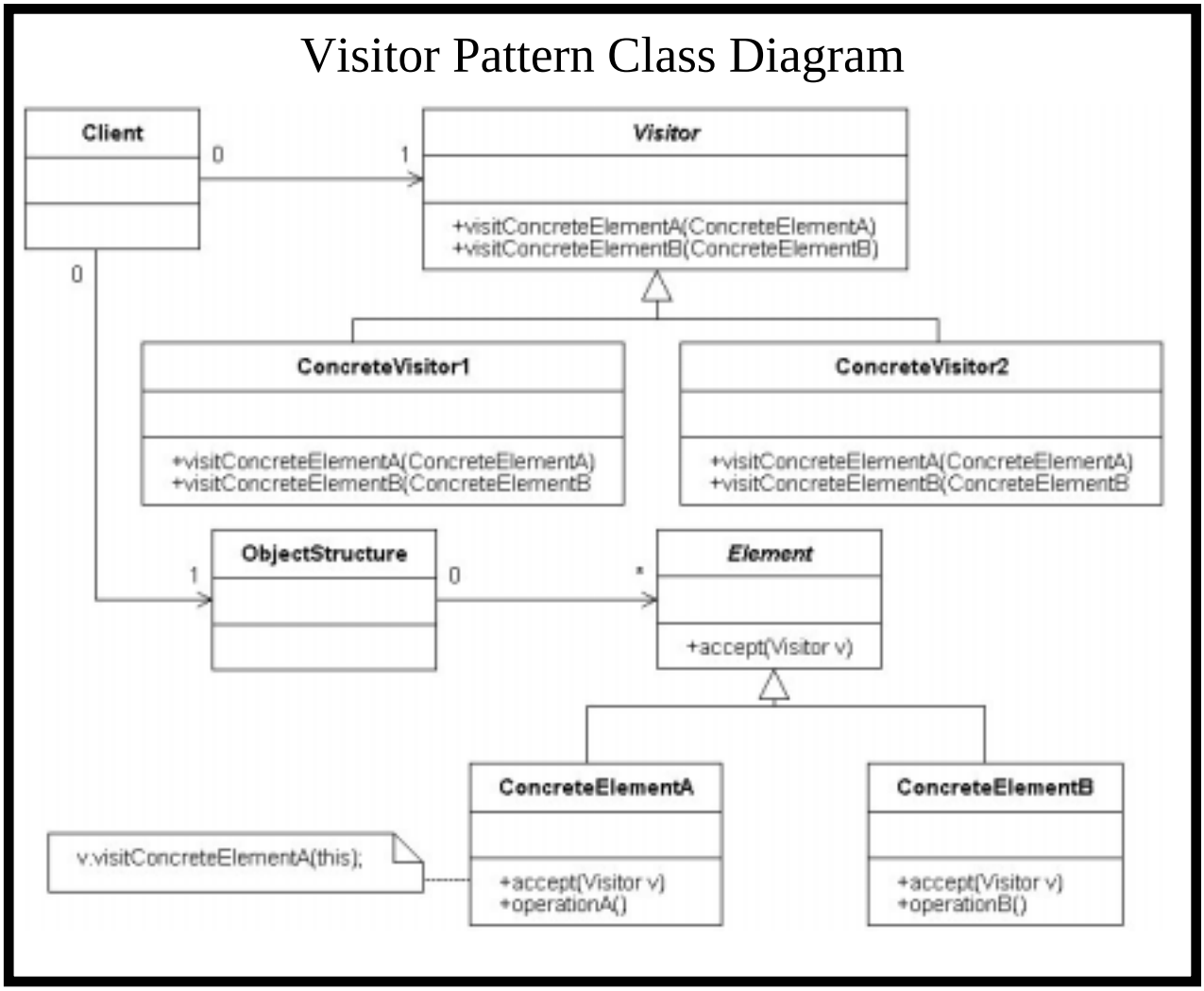
```
public class Client
{
    public Client(FormattedMessage fm)
    {
        System.out.println(fm.toString());
    }

    public static void main(String args[])
    {
        FormattedMessage fm;
        fm = new HTMLFormattedMessage();
        fm.setMessage("Hello, world");
        new Client(fm);
        fm = new QuotedFormattedMessage();
        fm.setMessage("Hello, world");
        new Client(fm);
    }
}
```

Visitor Pattern

- The Visitor Pattern is used to represent an operation to be performed on the elements of an object structure.
- Can be used to provide an extension “hook” to an object hierarchy.
- The Visitor abstract class defines methods which will be called whenever an instance of a Visitor is passed to an element in another hierarchy.
- Element defines the base class for a hierarchy which knows how to deal with Visitors. The accept() method in the element takes an instance of a Visitor and calls its methods.
- Visitors are great ways to extend an existing class hierarchy.

Page: 242

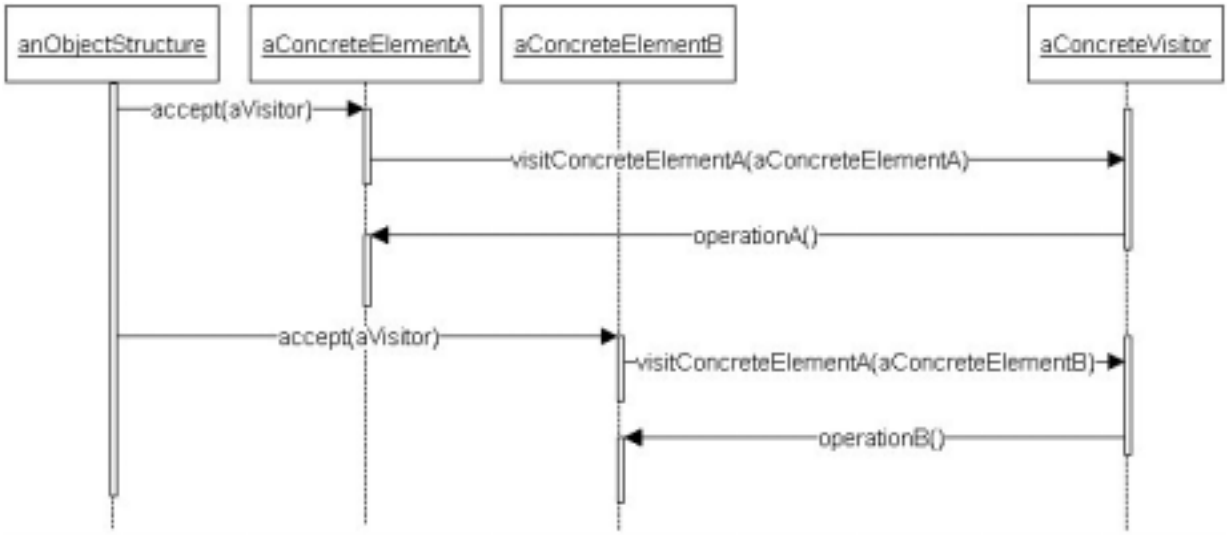
Visitor Pattern



Page: 243

Visitor Pattern

Visitor Pattern Sequence Diagram



Page: 244

Code Sample – Employee.java

```
public abstract class Employee
{
    String name;

    public abstract void accept(Visitor visitor);

    public String getName()
    {
        return name;
    }
}
```

Visitor Pattern

Code Sample – HourlyEmployee.java

```
public class HourlyEmployee extends Employee
{
    private double rate;
    public HourlyEmployee(String name,double rate)
    {
        this.name = name;
        this.rate = rate;
    }

    public double getRate()
    {
        return this.rate;
    }

    public void accept(Visitor v)
    {
        v.visitHourlyEmployee(this);
    }
}
```

Visitor Pattern

Code Sample – SalaryEmployee.java

```
public class SalaryEmployee extends Employee
{
    private double salary;
    public SalaryEmployee(String name,double salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public double getSalary()
    {
        return this.salary;
    }

    public void accept(Visitor v)
    {
        v.visitSalaryEmployee(this);
    }
}
```

Page: 247

Code Sample – Visitor.java

```
public interface Visitor
{
  public void visitHourlyEmployee(HourlyEmployee he);
  public void visitSalaryEmployee(SalaryEmployee se);
}
```

Visitor Pattern

Code Sample – WeeklyEmployeeCost.java

```
public class WeeklyEmployeeCost
    implements Visitor
{
    double weeklyCost=0;
    public void visitHourlyEmployee(HourlyEmployee he)
    {
        weeklyCost += (he.getRate() * 40);
    }

    public void visitSalaryEmployee(SalaryEmployee se)
    {
        weeklyCost += ( se.getSalary() / 52 );
    }

    public double getWeeklyCost()
    {
        return weeklyCost;
    }
}
```

Page: 249

Visitor Pattern

Code Sample – YearlyEmployeeCost.java

```
public class YearlyEmployeeCost
    implements Visitor
{
    double yearlyCost=0;
    public void visitHourlyEmployee(HourlyEmployee he)
    {
        yearlyCost += ((he.getRate() * 40)*52);
    }

    public void visitSalaryEmployee(SalaryEmployee se)
    {
        yearlyCost += se.getSalary();
    }

    public double getYearlyCost()
    {
        return yearlyCost;
    }
}
```

Page: 250

Visitor Pattern

Code Sample – Client.java

```
import java.util.*;
public class Client
{
  public static void runReport(List l, Visitor v){
      Iterator iterator = l.iterator();
      while (iterator.hasNext()){
            Employee e = (Employee)iterator.next();
            e.accept(v);
      }
  }

  public static void main(String args[]){
    ArrayList employees = new ArrayList();
    employees.add(new SalaryEmployee("Tom",65000.00));
    employees.add(new SalaryEmployee(“Sam",45000.00));
    employees.add(new HourlyEmployee("Barry",5.75));

    WeeklyEmployeeCost wec = new WeeklyEmployeeCost();
    runReport(employees,wec);
    System.out.println("Total weekly cost: "+
                       wec.getWeeklyCost());

    YearlyEmployeeCost yec = new YearlyEmployeeCost();
    runReport(employees,yec);
    System.out.println("Total yearly cost: "+
                       yec.getYearlyCost());
  }
}
```

Page: 251

Uses in java

- Now we will look at a few real examples from the Java class hierarchy.
- Throughout the design of the java class hierarchy, you can find good examples of Design Patterns in action.
- Studying the Java class hierarchy is a great way to increase your understanding of the Java language, and to help your Object oriented Design skills.
- All of the source comes with the class JDK.
- Set aside some time each week to analyze a piece of the hierarchy. Document what you learned in UML and generate some sample code.
- Look for patterns!

Observer - Observable

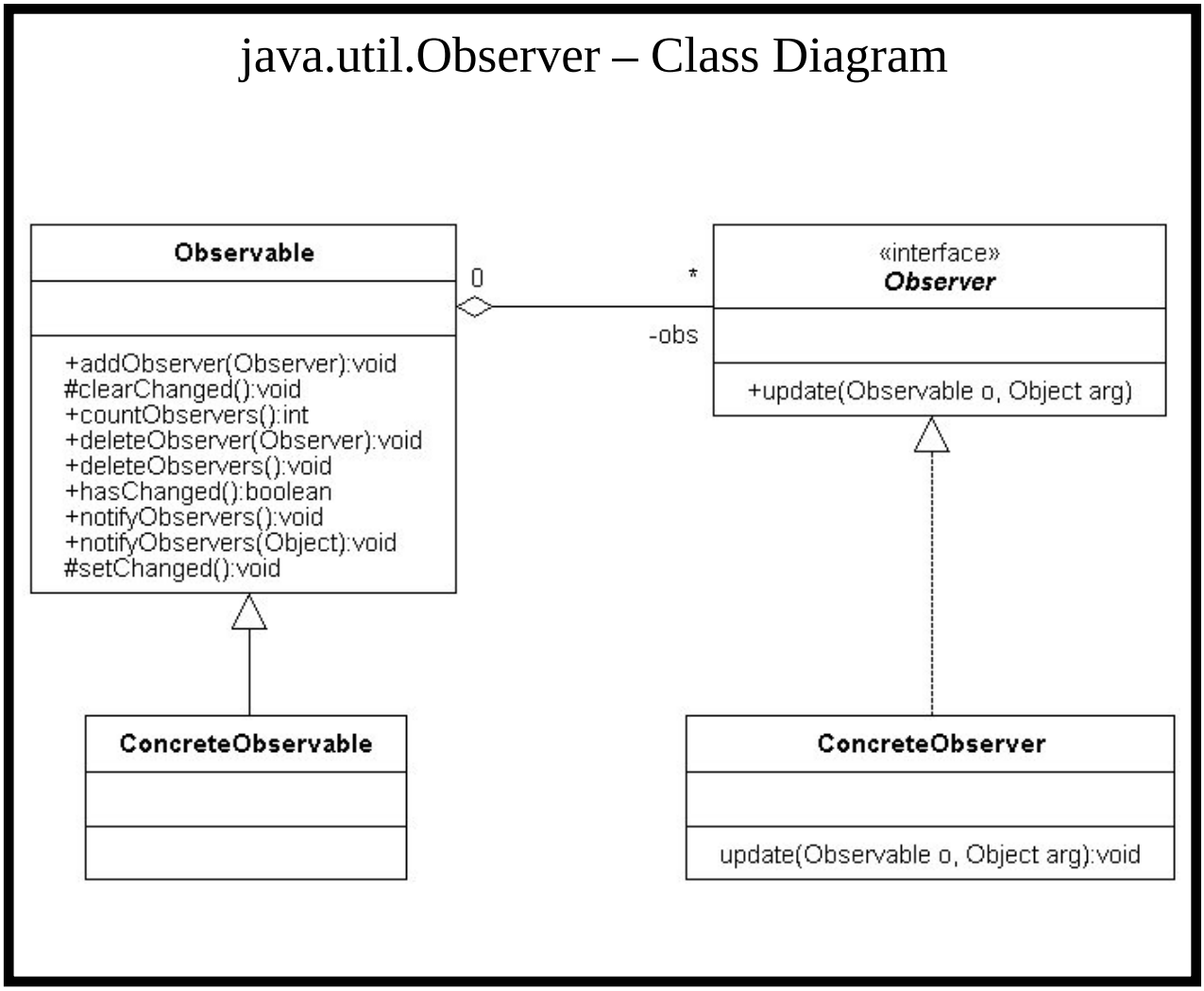
- The JDK comes with an implementation of the Observer pattern in java.lang.util.
- The Observer interface defines a single method called update().
- Classes wishing to observer another object will need to implement the Observer interface by providing an update() method.
  - Update takes two arguments.
    - Observable – the class being observed
    - Object – something that the Observable passes to all observers.

Observer - Observable

- The java.util.Observable class is a concrete class that contains the all the code needed to maintain a list of Observers, and to call their update() methods.
- A class wishing to be observed simply extends the Observable class and calls the setChanged() method followed by the notifyObservers() method.
- A class wishing to be an observer will call the Observable objects addObserver() method.
- An observer will be passed an instance of the Observable object as well as an Object containing some kind of additional data – most likely the thing that actually changed (its state).
- To make good use of the state, there is usually pretty tight coupling between the ConcreteObserver and ConcreteObservable.

Page: 254

Observer - Observable



Page: 255

Observer - Observable

- There is a pretty major usability flaw in this arrangement however.
- Observable is itself a concrete class. The only way you can make a class an Observable is to subclass it.
- If your class is already a sub class of another class, you are out of luck!
- Well – almost.
- Using composition and some of our other Design Patterns, we can still make some use out off all of this.

Page: 256

Observer - Observable

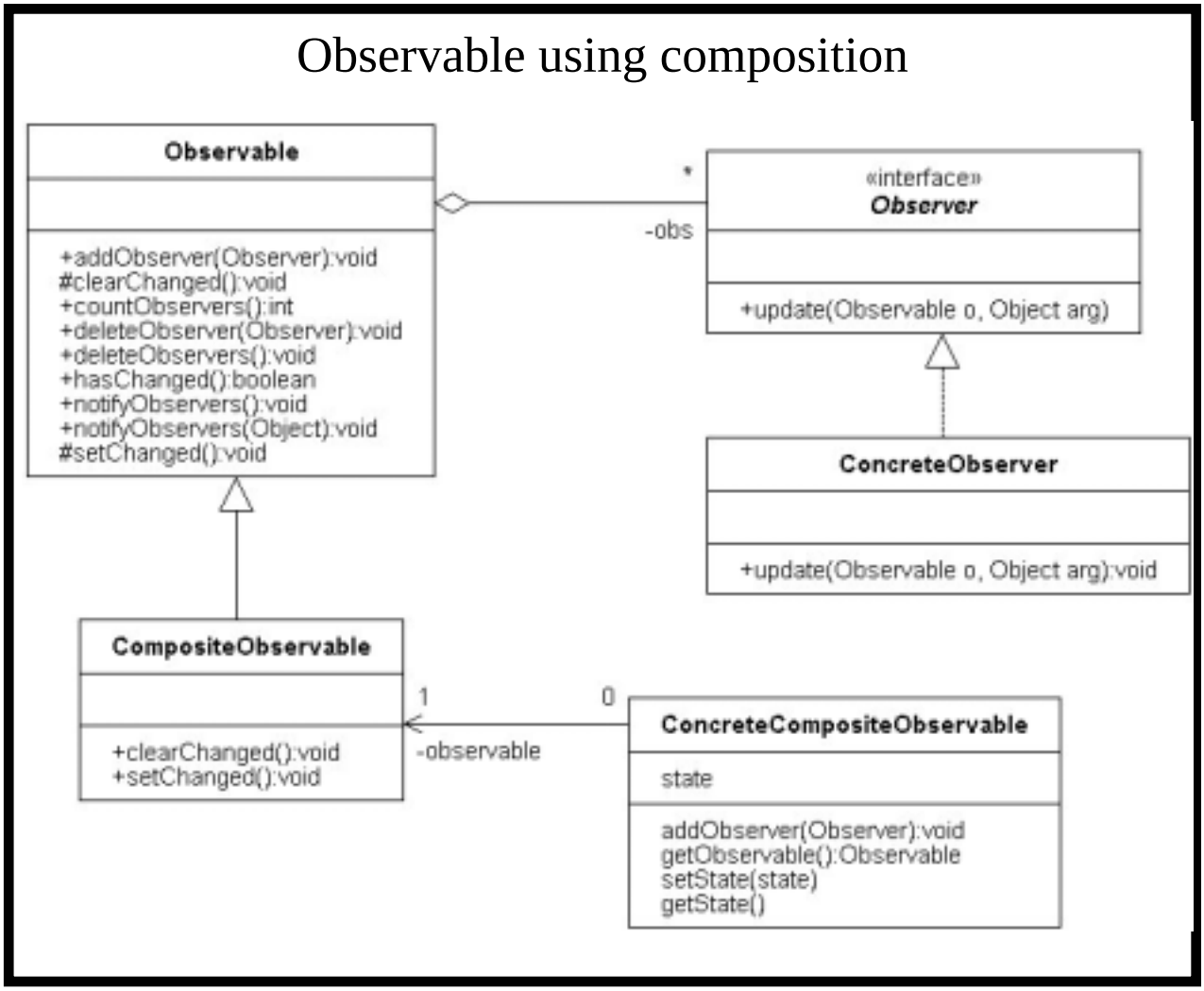
- You can take any class, create an instance of an Observable and contain it within. This is often called a backing object.
- Your class will then become a facade to the backing Observable.
- You will need to implement each of the Observable's methods by calling the equivalent method on your backing Observable object.

Page: 257

Observer – Observable

- But, Observable has two important methods in Observable that are declared protected.
  - setChanged()
  - clearChanged()
- Only sub classes can call protected methods.
- So, we can create a sub class of Observable which simply redefines those methods as public.
  - In my example, I call this class CompositeObservable.
- Now, our class should deal with CompositeObservable instead of Observables.

java.util.observer



Page: 259

java.util.observer

Code Sample – ConcreteObserver.java

```
import java.util.*;

public class ConcreteObservable extends Observable
{
    private String state;

    public void setState(String state)
    {
        this.state = state;
        setChanged();
        notifyObservers(state);
    }
}
```

Page: 260

java.util.observer

Code Sample – CompositeObservable.java

```
import java.util.Observer;
import java.util.Observable;

/* This class extends the Observable class by making
 * two methods public.
 */
class CompositeObservable extends Observable {

  public void clearChanged() {
    super.clearChanged();
  }

  public void setChanged() {
    super.setChanged();
  }

}
```

java.util.observer

Code Sample – CompositeConcreteObservable.java

```
import java.util.*;
public class ConcreteCompositeObservable {
    private CompositeObservable observable =
                                new CompositeObservable();
    private String state;

    public Observable getObservable() {
        return this.observable;
    }

    public void addObserver(Observer observer){
        observable.addObserver(observer);
    }

    public final void setState(String state) {
        this.state = state;
        observable.setChanged();
        observable.notifyObservers(state);
    }
}
```

Page: 262

java.util.observer

Code Sample –ConcreteObserver.java

```
import java.util.*;

public class ConcreteObserver implements Observer
{

   public void update(Observable o,Object arg)
   {
        System.out.println("Observer recived update");
        System.out.println("ConcreteObserver:“
                           +(String) arg);
   }

}
```

Page: 263

java.util.observer

Code Sample – ObserverClient.java

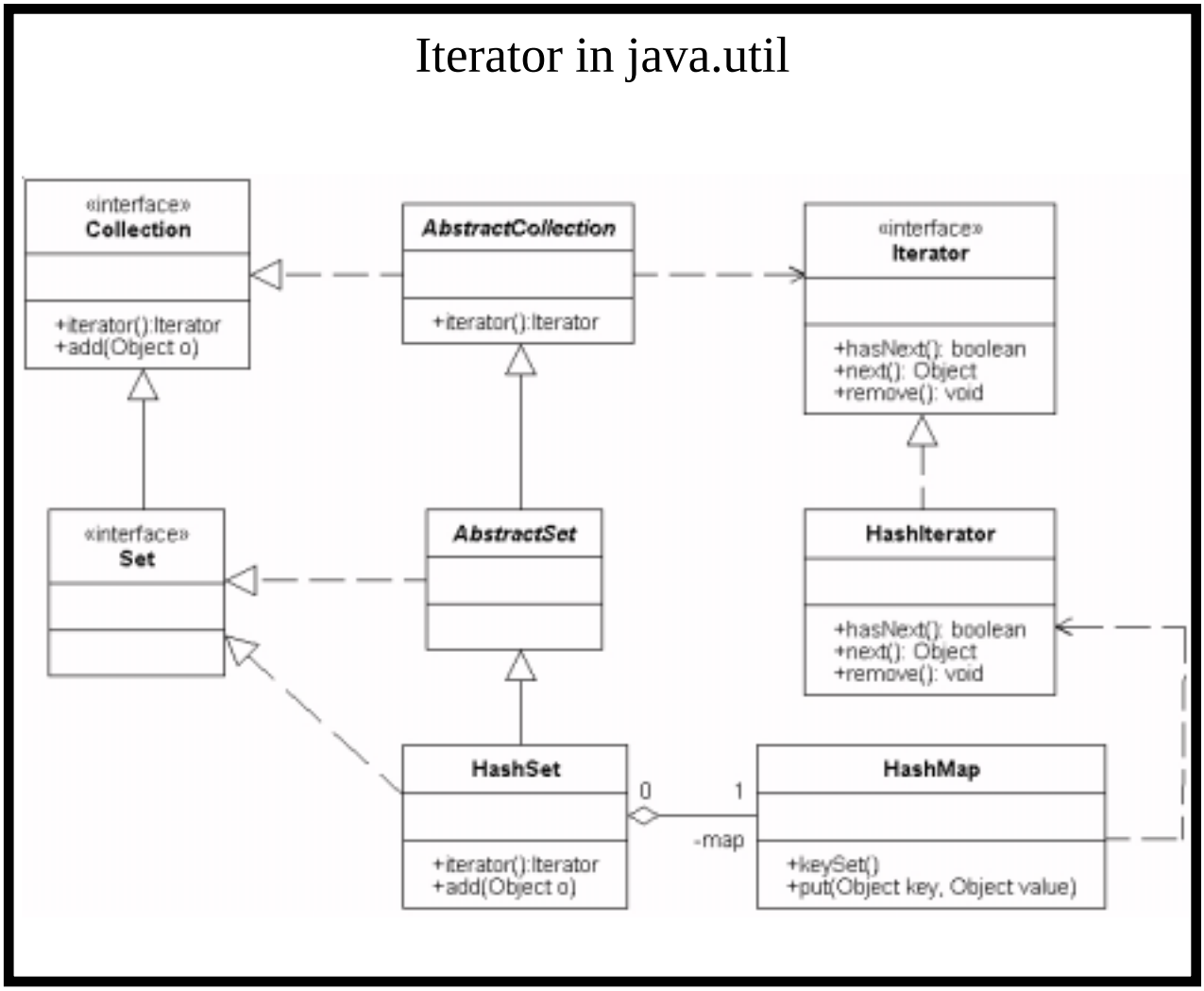
```
public class ObserverClient
{
    public static void main(String args[])
    {
        ConcreteCompositeObservable cco =
                    new ConcreteCompositeObservable();
        ConcreteObserver co = new ConcreteObserver();

        cco.addObserver(co);
        cco.setState("Hello, world");

        ConcreteObservable concreteObservable =
                    new ConcreteObservable();
        concreteObservable.addObserver(co);
        concreteObservable.setState("Goodbye cruel world");

    }
}
```

Page: 264

Iterator - Collection

- The java.util.Collection interface provides the basis for some very useful aggregation classes.
- They are very interesting to study to see many Design Patterns in action.
- Collections uses the Iterator Pattern to iterate through the collection.
- Various implementations of Collections make use of simpler backing storage classes, but they each implement some detailed behavior (such as sort, set, lists etc).
- The backing storage class provides the actual Iterator.

Iterator - Collection



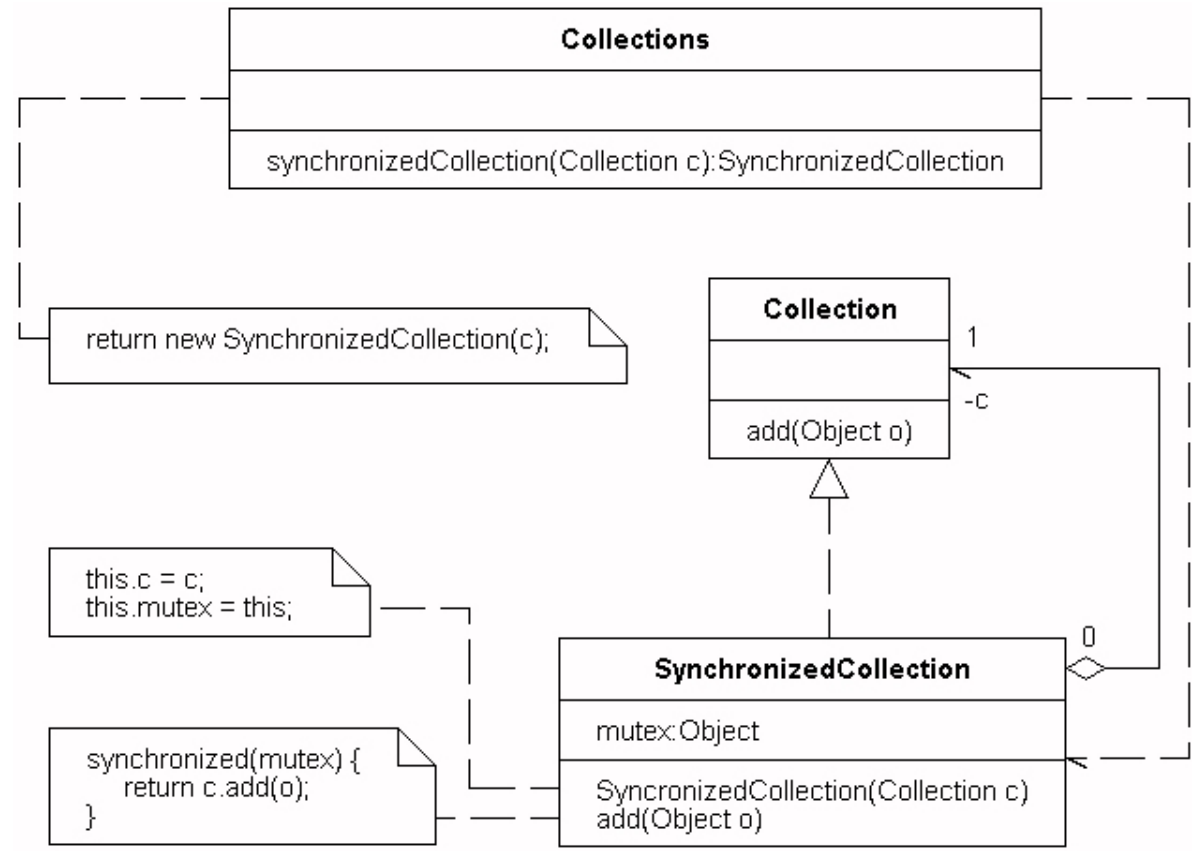
Page: 266

Decorator - Collections

- The java.util.Collections class provides a number of “interface wide” methods for extending the power of any given Collection.
- For efficiency sake, most of the Collections are not synchronzed.
- Collections.synchronizedCollection() will take any Collection and return a new SynchronzedCollection.
- A SyncronizedCollection is simply a decorator for a Collection. Each method in the Collection interface is implemented by wrapping backing Collection within a block synchronized on a local mutex.

Decorator - Collections

java.util.Collections used to create a synchronized java.util.Collection



Page: 268

Conclusions

- Design Patterns show us how to use object oriented programming techniques to their fullest.
- The patterns work well in Java if you make greater use of interfaces and composition.
- To maximize reuse, remember to strive to decouple classes from each other.
- Expose details about your classes only through interfaces.
  - Use Creational Patterns in instantiate classes
  - Use Structural Patterns to establish relationships
  - Use Behavioral Patterns provide extension points for implementations.
- Go out and try it!