Java Design Patterns

Kafi Rahman@ CS
Truman State University

Outline

Design Patterns

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

- Design patterns capture the best practices of experienced object-oriented software developers.
- Design patterns are solutions to general software development problems.

Design Patterns :: Motivation

- Suppose we have the following class
  - can we extend the Student class and have an address instance variable as well?

```
final class Student{
        int studentID;
        String name;
        Student(int id, String name){
                this.studentID=id;
                this.name = name;
        }

        String getName() { return this.name;}
        int getID() { return this.studentID;}
        void setName(String n) { this.name = n;}
        void setID(int id) { this.studentID = id;}
}
```

Design Patterns :: Motivation

- We can, by following decorate design patterns, and create a wrapper version of the Student class

```
class EnStudent{
        Student myStudent;
        String address;
        EnStudent(int id, String name, String address){
                myStudent = new Student(id, name);
                this.address = address;
        }

        String getName() { return myStudent.getName();}
        int getID() { return myStudent.getID();}
        void setName(String n) { myStudent.setName(n);}
        void setID(int id) { myStudent.setID(id);}

        String getAddress() { return this.address;}
        void setAddress(String a) { this.address= a;}
}
```

Design Patterns

- In general, a pattern has four essential elements.
  - The pattern name
  - The problem
  - The solution
  - The consequences

Design Patterns :: Name

- The pattern name is a handle we can use to describe a design problem, it's solutions, and consequences in a word or two.

- Naming a pattern immediately increases the design vocabulary.
- It makes it easier to think about designs and to communicate them and their trade-offs to others.

Design Patterns :: Problem

- The problem describes when to apply the pattern.
- It explains the problem and its context.
- It might describe class or object structures that are symptomatic of an inflexible design.
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Design Patterns :: Solution

- The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

Design Patterns :: Consequences

- The consequences are the results and trade-offs of applying the pattern.

- The consequences for software often concern space and time trade-offs.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.
- Listing these consequences explicitly helps you understand and evaluate them

Design Patterns :: Why Use Patterns with Java?

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later   (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, "We could use Strategy here."
- Provide a higher level prospective
  - Frees us from dealing with the details too early

Design Patterns:: Other Advantages

- Most design patterns make software more modifiable
  - we are using time tested solutions
- Using design patterns makes software systems easier to change
  - more maintainable
- Helps increase the understanding of basic object-oriented design principles
  - encapsulation, inheritance, interfaces, polymorphism

Design Patterns:: Types

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their Design Patterns book define 23 design patterns divided into three types:
  - Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
  - Structural patterns help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
  - Behavioral patterns help you define the communication between objects in your system and how the flow is controlled in a complex program.

Design Patterns:: Creational Patterns and Java I

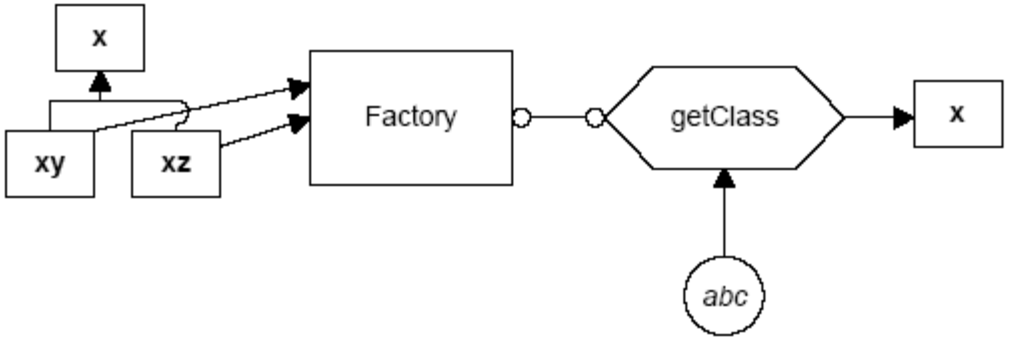
- The creational patterns deal with the best way to create instances of objects.
- In Java, the simplest way to create an instance of an object is by using the new operator.
  - Fred = new Fred(); //instance of Fred class
- In many cases, the exact nature of the object that is created could vary with the needs of the program and abstracting the creation process into a special "creator" class can make your program more flexible and general.

Design Patterns:: Creational Patterns and Java II

- The Factory Pattern provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- The Abstract Factory Pattern provides an interface to create and return one of several families of related objects.
- The Builder Pattern separates the construction of a complex object from its representation.
- The Singleton Pattern is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

The Factory Pattern :: How does it Work?

- The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



- Here, x is a base class and classes xy and xz are derived from it.
- The Factory is a class that decides which of these subclasses to return depending on the arguments you give it.
- The getClass() method passes in some value abc, and returns some instance of the class x. Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations.

The Factory Pattern:: The Base Class

- Let's consider a simple case where we could use a Factory class. Suppose we have an entry form and we want to allow the user to enter his name either as "firstname lastname" or as "lastname, firstname".
- Let's make the assumption that we will always be able to decide the name order by whether there is a comma between the last and first name.

```
class Namer { // a class to take a string apart into two names
        protected String last; // store last name here
        protected String first; // store first name here

        public String getFirst() {
                return first; // return first name
        }

        public String getLast() {
                return last; // return last name
        }
}
```

The Factory Pattern :: The First Derived Class

- In the FirstFirst class, we assume that everything before the last space is part of the first name.

```
class FirstFirst extends Namer {
        public FirstFirst(String s) {
                int i = s.lastIndexOf(" ");  //find separating space
                if (i > 0) {
                        first = s.substring(0, i).trim(); //left = first name
                        last =s.substring(i+1).trim(); //right = last name
                } else {
                        first = "" // put all in last name
                        last = s; // if no space
                }
        }
}
```

The Factory Pattern:: The Second Derived Class

- In the LastFirst class, we assume that a comma delimits the last name.

```
class LastFirst extends Namer { // split last, first
        public LastFirst(String s) {
                int i = s.indexOf(","); // find comma
                if (i > 0) {
                        last = s.substring(0, i).trim(); // left= last name
                        first = s.substring(i + 1).trim(); // right= first name
                } else {
                        last = s; // put all in last name
                        first = ""; // if no comma
                }
        }
}
```

The Factory Pattern:: Building the Factory

- The Factory class is relatively simple. We just test for the existence of a comma and then return an instance of one class or the other.

```
class NameFactory {
        // returns an instance of LastFirst or FirstFirst
        // depending on whether a comma is found
        public Namer getNamer(String entry) {
                int i = entry.indexOf(","); // comma determines name order
                if (i > 0)
                        return new LastFirst(entry); // return one class
                else
                        return new FirstFirst(entry); // or the other
        }
}
```

The Factory Pattern:: Using the Factory

```
NameFactory nfactory = new NameFactory();
String sFirstName, sLastName;….

private void computeName() {
        //send the text to the factory and get a class back
        namer = nfactory.getNamer(entryField.getText());
        //compute the first and last names using the returned class
        sFirstName = namer.getFirst();
        sLastName = namer.getLast();
}
```

The Factory Pattern:: When to Use a Factory Pattern

- You should consider using a Factory pattern when:
  - A class can't anticipate which kind of class of objects it must create.
  - A class uses its subclasses to specify which objects it creates.

- There are several similar variations on the factory pattern to recognize:
  - The base class is abstract and the pattern must return a complete working class.
  - The base class contains default methods.
- Parameters are passed to the factory telling it which of several class types to return.
  - In this case the classes may share the same method names but may do something quite different.

The Abstract Factory Pattern:: How does it Work?

- The Abstract Factory pattern is one level of abstraction higher than the factory pattern.
  - This pattern returns one of several related classes, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several factories.

- One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows, Motif or Macintosh:
- You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.

The Abstract Factory Pattern :: A Garden Maker Factory?

- Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:
    - What are good border plants?
    - What are good center plants?
    - What plants do well in partial shade?

- We want a base Garden class that can answer these questions

```
public abstract class Garden {
        public abstract Plant getCenter();
        public abstract Plant getBorder();
        public abstract Plant getShade();
}
```

The Abstract Factory Pattern:: The Plant Class

- The Plant class simply contains and returns the plant name:

```
public class Plant {
        String name;

        public Plant(String pname) {
                name = pname; // save name
        }

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

The Abstract Factory Pattern:: A Garden Class

- A Garden class simply returns one kind of each plant. So, for example, for the vegetable garden we simply write:

```
public class VegieGarden extends Garden {
        public Plant getShade() {
                return new Plant("Broccoli");
        }

        public Plant getCenter() {
                return new Plant("Corn");
        }

        public Plant getBorder() {
                return new Plant("Peas");
        }
}
```

The Abstract Factory Pattern:: A Garden Maker Class – The Abstract Factory

- We create a series of Garden classes - VegieGarden, PerennialGarden, and AnnualGarden, each of which returns one of several Plant objects. Next, we construct our abstract factory to return an object instantiated from one of these Garden classes and based on the string it is given as an argument:

```
class GardenMaker {
        // Abstract Factory which returns one of three gardens
        private Garden gd;

        public Garden getGarden(String gtype) {
                gd = new  VegieGarden(); // default
                if (gtype.equals("Perennial"))
                        gd = new PerennialGarden();
                if (gtype.equals("Annual"))
                        gd = new AnnualGarden();
                return gd;
        }
}
```

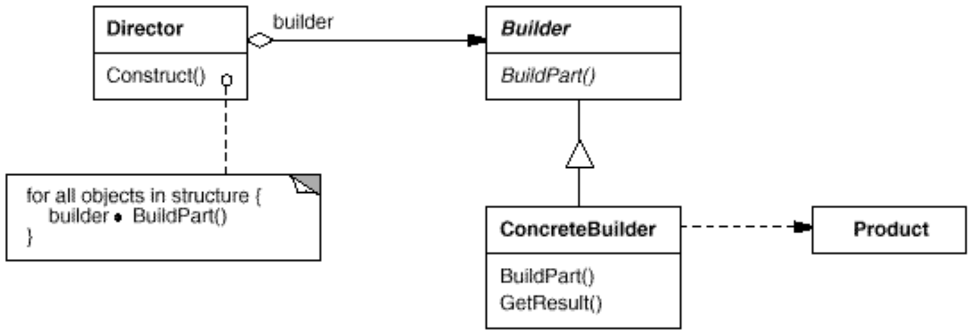
The Abstract Factory Pattern:: Consequences of Abstract Factory

- The actual class names of these classes are hidden in the factory and need not be known at the client level at all.

- Because of the isolation of classes, you can change or interchange these product class families freely.

The Builder Pattern :: How does it Work? - I

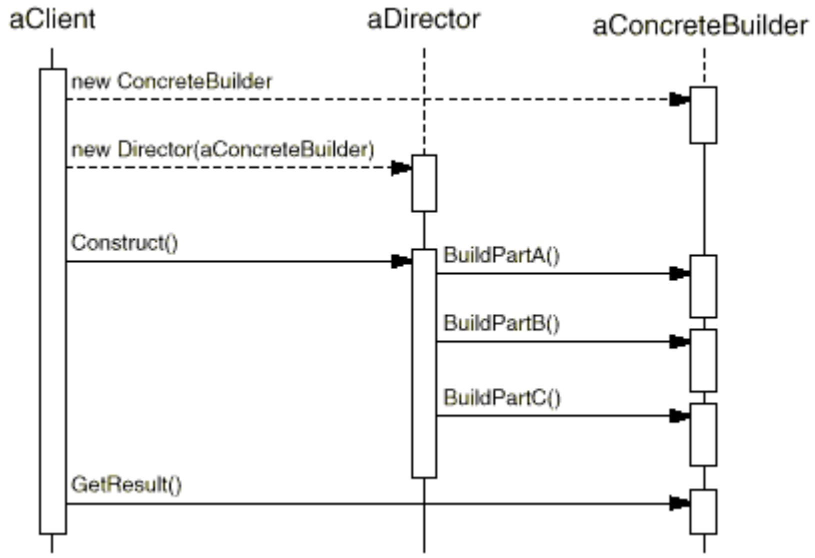
- The Builder Pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.



- Builder – specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder – constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product .
- Director – constructs an object using the Builder interface.
- Product – represents the complex object under construction.

The Builder Pattern:: How does it Work? - II

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.
- The following interaction diagram illustrates how Builder and Director cooperate with a client.



The Builder Pattern :: Applicability of Builder Pattern

- Use the Builder pattern when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that is constructed.

The Builder Pattern :: Consequences of Builder Pattern

- A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
- Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.

- Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.

The Builder Pattern :: Pizza Builder

```
/** "Product" */
class Pizza {
	private String dough = "";
	private String sauce = "";
	private String topping = "";

	public void setDough(String dough) {
		this.dough = dough;
	}

	public void setSauce(String sauce) {
		this.sauce = sauce;
	}

	public void setTopping(String topping) {
		this.topping = topping;
	}
}
```

The Builder Pattern :: Pizza Builder

```
/** "Abstract Builder" */
abstract class PizzaBuilder {
        protected Pizza pizza;

        public Pizza getPizza() {
                return pizza;
        }

        public void createNewPizzaProduct() {
                pizza = new Pizza();
        }

        public abstract void buildDough();

        public abstract void buildSauce();

        public abstract void buildTopping();
}
```

The Builder Pattern :: Pizza Builder

```java
/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
	public void buildDough() {
		pizza.setDough("cross");
	}

	public void buildSauce() {
		pizza.setSauce("mild");
	}

	public void buildTopping() {
		pizza.setTopping("ham+pineapple");
	}
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
	public void buildDough() {
		pizza.setDough("pan baked");
	}

	public void buildSauce() {
		pizza.setSauce("hot");
	}

	public void buildTopping() {
		pizza.setTopping("pepperoni+salami");
	}
}
```

The Builder Pattern :: Pizza Builder

```
/** "Director" */
class Waiter {
	private PizzaBuilder pizzaBuilder;

	public void setPizzaBuilder(PizzaBuilder pb) {
		pizzaBuilder = pb;
	}

	public Pizza getPizza() {
		return pizzaBuilder.getPizza();
	}

	public void constructPizza() {
		pizzaBuilder.createNewPizzaProduct();
		pizzaBuilder.buildDough();
		pizzaBuilder.buildSauce();
		pizzaBuilder.buildTopping();
	}
}
```

The Builder Pattern :: Pizza Builder

```
/** A customer ordering a pizza. */
class BuilderExample {
	public static void main(String[] args) {
		Waiter waiter = new Waiter();
		PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
		PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

		waiter.setPizzaBuilder(hawaiianPizzaBuilder);
		waiter.constructPizza();

		Pizza pizza = waiter.getPizza();
	}
}
```

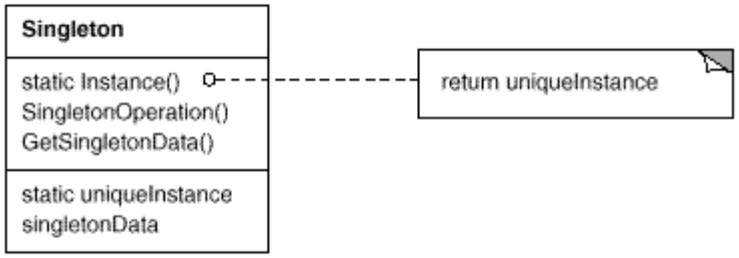
The Singleton Pattern:: Definition & Applicability - I

- Sometimes it is appropriate to have exactly one instance of a class:
  - window managers,
  - print spoolers,
  - filesystems.
- Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.

- The Singleton pattern addresses all the concerns above.
- With the Singleton design pattern you can:
  - Ensure that only one instance of a class is created.
  - Provide a global point of access to the object.
  - Allow multiple instances in the future without affecting a singleton class' clients.

The Singleton Pattern:: Definition & Applicability - II

- The class itself is responsible for keeping track of its sole instance.
- The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.
- Singletons maintain a static reference to the sole singleton instance and return a reference to that instance from a static getInstance() method.



The Singleton Pattern:: The Classic Singleton - I

- The ClassicSingleton class maintains a static reference to the lone singleton instance.

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

    protected ClassicSingleton() {
        }

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

The Singleton Pattern:: The Classic Singleton - II

- The ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time.
- The ClassicSingleton class implements a protected constructor so clients cannot instantiate ClassicSingleton instances; however, the following code is perfectly legal:

```
public class SingletonInstantiator {
  public SingletonInstantiator() {
        ClassicSingleton instance = ClassicSingleton.getInstance();
        ClassicSingleton anotherInstance = new ClassicSingleton();
       ...
  }
}
```

The Singleton Pattern:: Problems - I

- How can a class that does not extend ClassicSingleton create a ClassicSingleton instance if the ClassicSingleton constructor is protected?
  - Protected constructors can be called by subclasses and by other classes in the same package.
  - ClassicSingleton and SingletonInstantiator are in the same package (the default package), SingletonInstantiator() methods can create ClassicSingleton instances.

- Solutions:
  - We can make the ClassicSingleton constructor private so that only ClassicSingleton's methods can call it.
  - We can put your singleton class in an explicit package, so classes in other packages (including the default package) cannot instantiate singleton instances.

Java Design Patterns :: Structural Patterns and Java - I

- Structural patterns describe how classes and objects can be combined to form larger structures.
  - Class patterns describe how inheritance can be used to provide more useful program interfaces.
  - Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition
- The Structural patterns are:
  - Adapter
  - Composite
  - Proxy
  - Flyweight
  - Façade
  - Bridge
  - Decorator

Java Design Patterns:: Structural Patterns and Java - II

- The Adapter pattern can be used to make one class interface match another to make programming easier.
- The Composite pattern is a composition of objects, each of which may be either simple or itself a composite object.
- The Proxy pattern is frequently a simple object that takes the place of a more complex object that may be invoked later, for example when the program runs in a network environment.

Java Design Patterns :: Structural Patterns and Java - II

- The Façade pattern is used to make a single class represent an entire subsystem.
- The Bridge pattern separates an object's interface from its implementation, so you can vary them separately.
- The Decorator pattern, which can be used to add responsibilities to objects.

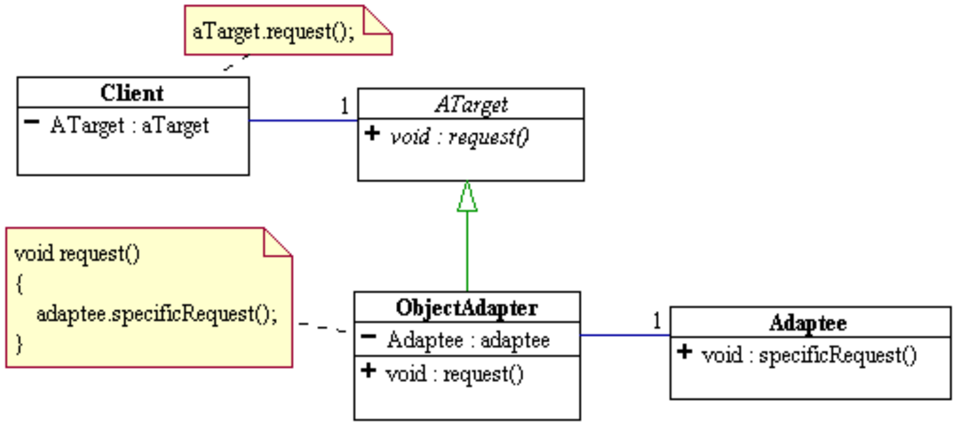
The Adapter Pattern :: Definition & Applicability

- Adapters are used to enable objects with different interfaces to communicate with each other.
- The Adapter pattern is used to convert the programming interface of one class into that of another.
- We use adapters whenever we want unrelated classes to work together in a single program.

- The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.
- Adapters in Java can be implemented in two ways: by inheritance, and by object composition.

The Adapter Pattern :: Object Adapters

- Object adapters use a compositional technique to adapt one interface to another.
- The adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee.
- When the client calls the request() method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee.
- Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.



The Adapter Pattern:: Example - I

- If a client only understands the SquarePeg interface for inserting pegs using the insert() method, how can it insert round pegs, which are pegs, but that are inserted differently, using the insertIntoHole() method?

```java
/**
 * The SquarePeg class. This is the Target class.
 */
public class SquarePeg {
        public void insert(String str) {
                System.out.println("SquarePeg insert(): " + str);
        }
}

/**
 * The RoundPeg class. This is the Adaptee class.
 */
public class RoundPeg {
        public void insertIntoHole(String msg) {
                System.out.println("RoundPeg insertIntoHole(): " + msg);
        }
}
```

The Adapter Pattern:: Example - II

- Solution:
  - Design a RoundToSquarePeg adapter that enables to insertIntoHole() a RoundPeg object connected to the adapter to be inserted as a SquarePeg, using insert().

```
/**
* The RoundToSquarePegAdapter class.
* This is the Adapter class.
* It adapts a RoundPeg to a SquarePeg.
* Its interface is that of a SquarePeg.
*/
public class RoundToSquarePegAdapter extends SquarePeg {
  private RoundPeg roundPeg;
  public RoundToSquarePegAdapter(RoundPeg peg) {
    //the roundPeg is plugged into the adapter
    this.roundPeg = peg;}
  public void insert(String str) {
    //the roundPeg can now be inserted in the same manner as a squarePeg!
    roundPeg.insertIntoHole(str);}
}
```

The Adapter Pattern:: Example - III

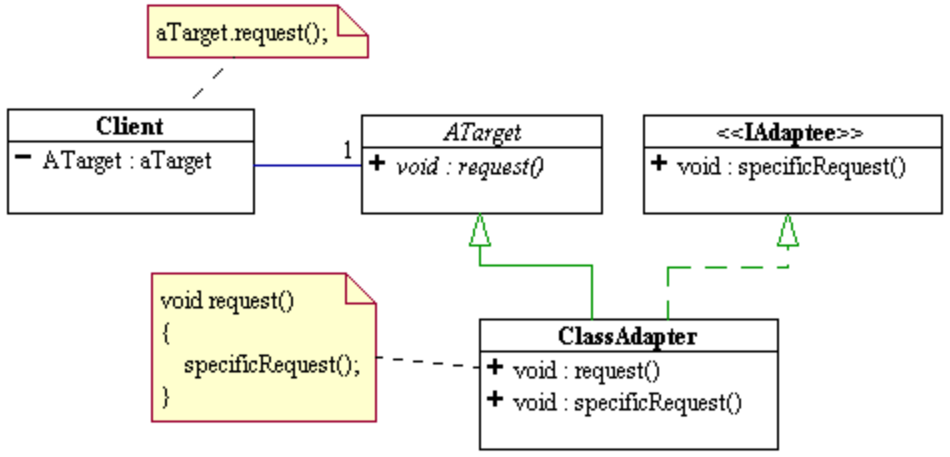
```
// Test program for Pegs.
public class TestPegs {
        public static void main(String args[]) {
            // Create some pegs.
            RoundPeg roundPeg = new RoundPeg();
            SquarePeg squarePeg = new SquarePeg();

            // Do an insert using the square peg.
            squarePeg.insert("Inserting square peg...");

            // Now we'd like to do an insert using the round peg.
            // But this client only understands the insert()
            // method of pegs, not a insertIntoHole() method.
            // The solution: create an adapter that adapts
            // a square peg to a round peg!

            RoundToSquarePegAdapter adapter = new RoundToSquarePegAdapter(roundPeg);
            adapter.insert("Inserting round peg...");}
        }
```

Execution trace:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
```

The Adapter Pattern:: Class Adapters

- Class adapters use multiple inheritance to achieve their goals.
- As in the object adapter, the class adapter inherits the interface of the client's target. However, it also inherits the interface of the adaptee as well.
- Since Java does not support true multiple inheritance, this means that one of the interfaces must be inherited from a Java Interface type.
- Both of the target or adaptee interfaces could be Java Interfaces.
- The request to the target is simply rerouted to the specific request that was inherited from the adaptee interface.



The Adapter Pattern :: Example

- Here are the interfaces for round and square pegs:

```
**
*The IRoundPeg interface.
*/
public interface IRoundPeg {
public void insertIntoHole(String msg);
}

/**
*The ISquarePeg interface.
*/
public interface ISquarePeg {
public void insert(String str);
}
```

The Adapter Pattern :: Example

- Here are the new RoundPeg and SquarePeg classes.
- These are essentially the same as before except they now implement the appropriate interface.

```
// The RoundPeg class.
public class RoundPeg implements IRoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);}
}

// The SquarePeg class.
public class SquarePeg implements ISquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}
```

The Adapter Pattern :: Example

- And here is the new PegAdapter class:

```
/**
* The PegAdapter class.
* This is the two-way adapter class.
*/
public class PegAdapter implements ISquarePeg, IRoundPeg {
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;

  public PegAdapter(RoundPeg peg) {
    this.roundPeg = peg;}
  public PegAdapter(SquarePeg peg) {
    this.squarePeg = peg;}

  public void insert(String str) {
    roundPeg.insertIntoHole(str);}
  public void insertIntoHole(String msg){
    squarePeg.insert(msg);}
}
```

The Adapter Pattern :: Example

- A client that uses the two-way adapter:

```
//Test program for Pegs.
public class TestPegs {
        public static void main(String args[]) {

                // Create some pegs.
                RoundPeg roundPeg = new RoundPeg();
                SquarePeg squarePeg = new SquarePeg();

                // Do an insert using the square peg.
                squarePeg.insert("Inserting square peg...");

                // Create a two-way adapter and do an insert with it.
                ISquarePeg roundToSquare = new PegAdapter(roundPeg);
                roundToSquare.insert("Inserting round peg...");

                // Do an insert using the round peg.
                roundPeg.insertIntoHole("Inserting round peg...");

                // Create a two-way adapter and do an insert with it.
                IRoundPeg squareToRound = new PegAdapter(squarePeg);
                squareToRound.insertIntoHole("Inserting square peg...");
        }
}
```

The Adapter Pattern :: Example

- Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
RoundPeg insertIntoHole(): Inserting round peg...
SquarePeg insert(): Inserting square peg...
```

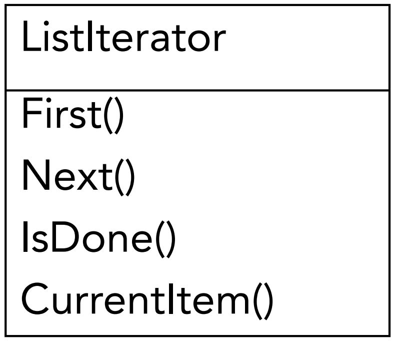
The Iterator Pattern :: Definition & Applicability

- Recurring Problem: How can you loop over all objects in any collection.
- You don't want to change client code when the collection changes.
  - Want the same methods
- Solution: 1) Have each class implement an interface, and 2) Have an interface that works with all collections
- Consequences: Can change collection class details without changing code to traverse the collection

The Iterator Pattern :: Definition & Applicability

- A basic version of the iterator

```
ListIterator<Employee> itr = list.iterator();

for(itr.First(); !itr.IsDone(); itr.Next()) {
   cout << itr.CurrentItem().toString();
```



The Iterator Pattern :: Definition & Applicability

- Java version of the iterator
  - interface Iterator
- boolean hasNext()
  - Returns true if the iteration has more elements.
- Object next()
  - Returns the next element in the iteration and updates the iteration to refer to the next (or have hasNext() return false)
- void remove()
  - Removes the most recently visited element

The Iterator Pattern :: Example

```
public class Range implements Iterable<Integer> {
            private int start, end;

            public Range(int start, int end) {
                  this.start = start;
                  this.end = end;
            }

            public Iterator<Integer> iterator() {
      return new RangeIterator();
   }
```

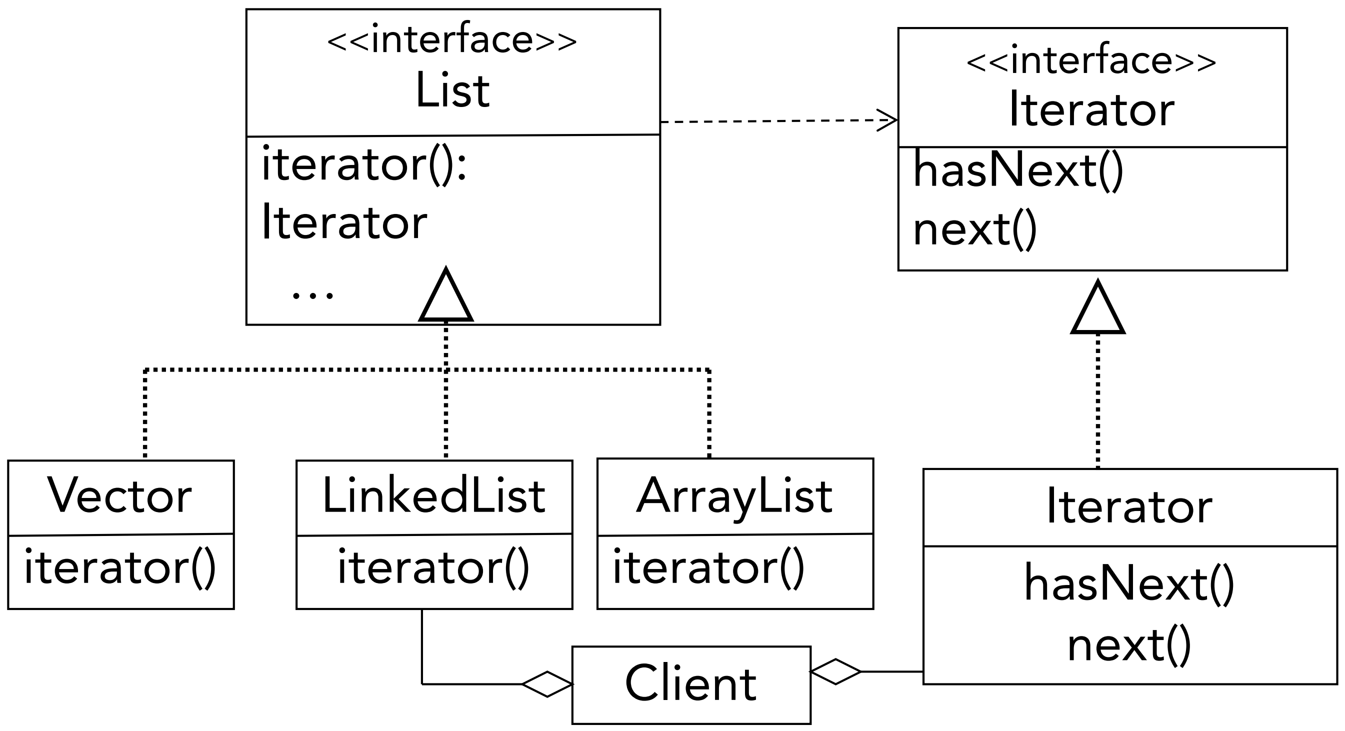
The Iterator Pattern :: Example

```
//Inner class example
private class RangeIterator implements
                Iterator<Integer> {
    private int cursor;

    public RangeIterator() {
        this.cursor = Range.this.start;
    }

    public boolean hasNext() {
        return this.cursor < Range.this.end;
    }

    public Integer next() {
        if(this.hasNext()) {
            int current = cursor;
            cursor ++;
            return current;
        }
        throw new NoSuchElementException();
    }

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

The Iterator Pattern :: Example

```
public static void main(String[] args) {
    Range range = new Range(1, 10);

    // Long way
    Iterator<Integer> it = range.iterator();
    while(it.hasNext()) {
        int cur = it.next();
        System.out.println(cur);
    }
  }
}
```

The Iterator Pattern :: Consequence



Resources

- 〔1〕Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977.
- 〔2〕Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Adisson-Wesley, 1995.
- 〔3〕James W. Cooper, The Design Patterns – Java Companion Elements of Reusable Object-Oriented Software, Adisson-Wesley, 1998.
- 〔4〕James O. Coplien, Advanced C++ Programming Styles and Idioms, Addison-Wesley, Reading, MA., 1992.
- 〔5〕David Geary, Simply Singleton, Java Design Patterns at JavaWorld, April 2003, http://www.javaworld.com/columns/jw-java-design-patterns-index.shtml
- 〔6〕Design Patterns, http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/
- 〔7〕Robert Eckstein, Java SE Application Design With MVC, Oracle Technology Network, March 2007. http://www.oracle.com/technetwork/articles/javase/mvc-136693.html

Thank you

Questions?