Exploring Design Patterns

LearningPatterns, Inc. | www.learningpatterns.com
Collaborative Education Services™

Training | Mentoring | Courseware | Consulting

Student Guide
# Table of Contents - Exploring Design Patterns

## TABLE OF CONTENTS - EXPLORING DESIGN PATTERNS ......................................................... I

### EXPLORING DESIGN PATTERNS ............................................................................................... 1

- Workshop Overview .................................................................................................................. 2
- Workshop Objectives .................................................................................................................. 3
- Workshop Agenda ..................................................................................................................... 5
- This is a Discussion Oriented Class .......................................................................................... 7
- There are Many "Right" Solutions in Design ............................................................................. 8

### SESSION 1 - AN INTRODUCTION TO DESIGN PATTERNS .................................................... 9

- Session Objectives .................................................................................................................. 10

### INTRODUCTION ....................................................................................................................... 11

- What's Our World? .................................................................................................................... 12
- OK - So Just What is a Design Pattern? ................................................................................... 13
- Design Patterns are Not Esoteric ............................................................................................ 14
- Why Use Patterns? ................................................................................................................... 15
- An Example of a Pattern ......................................................................................................... 16
- The Adapter Pattern ............................................................................................................... 17
- Solution - Use the Adapter Pattern ....................................................................................... 18
- Code Solution - EnumerationAdapter ................................................................................... 19
- Reviewing Interfaces & Abstract Classes ............................................................................... 20
- Interface Types ...................................................................................................................... 21
- Interface Definitions ............................................................................................................... 22
- Abstract Methods ................................................................................................................... 23
- Abstract Classes .................................................................................................................... 24
- Using Abstract Classes .......................................................................................................... 25
- Important Principal of OO Design ......................................................................................... 26

### THE ITERATOR PATTERN ....................................................................................................... 27

- Patterns: Traversing A Collection .......................................................................................... 28
- A Simple ArrayList ................................................................................................................... 29
- Using Our ArrayList ............................................................................................................... 30
- A More Sophisticated ArrayList ............................................................................................. 31
- Using Our Simple Collection ................................................................................................. 32
- Lab 1.1 - Examine Collection Traversal with Our Simple Implementation .......................... 33
  - Another Design for Collection Traversal ............................................................................ 36
  - Using Our New Collection ................................................................................................ 37
  - Differences in Traversing Our Collections ....................................................................... 38
- Lab 1.2 - Examine Iterator Based Collection ....................................................................... 39
  - Why is This Important? ....................................................................................................... 42
  - Why is This a Design Pattern? ........................................................................................... 43
  - We Will Expand on Our Design ......................................................................................... 44

### DESIGN PATTERNS - BACKGROUND .................................................................................... 45

- Design Patterns Arise from Architecture .............................................................................. 46
- Christopher Alexander ......................................................................................................... 47
- The Timeless Way ................................................................................................................... 48
- A Core Principles of His Books ........................................................................................... 49
- Patterns In A Pattern Language ............................................................................................ 51
- Sitting Circle (185) ............................................................................................................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERVIEW</td>
<td>118</td>
</tr>
<tr>
<td>MOTIVATION - FORCES AND SOLUTION</td>
<td>119</td>
</tr>
<tr>
<td>MOTIVATION</td>
<td>120</td>
</tr>
<tr>
<td>FACTORY METHOD: ITERATOR USAGE</td>
<td>122</td>
</tr>
<tr>
<td>FACTORY METHOD: GENERAL STRUCTURE</td>
<td>124</td>
</tr>
<tr>
<td>PARTICIPANTS</td>
<td>125</td>
</tr>
<tr>
<td>COLLABORATIONS AND APPLICABILITY</td>
<td>126</td>
</tr>
<tr>
<td>APPLICABILITY</td>
<td>127</td>
</tr>
<tr>
<td>CONSEQUENCES</td>
<td>128</td>
</tr>
<tr>
<td>SAMPLE CODE</td>
<td>129</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td>132</td>
</tr>
<tr>
<td>KNOWN USES AND RELATED PATTERNS</td>
<td>134</td>
</tr>
<tr>
<td>LAB 3.1 - USING FACTORY METHOD</td>
<td>135</td>
</tr>
<tr>
<td>REVIEW QUESTIONS</td>
<td>1402</td>
</tr>
<tr>
<td>LESSON SUMMARY</td>
<td>143</td>
</tr>
<tr>
<td>STRATEGY PATTERN</td>
<td>144</td>
</tr>
<tr>
<td>OVERVIEW</td>
<td>145</td>
</tr>
<tr>
<td>MOTIVATION - FORCES AND SOLUTION</td>
<td>146</td>
</tr>
<tr>
<td>SAMPLE: TAX CALCULATION</td>
<td>147</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>149</td>
</tr>
<tr>
<td>ALTERNATIVES TO STRATEGY</td>
<td>150</td>
</tr>
<tr>
<td>ALTERNATIVES TO STRATEGY</td>
<td>151</td>
</tr>
<tr>
<td>HOW DO WE CHOOSE AMONG ALTERNATIVES</td>
<td>152</td>
</tr>
<tr>
<td>PARTICIPANTS</td>
<td>153</td>
</tr>
<tr>
<td>COLLABORATIONS AND APPLICABILITY</td>
<td>154</td>
</tr>
<tr>
<td>CONSEQUENCES</td>
<td>155</td>
</tr>
<tr>
<td>CONSEQUENCES</td>
<td>156</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td>157</td>
</tr>
<tr>
<td>KNOWN USES AND RELATED PATTERNS</td>
<td>158</td>
</tr>
<tr>
<td>DIFFERENCE FROM FACTORY METHOD</td>
<td>159</td>
</tr>
<tr>
<td>LAB 3.2 - WHAT'S OUR STRATEGY?</td>
<td>160</td>
</tr>
<tr>
<td>REVIEW QUESTIONS</td>
<td>169</td>
</tr>
<tr>
<td>LESSON SUMMARY</td>
<td>170</td>
</tr>
<tr>
<td>DECORATOR PATTERN</td>
<td>172</td>
</tr>
<tr>
<td>OVERVIEW</td>
<td>173</td>
</tr>
<tr>
<td>MOTIVATION - FORCES AND SOLUTION</td>
<td>174</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>175</td>
</tr>
<tr>
<td>PARTICIPANTS AND COLLABORATIONS</td>
<td>179</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>180</td>
</tr>
<tr>
<td>SAMPLE CODE</td>
<td>181</td>
</tr>
<tr>
<td>WRITER AND FILTERWRITER CLASSES</td>
<td>182</td>
</tr>
<tr>
<td>UPCERASFilterWriter Class</td>
<td>183</td>
</tr>
<tr>
<td>CONSEQUENCES</td>
<td>184</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td>186</td>
</tr>
<tr>
<td>KNOWN USES AND RELATED PATTERANS</td>
<td>188</td>
</tr>
<tr>
<td>LAB 3.3 - LET'S DECORATE THIS PLACE</td>
<td>189</td>
</tr>
<tr>
<td>REVIEW QUESTIONS</td>
<td>194</td>
</tr>
<tr>
<td>LESSON SUMMARY</td>
<td>195</td>
</tr>
<tr>
<td>COMPOSITE PATTERN</td>
<td>197</td>
</tr>
<tr>
<td>OVERVIEW</td>
<td>198</td>
</tr>
<tr>
<td>MOTIVATION - FORCES</td>
<td>199</td>
</tr>
<tr>
<td>MOTIVATION - SOLUTION</td>
<td>200</td>
</tr>
<tr>
<td>STRUCTURE - SWING EXAMPLE</td>
<td>201</td>
</tr>
</tbody>
</table>
PATTERNS FOR ENTERPRISE SYSTEMS ................................................................................. 287

OVERVIEW ............................................................................................................................... 288
MEETING THE CHALLENGE - TECHNOLOGIES ................................................................. 289
MEETING THE CHALLENGE - BEST PRACTICES ............................................................... 290
SOME PATTERNS FOR ENTERPRISE SYSTEMS ............................................................... 291
PATTERNS FOR ENTERPRISE SYSTEMS ......................................................................... 292
BUSINESS DELEGATE ......................................................................................................... 293
BUSINESS DELEGATE: SOLUTION ...................................................................................... 294
BUSINESS DELEGATE: STRUCTURE ................................................................................... 295
BUSINESS DELEGATE: CONSEQUENCES ......................................................................... 296
VALUE OBJECT ................................................................................................................... 298
VALUE OBJECT: SOLUTION ............................................................................................... 299
VALUE OBJECT: STRUCTURE ............................................................................................ 300
VALUE OBJECT: CONSEQUENCES .................................................................................... 301
DATA ACCESS OBJECT (DAO) ........................................................................................... 302
DAO: SOLUTION ................................................................................................................ 303
DAO: STRUCTURE ............................................................................................................ 304
DAO: CONSEQUENCES ..................................................................................................... 305
LAZY LOAD ........................................................................................................................ 306
LAZY LOAD: SOLUTION..................................................................................................... 307
LAZY LOAD: CONSEQUENCES ......................................................................................... 308

SESSION 5 - WRAP-UP ....................................................................................................... 309

SESSION OBJECTIVES ....................................................................................................... 310
WHAT HAVE WE DONE? .................................................................................................. 311
SO - WHAT DO YOU THINK ABOUT PATTERNS? ......................................................... 312
WHERE DO WE GO FROM HERE? .................................................................................... 313
DO WE FIT INTO ALEXANDER’S VISION? ....................................................................... 314
DESIGN PATTERNS ISN’T ALL YOU NEED ..................................................................... 315
HAVE FUN .......................................................................................................................... 316
RESOURCES AND FURTHER READING ......................................................................... 317
END OF SESSION ............................................................................................................... 318
EXPLORING DESIGN PATTERNS - LAB SOLUTIONS ...................................................... 319
LAB 1.1 - ITERATION WITHIN ARRAYLIST API ........................................................... 320
LAB 1.2 - ITERATOR-BASED COLLECTION ................................................................. 321
LAB 2.1 - REGARDING THE ITERATOR PATTERN .......................................................... 322
LAB 2.1 - REGARDING DESIGN PATTERNS ..................................................................... 323
LAB 3.1 - FACTORY METHOD .......................................................................................... 324
LAB 3.1 - USING A FACTORY METHOD IN CONTENT ................................................. 325
LAB 3.1 - FACTORY METHOD .......................................................................................... 326
LAB 3.2 - STRATEGY ......................................................................................................... 327
LAB 3.2 - CREATING/USING STRATEGY OBJECTS ....................................................... 328
LAB 3.2 - [OPTIONAL] THE WATERMARK ISSUE ....................................................... 329
LAB 3.2 - [OPTIONAL] CONTENT - WATERMARK ....................................................... 330

IMPLEMENTATION ............................................................................................................... 277
KNOWN USES AND RELATED PATTERNS ......................................................................... 278

FAÇADE PATTERN ................................................................................................................ 279
OVERVIEW ........................................................................................................................... 280
MOTIVATION - FORCES AND SOLUTION ........................................................................ 281
STRUCTURE .......................................................................................................................... 282
PARTICIPANTS AND COLLABORATIONS ....................................................................... 283
CONSEQUENCES / APPLICABILITY ................................................................................... 284
IMPLEMENTATION ............................................................................................................... 285
KNOWN USES ..................................................................................................................... 286

DATA ACCESS OBJECT (DAO) .......................................................................................... 302
DAO: SOLUTION ................................................................................................................ 303
DAO: STRUCTURE ............................................................................................................ 304
DAO: CONSEQUENCES ..................................................................................................... 305
LAZY LOAD ........................................................................................................................ 306
LAZY LOAD: SOLUTION..................................................................................................... 307
LAZY LOAD: CONSEQUENCES ......................................................................................... 308

VALUE OBJECT ................................................................................................................... 298
VALUE OBJECT: SOLUTION ............................................................................................... 299
VALUE OBJECT: STRUCTURE ............................................................................................ 300
VALUE OBJECT: CONSEQUENCES .................................................................................... 301
DATA ACCESS OBJECT (DAO) ........................................................................................... 302
DAO: SOLUTION ................................................................................................................ 303
DAO: STRUCTURE ............................................................................................................ 304
DAO: CONSEQUENCES ..................................................................................................... 305
LAZY LOAD ........................................................................................................................ 306
LAZY LOAD: SOLUTION..................................................................................................... 307
LAZY LOAD: CONSEQUENCES ......................................................................................... 308
DIFFERENCE BETWEEN STRATEGY AND FACTORY METHOD ................................................................. 331
LAB 3.3 - DECORATOR .......................................................................................................................... 332
LAB 3.3 - CUSTOMIZING PLAYBACK ...................................................................................................... 333
LAB 4.1 - COMMAND PATTERN AND LOGGING ............................................................................... 334
LAB 4.1 - COMMAND PATTERN AND UNDO/REDO ......................................................................... 335
LAB 4.1 - COMMAND UNDO ............................................................................................................... 336
LAB 4.1 - PLAYLIST UNDO................................................................................................................. 337
LAB 4.2 - CHAIN OF RESPONSIBILITY ............................................................................................... 338
LAB 4.2 - CHAIN OF RESPONSIBILITY ............................................................................................... 339
END OF SESSION ................................................................................................................................. 340
Exploring Design Patterns

The Java Developer Education Series

LearningPatterns Inc.

services@learningpatterns.com

www.learningpatterns.com

Notes:
# Workshop Overview

- This is a two day course that introduces students to the concepts of **design patterns**
  - These are ways to describe best practices and good designs for computer programming in a way they can be reused

- We include coverage of the general concepts of patterns, and of a representative sampling of patterns that have been identified as being useful to developers first learning patterns
  - The main patterns covered are drawn from the Gang of Four
  - There will also be a brief overview of some of the important patterns used in building distributed / Web based systems.

- It includes lectures, discussion, and group exercises
  - Most of the exercises follow a common fictional case study - JavaTunes, an online music store
Workshop Objectives

◆ You will acquire practical skills in using design patterns to build well designed software systems
  – Of course, this is only the start of a long road of learning

◆ You will use a number of patterns drawn from the core patterns useful for general purpose programming
  – Chain of Responsibility, Command, Composite, Decorator, Façade, Factory Method, Iterator, Proxy, Strategy, Template Method

◆ You will learn the foundations and general ideas underlying design patterns
  – You should also gain the formal knowledge and terminology you need to be fully conversant in this important area

Notes:
Workshop Objectives

◆ You will learn the general details of other patterns
  – Patterns that may be just as useful, but for which there is no time to cover in class

◆ You will see examples of where and when **not** to use design patterns in applications
  – You will discuss and become familiar with the tradeoffs, advantages and disadvantages of the patterns covered

◆ You will be exposed to additional patterns for building distributed systems
  – Business Delegate, Data Access Object (DAO), Lazy Load, Value Object

Notes:
**Workshop Agenda**

- **Session 1: An Introduction to Design Patterns**
  - Introduction
  - The Iterator Pattern
  - Background

- **Session 2: Design Patterns - A More Formal Approach**
  - UML Overview
  - The Gang Of Four Description
  - The GOF Patterns
Workshop Agenda

◆ Session 3: Moving Deeper Into Patterns
   – Factory Method Pattern
   – Strategy Pattern
   – Decorator Pattern
   – Composite Pattern
   – Template Method Pattern

◆ Session 4: And Still Yet More Patterns
   – Command Pattern
   – Chain of Responsibility Pattern
   – Proxy Pattern
   – Façade Pattern
   – Patterns for Enterprise Systems

◆ Session 5: Wrap-up

Notes:
This is a Discussion Oriented Class

◆ It is an exploration into using design patterns to help you write better code

  – We are all exploring together
  – The more you participate, the more you'll learn
  – The more questions you ask, the more the whole class will learn
  – Don't be shy!

◆ The instructor is likely to learn as much as the students

  – No matter how much you've done in design, it seems there is always another way to do things that you haven't thought of

Notes:
There are Many "Right" Solutions in Design

◆ There is no one right way to design something
  – It's all about tradeoffs
  – Tradeoffs depend on what is important in a particular situation
  – This will vary in different contexts, and with different viewpoints

◆ Intelligent people will disagree
  – Given the exact same design forces, different people will think
    that different designs are the best solution
  – That's OK, there is no **Ultimate Design Authority**
  – People are naturally drawn to different ways of doing things
    • Often neither design is "better;" they are just different
  – People also often assign different priorities to different things
  – It's all about the tradeoffs

◆ Of course, given two different designs for the same situation, one
design **Might** be better than the other
  – But you'll need to show that it is due to some particular
    advantage the design gives
  – There is no design that is inherently better than another
Session 1 - An Introduction to Design Patterns

Introduction
The Iterator Pattern
Background

Notes:
**Session Objectives**

- Understand the basics of Object-Oriented programming
- Understand what Design Patterns are in a general sense
- Get a feeling for how Design Patterns work by exploring the Iterator pattern
- Become familiar with how Design Patterns usage evolved in the software development community

Notes:
Introduction

The Iterator Pattern

Background

Notes:
**What's Our World?**

◆ First - Let's set the domain that we are talking about
  – We are dealing with writing software systems
  – We'll see later that patterns can occur in many domains
  – That's interesting, but we'll focus mostly on software here

◆ Second - we are talking about **Object-Oriented** (OO) software systems
  – We will work with patterns that are structured in an OO way

◆ How many people are familiar with OO?
  – How many are comfortable with OO concepts?

◆ We'll briefly review OO soon
OK - So Just What is a Design Pattern?

◆ A solution to a problem in a context

◆ Patterns arise when you look for the commonalities in the systems you are creating

  – Each pattern describes a problem which occurs over and over again in our environment.

  It 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 ¹

◆ We will use the term "pattern" and "design pattern" interchangeably

  – Pattern is a generally accepted shorthand for design pattern

---


◆ As you can see, patterns arise from recurring problems

  – That's part of what makes it a pattern

◆ There are other kinds of patterns, for example "analysis patterns"

  – Generally people use the name "pattern" and it is understood from the context exactly what you are talking about
Design Patterns are Not Esoteric

◆ You already use patterns
  – By clarifying them and working more consciously with them, you can greatly increase the benefit you get from them
  – When you see a pattern described, very often you will think "Of course."

◆ You might ask "Why bother writing patterns that just boil down to advice my grandmother would give me?"
  – Because some patterns are so good and useful that even your grandmother knows them
  – Writing them down makes the context, value and implications of the advice clearer than your grandmother probably did

Notes:
### Why Use Patterns?

- **They have been proven**
  - Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work

- **They are reusable**
  - Patterns provide a ready-made solution that can be adapted to different problems as necessary

- **They are expressive**
  - Patterns provide a common vocabulary of solutions that can express large solutions succinctly
An Example of a Pattern

- Let's start off by looking at one pattern in a very informal way

- Have you ever tried to plug a cord with three prongs into an outlet designed for two prongs?
  - How would you do this?
  - An adapter of course!
The Adapter Pattern

- How about Software: What if you have code expecting an
  **Iterator** that must work with code written for **Enumeration**
- **Enumeration** API
  - `public boolean hasMoreElements()`
  - `public Object nextElement()`
- **Iterator** API
  - `public boolean hasNext()`
  - `public Object next()`
- The issue is – how do we get instances of the existing class to work
  with instances of the new class

```java
public ExistingClass {
    public Enumeration getSomeData(){...}
}

public NewClass {
    public void processSomeData( Iterator i ){...}
}
```

- Iterator was introduced in Java 1.2, Enumeration was used in Java
  1.1
  - Very often, you have to find a way to work with both types at
    the same time
  - That's where the adapter pattern comes in

**Notes:**
Solution - Use the Adapter Pattern

- Adapt an Enumeration (adaptee) to Iterator

Notes:
Code Solution - EnumerationAdapter

- We write a class that exposes an *Enumeration* instance via an *Iterator* interface - as shown below

```java
public class EnumerationAdapter implements Iterator {
    private Enumeration theEnum;

    public EnumerationAdapter(Enumerator enum) {
        theEnum = enum;
    }

    public boolean hasNext() {
        return theEnum.hasMoreElements();
    }
    public boolean next() {
        return theEnum.nextElement();
    }
}
```

// Usage example …
ExistingClass existingC = new Existing();
NewClass newC = new NewClass();
newC.processSomeData(
    new EnumerationAdapter(existingC.getSomeData()));

Notes:
Reviewing Interfaces & Abstract Classes

◆ We'll assume that you are already familiar with the OO model and how it works
  – We will discuss abstract types and interfaces here, which some people have had less exposure to
  – They are important when working with design patterns

◆ Interfaces and abstract classes are used extensively in design patterns
  – They are a way to share code, and define protocols
  – We'll look at them briefly - using Java for our examples
Interface Types

◆ An interface defines a type that is similar to a class
  – In Java, an interface can have properties, but all properties are static final constants
  – It can have methods, but all methods are abstract
  – Interfaces instances can’t be instantiated with the new keyword

◆ Interfaces can be used to declare a protocol, that serves as a blueprint of the common methods that implementing classes are guaranteed to implement and respond to
  – Interfaces are often used to define roles played by objects
  – Whereas a class can define what a type is (via instance data and behavior), an interface can define what roles a type fulfills (via behavior)
  – The use of interfaces is often called programming by contract

◆ A Person can play roles such as Programmer or Instructor.
  – If Programmer is an interface with code and test methods, and
  – Instructor is an interface with teach and debugLabs methods,
  – then Person can play the role of Programmer if it implements the code and test methods, and Person can also play the role of Instructor if it implements the teach and debugLabs methods.
  – This means that you can use a Programmer or Instructor reference with this Person object, i.e., you can treat a Person as simply a Programmer or Instructor.
Interface Definitions

◆ A Java interface definition uses the *interface* keyword, in a way similar to the *class* keyword

```java
// definition of interface Collection
package java.util;
public interface Collection {
    public boolean add(Object o);
    // ...
}
```

◆ The *implements* keyword is used in classes using interfaces
  – Classes provide implementations for all methods of the interface

```java
package java.util;
public class ArrayList implements Collection {
    public boolean add(Object o) {/* ... */}
    // ...
}
```

◆ Note that this isn't exactly how the Java source defines these
  – This is a simplified version used to illustrate the concepts

**Notes:**
Abstract Methods

- **Abstract methods** are methods for which no implementation is defined.
- Implementation is deferred to a subclass which overrides the abstract method with a *concrete method*.
- The method ends with a semicolon (; ) instead of a method body.

```java
public abstract boolean add(Object o);
```

- Interface methods are *implicitly abstract*.
  - In Java, they may be explicitly declared as `abstract`, but generally they aren't.

Notes:
Abstract Classes

- **Abstract classes** in Java are classes which can't be instantiated
  - Abstract classes may, and usually do, have abstract methods
  - Implementation is deferred to a subclass, but this ensures that all subclasses support the method

```java
public abstract class AbstractList implements Collection {
    /* variables, method definitions, etc. */

    public abstract Iterator iterator();

    // ...
}
```

- Again, we've simplified and changed this from the actual implementation

- Another example is `Number`, an abstract class in the `java.lang` package. It looks like this:

```java
public abstract class Number implements Serializable {
    // these methods define the conversion functions
    // that the numeric wrapper classes (which are subclasses of Number) must implement
    public byte byteValue();
    public short shortValue();
    public abstract int intValue();
    public abstract long longValue();
    public abstract float floatValue();
    public abstract double doubleValue();
}
```

Notes:
Using Abstract Classes

- You use an abstract class to specify a protocol or interface (a set of standard methods)
  - And at the same time have the ability to share implementation

- You must create a concrete subclass, which overrides any abstract methods with actual code, in order to define a class which may be instantiated

```java
public class ArrayList extends AbstractList {
    public Iterator iterator {
        // implement ArrayList's iterator() here
    }
}
```

- Can you think of any examples:
  - when you do not have enough information to specify the methods' implementations?
  - when the implementation varies from one subclass to another?

- `java.lang.Number` is the superclass for the numeric wrapper classes.
  - All these subclasses must implement the abstract methods of `Number` to become concrete, instantiable classes.

```
Object

Number

Byte Short Integer Long Float Double
```
Important Principal of OO Design

◆ Program to an interface, not an implementation

– If you need flexibility/variability/change, references should not be to instances of an implementation class, but to an interface

– Clients can remain unaware of the specific type they use

– Clients don't need to care about the details of the implementing class

– Different implementations can be used

– The implementation can be easily changed

– Many design patterns use interfaces

Notes:
The Iterator Pattern

Introduction

The Iterator Pattern

Background

Notes:
## Patterns: Traversing A Collection

- Let's take a look at another pattern in more depth
  - We'll also look at and compare different choice you might make
  - We purposely keep the pattern very simple, in order to focus on the conceptual ideas of patterns themselves
- As an introductory example of using patterns, we're going to look at the issues involved in **traversing a collection**
  - It's one of the most common operations on a collection
- Let's say we have a simple collection that we want the following operations on
  - **Add** an object
  - **Remove** an object
  - **Traverse** all the objects in the collection
- Let's take a look at some types that do this for us
  - We'll do it in some simple, straightforward ways first

---

*Notes:*
A Simple ArrayList

```java
public class ArrayList {  // Implementation not shown
    public boolean add(Object o);

    // Traversal methods
    public int size();      // Return number of elements in list
    public Object get(int index); // Return element at index position
}
```

◆ Let's assume that these methods were the only way to use the list
   – What would a traversal look like?
Using Our ArrayList

// Code fragment

// Assume you have a concrete implementation to create ...
ArrayList aList = new ArrayList();
aList.add("first");
aList.add("middle");
aList.add("last");

// Traversal
for (int i=0; i<aList.size(); i++)
    System.out.println(aList.get(i));
}

◆ Is this a good design in terms of traversal?

  – What are it's strengths?

  – What are it's weaknesses?
A More Sophisticated ArrayList

// Implementation not shown
public class ArrayList implements java.util.Collection {

    public void add(Object o);
    // Other methods not shown ...

    // Traversal methods
    boolean hasNext(); // Returns true if more elements to traverse
    Object next();    // Gets the next element and advances iteration

}

◆ This type encapsulates implementation details of iteration

    – In that sense it has a better design than the previous one

    – Let's take a look at how we use it

    – Loosely based on Java 2 Collection
Using Our Simple Collection

// Code fragment

Collection al = new ArrayList();
al.add("first");
al.add("middle");
al.add("last");

// Traversal
while (al.hasNext()) {
    System.out.println(al.next());
}

◆ Is this a good design in terms of traversal?

– What are it's strengths?

– What are it's weaknesses?
Lab 1.1 - Examine Collection Traversal with Our Simple Implementation
Discuss Our Initial Collection Design

◆ Purpose:
  – Examine our Collection type and explore its strengths and weaknesses

◆ Break up into small groups (you'll do all the labs in the course within these groups)
  – Look at our ArrayList type
  – Look at how we currently do traversals
  – Discuss its strengths and weaknesses
  – Look at the next slide for some issues to consider

◆ After a few minutes, let each group present one issue they found to the whole class
  – Discuss these issues among everyone

◆ The instructor will help you break into groups

Notes:
Here Are Some Issues to Consider

◆ What if we wanted to add another way of traversing the collection?
  – For example, backwards
  – What affect would this have on the collection?

◆ What if we wanted to have two active traversals of the list going on at once?
  – Is this possible with our current type definition?
  – Could we add in this capability using our current design, and what affect would that have on the type?
  – What about more traversals? Three, four ...?

◆ Are there advantages to our current design?

Notes:
Another Design for Collection Traversal

Let’s look at a design that puts responsibility for traversing the list into another type - the *Iterator* type

```java
public interface Collection {
    public void add(Object o);
    // Other methods omitted ...

    // Get an iterator to traverse the collection
    public Iterator iterator();
}
```

```java
public interface Iterator {
    // Traversal methods
    boolean hasNext(); // Returns true if more elements
    Object next();    // Gets the next element and advances iteration

    // Other methods omitted ...
}
```

This is a simplified representation of the `java.util.Collection` interface
Using our New Collection

// Code fragment
Collection c = new java.util.ArrayList();
c.add("first");
c.add("middle");
c.add("last");

// Traversal
Iterator i = c.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}

◆ Does this design give us the same functionality?

– Is it harder or easier to use?

– This design illustrates the core of the Iterator design pattern

◆ The java.util.ArrayList class implements the Collection interface

    – We've only shown a small piece of that here
    – The rest is not central to our discussion

Notes:
Differences in Traversing Our Collections

- Operate directly on collection
- Code to specific implementation

- Operate directly on collection
- Code to generic interface

- Operate on iteration object
- Code to generic interface

Notes:
Lab 1.2 - Examine Iterator Based Collection

Notes:
Discuss Our New Collection Design

◆ **Purpose:**
  – Examine our new *Collection* type and explore its strengths and weaknesses regarding iteration

◆ **Break up into the same groups as the previous lab**
  – Look at our new *Collection* and *Iterator* types
  – Discuss its strengths and weaknesses regarding traversals
  – Look at the next slide for some issues to consider

◆ **After a few minutes, let each group present one issue they found to the whole class**
  – Discuss these issues among everyone
Here Are Some Issues to Consider

◆ What if we wanted to add another way of traversing the collection?
  – For example, backwards
  – What affect would this have on the collection?
◆ What if we wanted to have two active traversals of the list going on at once?
  – Is this possible with our current type definition?
  – What about more traversals? Three, four ...?
  – Are there issues with multiple traversals?
◆ What happens if we remove something from the collection during a traversal?
◆ Are there advantages/disadvantages to our new design?
Why is This Important?

◆ We came up with different designs in these two examples of traversing a collection
  – Our second design has a number of advantages over the first
  – So what's the big deal?

◆ The big deal is that we've:
  – Recognized that this situation happens a lot
  – We've named it, so we can talk about it concisely with our peers
    • Hey - we can use Iterator here!
  – We've looked at some of the consequences of using this design
    • Hey - we can have multiple iterators, but we better be careful if we are modifying the collection
  – We've taken our fuzzy thoughts about traversing a collection and turned them into a clearly defined design with well understood ramifications
## Why is This a Design Pattern?

- We defined a design pattern as "A solution to a problem in a context"
  - With a solution that you can use many times over, and that has advantages over doing it in other ways
  - It is a recurring problem, and we have a good solution

- Our second design seems to fit the definition
  - Of course, our definition was very loose

- As the use of design patterns has evolved, a body of generally accepted knowledge has grown around it
  - Standard ways of describing patterns
  - Agreement on patterns that are important
    - Though there are many schools of thought on that
    - Also, different patterns are important to different tasks

**Notes:**
We Will Expand on our Design

◆ We will soon introduce some formality into describing our design, and in describing patterns in general
  – A UML diagram describing the solution
  – A more formal description of what constitutes our design
  – An explicit coverage of the design ramifications, its advantages, and its disadvantages

◆ First, let's take a look at the history of how design patterns entered into the software development world

  – Once upon a time ...
Session 4 - And Still Yet More

Command Pattern
Chain of Responsibility Pattern
Proxy Pattern
Façade Pattern
Patterns for Enterprise Systems

Notes:
Session Objectives

◆ Learn the following patterns:
  – Command Pattern
  – Chain of Responsibility Pattern
  – Proxy Pattern
  – Façade Pattern

◆ Become familiar with the following patterns for distributed/enterprise development
  – Business Delegate
  – Value Object
  – Data Access Object (DAO)
  – Lazy Load

Notes:
Command Pattern

Chain of Responsibility Pattern
Proxy Pattern
Façade Pattern
Patterns for Enterprise Systems

Notes:
## Overview

- **Intent**: Encapsulate a request as an object  
  - This lets you parameterize clients with different requests, queue or log requests, and support undoable operations

- **Classification**: Object Behavioral

- **Also Known As**: Action, Transaction
Motivation - Forces and Solution

◆ **Forces:** Very often, you need to treat an operation as if it were an object
  – Pass it from one part of your program to another
    • For example, in a user interface a button press may trigger an action, but the button doesn't know anything about the action
  – Store it
    • For example, to keep an undo/redo list of actions

◆ **Solution:** Define a Command type which declares the interface for executing operations
  – Concrete implementations define the operations
  – Instances of the concrete implementations are stored by the invoking class
  – The invoking class executes the command operation as appropriate

◆ The Command pattern is used a great deal in user interface frameworks
  – The Java UI controls use the Command pattern
This is the simplest form of the Command pattern
- The command subclasses do everything that is needed to accomplish the operation
- We'll take a look at an alternative later

**Notes:**
Participants and Collaborations

**Participants:**
- **Command:** Declares an interface for executing an operation
- **ConcreteCommand:** Implements execute() to perform the operation
- **Client:** Creates a ConcreteCommand object
- **Invoker:** Asks a command to carry out the request

**Collaborations:**
- The client creates a ConcreteCommand instance
- An Invoker object stores the ConcreteCommand instance
- The invoker issues a request by calling execute() on the command

Notes:
The Java GUI framework uses the Command pattern
- UI Controls have commands associated with them
- When an event triggers the control, the command is executed

We'll look at a small portion of this event mechanism
- We'll look at some of the interfaces and classes for dealing with action events
  - `interface java.awt.event.ActionListener`
  - `class javax.swing.JButton`
### Sample Code

```java
package java.awt.event;
import java.util.EventListener;

// This corresponds to the Command interface
public interface ActionListener extends EventListener {
    /** Invoked when an action occurs. */
    public void actionPerformed(ActionEvent e);
}

import java.awt.event.*;

// This is a (trivial) concrete command class
public class HelloCommand implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello");
    }
}
```

---

Notes:
import javax.swing.*;

public class TestHelloCommand {

    // This is our client. The Swing specific details aren't
    // important, except for those related to the Command pattern
    public static void main(String[] args) {
        JFrame f = new JFrame("TestCommand");

        // The JButton is the Invoker
        JButton b = new JButton("Say Hello");
        // We add an instance of ConcreteCommand to the Invoker
        b.addActionListener(new HelloCommand());

        f.getContentPane().add(b);
        f.pack();
        f.setVisible(true);
    }
}

Notes:
Sample Code

- When we run this program and press the button, then the command is executed
  - Which just prints Hello on the console

![Command output]

Button Pressed

Notes:
Consequences

- **Command decouples** the object invoking the operation (Invoker) from the one that knows how to perform it (Command)
  - Which is what we need in something like a UI toolkit
  - A `JButton` has no idea what to do if it is pressed
  - It just tells something else "Hey I'm pressed - do the right thing"

- **Commands are first class objects**
  - They can be manipulated, extended, etc.
  - In our sample, we created an instance of `HelloCommand`
  - We could easily store commands if we wanted to keep something like a history list
  - We may need to make copies of the command, since a single command instance may be executed many times
Consequences

◆ Creating new commands is easy
  – *HelloCommand* just implemented the appropriate interface
  – Notice that we didn’t need to change anything in any of the Swing classes
  – A JButton is still just a JButton

◆ You end up with more types and objects
  – Possibly many more types if the commands are very fine grained
  – May be harder to maintain/debug
  – One strategy to help with this, is to have multiple related operations in a single command type
  – For example, all the operations associated with a use case
  – This often makes sense as they may be able to share code
  – It reduces the number of types, and possibly instances

Notes:
How intelligent should the command be?
- Our (very simple) command did everything by itself
- Sometimes commands collaborate with another object (Receiver) that actually executes the command

```
Client creates

Invoker

Command
  (abstract)
  execute()

ConcreteCommand
  receiver : Receiver
  execute()
  receiver->action();

Receiver
  action()
```

Notes:
import java.awt.event.*;
import java.io.*;

public class WriterCommand implements ActionListener {

    // The Receiver
    private Writer m_writer;

    WriterCommand(Writer w) {
        m_writer = w;
    }

    public void actionPerformed(ActionEvent e) {
        try {
            // The receiver accomplishes the work
            m_writer.write("Hello");
            m_writer.flush();
        } catch (IOException ex) { ex.printStackTrace(); }
    }
}
Sample Code With Receiver

```java
import javax.swing.*;
import java.io.*;

public class TestWriterCommand {

    public static void main(String[] args) {
        JFrame f = new JFrame("TestCommand");
        JButton b = new JButton("Write Hello");
        try {
            // Our client passes in a receiver to the command
            b.addActionListener(new WriterCommand(
                new FileWriter("out.txt")));
        }
        catch(IOException ex) { ex.printStackTrace(); }
        f.getContentPane().add(b);
        f.pack();
        f.setVisible(true);
    }
}
```

Notes:
II. Understanding Command Patterns

A. Introduction to Command Patterns

- The command pattern is a behavioral design pattern that allows you to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

B. Application of Command Pattern

- Use cases:
  - Task composition: Combining a group of commands to create a new command.
  - Task decomposition: Breaking down a complex task into a series of commands.
  - Accessibility: Providing a uniform interface to invoke commands.
  - Reversibility: Enabling the ability to undo and redo actions.
  - Usability: Enhancing the user experience with an intuitive command interface.

C. Implementing Command Pattern

- Components of Command Pattern:
  - Command: Specifies an interface for executing an action.
  - Receiver: Objects that perform commands.
  - Invocation: How commands are executed.
  - Invoker: An object that can invoke a command.
  - Aggregate: An object that holds a collection of commands.

D. Advantages of Command Pattern

- Modular design:
  - Commands are independent, making the system more modular.
  - Commands can be reused across different parts of the system.

- Redundancy control:
  - Minimizes redundancy in the system.
  - Reduces code duplication.

- Dynamic control:
  - Commands can be added, removed, or replaced at runtime.
  - Allows for dynamic composition of tasks.

- Improved design flexibility:
  - Enables easier modification and extension.
  - Supports variability in the system.

E. Challenges of Command Pattern

- Resource management:
  - Ensuring proper resource management when dealing with commands.

- Security implications:
  - Considering security aspects when executing commands.

F. Conclusion

- Command pattern is a powerful tool for implementing flexible and reusable systems.

III. Exploring Design Patterns

A. Undo and Redo

- Can be supported by commands.

- Commands need to provide a way to reverse their action:
  - Undo, Redo operations.

- This is easy conceptually:
  - Often hard in practice:
    - What does the command need to keep track of to undo an operation?
    - Some commands can't be undone.
    - Will repeated Do/Undo/Redo cycles have side effects?
      - For example, in a calculator, you may get rounding errors.
    - You may need to make a copy of the command item since it may get executed repeatedly.

Notes:
Known Uses

◆ Known Uses:

– The Java UI framework, and many other UI frameworks
  • Apples MacApp
  • Many of the C++ frameworks, e.g. InterViews

– The Struts framework for building Servlet/JSP presentation layers uses the command pattern
  • As do some PHP frameworks

Notes:
Lab 4.1 – Whoops! Didn’t Mean to do That
**Objectives**

- **Purpose:** Explore the Command pattern
  - You will focus on the complexities of using the Command pattern for Logging and Undo/Redo
  - While doing this, you'll need to think about the responsibilities that each component needs to carry out

- In this lab, we'll take as a given that you're using the Command pattern
  - We'll think about how you'd use it a couple of different situations, and how this affects the way the Command objects are designed
The JavaTuner Playlist

- The JavaTuner allows users to create a playlist of content that they want to play
  - JavaTuner plays each item on the list in order
  - For example, you might put 10 MP3 titles on the playlist, press play, and JavaTuner will play through all 10 songs
- JavaTuner has the following operations to manage the playlist
  - Add item to playlist (at the current cursor position)
  - Remove selected item from playlist
  - Move selected item up one slot in the playlist order
  - Move selected item down one slot in the playlist order
  - Each operation has a button on the JavaTuner interface for users to invoke it
- Assume that we have buttons on JavaTuner to manage the playlist

Notes:
Logging

- Assume we're using the command pattern to carry out the actions of the button, and have commands for each operation
  - AddCommand, RemoveCommand, MoveUpCommand, etc…

- We want the ability to log how users are using JavaTuner
  - Knowing how users use JavaTuner helps us improve the design

- Each time a user invokes an operation, we want to log it

- How might we include the logging functionality in our playlist commands?
  - Think about where the logging should be performed, and how it might be implemented
  - Don’t get too detailed, just enough to make sure you understand how it will work to your own satisfaction
We also want to have the ability to Undo changes to the playlist
  – Just in case your favorite DJ accidentally deletes the playlist for your party that he or she has worked on for 12 hours

Think about what information has to be recorded in order to have Undo functionality
  – Think about what needs to be done to undo something like a delete
  – Think about being able to undo multiple commands

Think about what components should be responsible for handling the different responsibilities required for undo
**Undo**

- Come up with a design that supports the undo functionality
  - Determine what each component (JavaTuner, Playlist, Commands) should be responsible for, and how they collaborate
  - As before, go into enough detail that you have are confident that your design will work

- **Optional**: Think about what needs to change to support Redo functionality
  - Add this to your design
### Review Questions

- Why would we want to use the Command pattern?
- In a general sense, what should a command do?
- What advantages does the Command pattern give?
- Are there disadvantages?
Lesson Summary

- The Command pattern is used when we want to treat a request as an object
  - We create a class that encapsulates the request
  - We can then pass it around, store it, etc.

- The command class can be as simple or complex as you need
  - This will mostly be determined by your application

- For complex requests, it may be cleaner to separate out the actual request code into another class
  - The command class is then just used to pass on the request to the actual receiver
Lesson Summary

- Command allows us to manipulate requests as we do other objects
  - Store them or use them again (e.g. Undo)

- Command makes it easy to add new functionality to frameworks

- In big systems, Command may lead to the creation of a large number of types and objects
  - But there are strategies to deal with this issue
  - Making the commands do more is one strategy

Notes: