Lecture 13: Design Patterns

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2005

Pattern Resources

- Pattern Languages of Programming
  - Technical conference on Patterns
- The Portland Pattern Repository
  - http://c2.com/ppr/
- Patterns Homepage
  - http://hillside.net/
- Go to page then click on “Patterns tab”
Design Patterns

- Addison-Wesley book published in 1995
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Known as “The Gang of Four”
- Presents 23 Design Patterns
- Material in this lecture and lecture 26 is drawn from this book, and is thus copyright © 1995 by Addison-Wesley Publishing Company

February 22, 2005 © University of Colorado, Boulder, 2005

What are Patterns?

- Christopher Alexander talking about buildings and towns
  - “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”

February 22, 2005 © University of Colorado, Boulder, 2005
Patterns, continued

- Patterns can have different levels of abstraction
- In Design Patterns (the book),
  - Patterns are not classes
  - Patterns are not frameworks
  - Instead, Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

So, patterns are formalized solutions to design problems

- They describe techniques for maximizing flexibility, extensibility, abstraction, etc.
- These solutions can typically be translated to code in a straightforward manner
Elements of a Pattern

Pattern Name
- More than just a handle for referring to the pattern
- Each name adds to a designer’s vocabulary
- Enables the discussion of design at a higher abstraction

The Problem
- Gives a detailed description of the problem addressed by the pattern
- Describes when to apply a pattern
- Often with a list of preconditions

The Solution
- Describes the elements that make up the design, their relationships, responsibilities, and collaborations
- Does not describe a concrete solution
- Instead a template to be applied in many situations
Elements of a Pattern, continued

- The consequences
  - Describes the results and tradeoffs of applying the pattern
    - Critical for evaluating design alternatives
  - Typically include
    - Impact on flexibility, extensibility, or portability
    - Space and Time tradeoffs
    - Language and Implementation issues

Design Pattern Template

- Pattern Name and Classification
  - Creational
  - Structural
  - Behavioral
- Intent
- Also Known As
- Motivation and Applicability
- Structure and Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns
Examples

- Singleton
- Factory Method
- Adapter

Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it

- Motivation
  - Some classes represent objects where multiple instances do not make sense or can lead to a security risk (e.g. Java security managers)
Singleton, continued

Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton Structure

```java
Singleton

static Instance()
public SingletonOperation()
public GetSingletonData()

private static uniqueInstance
private singletonData
{return uniqueInstance}
```
Singleton, continued

- **Participants**
  - Just the Singleton class

- **Collaborations**
  - Clients access a Singleton instance solely through Singleton's Instance operation

- **Consequences**
  - Controlled access to sole instance
  - Reduced name space (versus global variables)
  - Permits a variable number of instances (if desired)

---

Implementation

```java
import java.util.Date;

public class Singleton {
    private static Singleton theOnlyOne;
    private Date d = new Date();

    private Singleton() {
    }
    public synchronized static Singleton instance() {
        if (theOnlyOne == null) {
            theOnlyOne = new Singleton();
        }
        return theOnlyOne;
    }
    public Date getDate() {
        return d;
    }
}
```
Using our Singleton Class

```java
public class useSingleton {
    public static void main(String[] args) {
        Singleton a = Singleton.instance();
        Singleton b = Singleton.instance();
        System.out.println("" + a.getDate());
        System.out.println("" + b.getDate());
        System.out.println("" + a);
        System.out.println("" + b);
    }
}
```

Output:
Sun Apr 07 13:03:34 MDT 2002
Sun Apr 07 13:03:34 MDT 2002
Singleton@136646
Singleton@136646

Names of Classes in Patterns

Are the class names specified in a pattern required?

- No!

  Consider an environment where a system has access to only one printer

  Would you want to name the class that provides access to the printer “Singleton”??!!

  - No, you would want to name it something like “Printer”!

On the other hand

- Incorporating the names of a pattern’s roles can help to communicate their use to designers

  “Oh, I see you have a “PrinterObserver” class, are you using the Observable design pattern?”
Names, continued

- So, if names are unimportant, what is?
  - Structure!

- We can name our Singleton class anything so long as it
  - has a private or protected constructor
    - need a protected constructor to allow subclasses
  - has a static “instance” operation to retrieve the single instance

Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate

- Also Known As
  - Virtual Constructor

- Motivation
  - Frameworks define abstract classes, but any particular domain needs to use specific subclasses; how can the framework create these subclasses?
    - See example on page 107 of the design patterns book
Factory Method, continued

- **Applicability**
  - Use the Factory Method pattern when
    - a class can’t anticipate the class of objects it must create
    - a class wants its subclasses to specify the objects it creates
    - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
  - **In a nutshell**
    - A “factory” object creates “products” for a client; the type of products created depends on the subclass of the factory object used; the client knows only about the factory, not its subclasses

Factory Method, continued

- **Participants**
  - **Product**
    - Defines the interface of objects the factory method creates
  - **Concrete Product**
    - Implements the Product Interface
  - **Creator**
    - declares the Factory method which returns an object of type Product
  - **Concrete Creator**
    - overrides the factory method to return an instance of a Concrete Product
Factory Method Structure

- **Product**
  - **ConcreteProduct**
  - **Creator**
    - `FactoryMethod()`
    - `AnOperation()`
  - **ConcreteCreator**
    - `FactoryMethod()`

```
Product p = FactoryMethod();
return new ConcreteProduct();
```

Factory Method Consequences

- Factory methods eliminate the need to bind application-specific classes into your code
- Potential disadvantage is that clients must use subclassing in order to create a particular `ConcreteProduct`
  - In single-inherited systems, this constrains your partitioning choices
- Provides hooks for subclasses
- Connects parallel class hierarchies
  - See page 110 of the design patterns book
Implementation

- See code example (available on class website)
- A factory can return balloons of different colors
  - The factory hides several specific creators and cycles among them to create balloons
- A client retrieves multiple balloons and displays their colors

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces
- Also Known As
  - Wrapper
- Motivation
  - Sometimes a toolkit class that is designed for reuse is not reusable because its interface does not match the domain-specific interface an application requires
  - Page 139-140 of Design Patterns provides an example
Adapter, continued

Applicability

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need
- you want to create a reusable class that cooperates with unrelated or unforeseen classes

Participants

- Target
defines the domain-specific interface that Client uses
- Client
collaborates with objects conforming to the Target interface
- Adaptee
defines an existing interface that needs adapting
- Adapter
adapts the interface of Adaptee to the Target interface
Adapter Structure

Class Adapter

Client

Target

Request()

Adapter

Request()

Adaptee

SpecificRequest()

target.Request()

Adapter Structure

Object Adapter

Client

Target

Request()

Adaptee

SpecificRequest()

adaptee

SpecificRequest()

 adaptor

adaptee.SpecificRequest()
Adapter, continued

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request

Consequences

- Class Adapters
  - adapts Adaptee to Target by committing to concrete Adapter class; Adapter can override Adaptee behavior

- Object Adapters
  - lets a single Adapter work with many Adaptees; makes it harder to override Adaptee behavior

Implementation

- See code example (available on class website)

- Very simple implementation of the object adapter but it shows the basic idea

  - object adapter chosen simply because I don’t like multiple inheritance :-)