B16 Design Patterns

Lecture 1

Victor Adrian Prisacariu
http://www.robots.ox.ac.uk/~victor
Course Content

I. Code Design Patterns
   1. Motivation, Classification, UML
   2. Creational Patterns
   3. Structural Patterns
   4. Behavioral Patterns

II. Algorithm Design Patterns

Slides on Weblearn
Course Content

I. Code Design Patterns
   1. Motivation, Classification, UML
   2. Creational Patterns
   3. Structural Patterns
   4. Behavioral Patterns

II. Algorithm Design Patterns

Slides on Weblearn
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.

Timeless Way of Building, *Christopher Alexander*, 1979
• **Carpenter 1:** How do you think we should build these drawers?

• **Carpenter 2:** Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...
• **Carpenter 1**: Should we use a dovetail joint or a miter joint?

• **Subtext:**
  – miter joint: cheap, invisible, breaks easily;
  – dovetail joint: expensive, beautiful, durable.
(Software) Design Patterns

A (software) design pattern is a time-tested solution to a common problem.

- captures design experience;
- allow that expertise to be transferred;
- enable a shared design language, improving communication, easing documentation.
Code Design Pattern Books

Original Gang of Four Book

• Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson

• Uses C++ (and Smalltalk)

Many, many other online resources, e.g. C# 3.0 Design Patterns, Judith Bishop; lots of Java books, etc.
Before we can start describing patterns ... UML

• The Unified Modeling Language (UML) is a standard modeling language (language for drawing diagrams) that allows developers to express software designs using a visual notation.

• UML is (somewhat) universally disliked.

• We don’t care about all of UML – just UML class diagrams.
  – Allow us to visualise each design pattern at a high level.
  – Inform us about the **class structure** and the **relationships between classes** (inheritance and containment).
Associations capture relationships between objects:

• Structural: An Animal has a Tail
• Interactions: An Animal plays with a Toy
# Associations

<table>
<thead>
<tr>
<th>Generic association</th>
<th>One way association</th>
</tr>
</thead>
<tbody>
<tr>
<td>A and B call and access each other’s elements</td>
<td>A can access B’s elements, but B cannot access A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A has B (B is “part of” A)</td>
<td>A has a B (B is “part of” A)</td>
</tr>
<tr>
<td>B does not depend on A</td>
<td>B depends on A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>B inherits from A</td>
<td>B implements A</td>
</tr>
</tbody>
</table>
Associations

An **Animal** (abstract class/interface here)
- must eat **Food**.
- may dress with **Clothes**.
- must own a **Tail**.
- can own a **Toy**.

A **Cat** is a type of **Animal**.
• We can represent various multiplicities, eg. 0..*, 1 to 1, ...

<table>
<thead>
<tr>
<th></th>
<th>Human</th>
<th>Animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>● +name: string +Talk()</td>
<td>0..1 0..*</td>
<td>+name: string +MakeSound()</td>
</tr>
</tbody>
</table>

• Example:
  – A **Human** can have an arbitrary number of **Animals**
  – An **Animal** has 0 or 1 **Humans**
Multiplicities

• We can represent various multiplicities, eg. 0..*, 1 to 1, ...

• Example:
  – An Animal can have 0 or 1 Tails
  – A Tail belongs to exactly one Animal and cannot exist without the Animal
My pseudocode

- I will show lots of simple code examples.
- For that I will use a simplified version of C++ (no pointers 😊).

```java
public class Animal {
    public string name;
    public void MakeSound() {
    }
}

public class Animal {
    private string name;
    public void MakeSound() {
    }
}

public interface Animal {
    void MakeSound();
}

public class Animal {
    private string name;
    public abstract void MakeSound();
    
}
```
My pseudocode

- I will show lots of simple code examples.
- For that I will use a simplified version of C++ (no pointers 😊).

```java
public interface Animal {
  void MakeSound();
}

public class Cat : Animal {
  public string name;
  public void MakeSound();
}
```
The GoF books defines 23 patterns, split into three fundamental groups:

- **Creational**
  - They abstract the instantiation process.
  - Make systems independent on how objects are compared, created and represented.

- **Structural**
  - Focus on how classes and objects are composed to form (relatively) large structures.
  - Generally use inheritance.

- **Behavioral**
  - Describe how different objects work together.
  - Focus on
    - The algorithms and assignment of responsibilities among objects.
    - The communication and interconnection between objects.
Elements of a Pattern

• **Name**: Important as a part of the design vocabulary.
• **Problem**: When the apply the pattern.
• **Solution**: Design elements, along with their relationships, responsibilities and collaborations.
Creational Patterns: Singleton

• We can only have a single instance of one class.

• Example:
  – We have global data and operations:
    • configuration files: you may only have one file on disk containing app settings.
  – We can only have a single instance of a resource
    • a library to access specialized neural network hardware.
public class Configuration
{
    // Internal singleton variable
    private static Configuration config = null;

    // Private constructor
    private Configuration() {
    }

    // Public access method
    public static Configuration Instance() {
        if (config == null) config = new Configuration();
        return config;
    }
}

Creational Patterns: Prototype

• You want to create new objects quickly from previously created objects.

• Example:
  – Same as before, but now you are allowed to have multiple instances of `Configuration` class.
public interface Prototype { Prototype Clone(); }

public class Configuration : Prototype
{
    public string setting;

    public Prototype Clone() {
        Configuration newConfig = new Configuration();
        newConfig.setting = this.setting;
        return newConfig;
    }
}

class Example
{
    static void Main(string[] args)
    {
        Configuration conf1 = new Configuration("setting");
        Configuration conf2 = (Configuration)conf1.Clone();
    }
}
Creational Patterns: Factory Method

• You need to create objects of a certain type.
• Each subclass must be able to define its own domain.

• Example:
  – We have an abstract `SortingAlgorithm` class, and we want to deal with the algorithm in a way that is agnostic of the specific algorithm.
  – We need to create various types of `SortingAlgorithms` including, e.g. `BubbleSort` and `MergeSort`. 
**Creational Patterns: Factory Method**

```java
public class BubbleSort : SortingAlgorithm {
    public void Sort() { /* bubble sort method */ }
}

class MergeSort : SortingAlgorithm {
    public void Sort() { /* merge sort method */ }
}

class AlgorithmCreator {
    public static SortingAlgorithm FactoryMethod(string type) {
        if (type == "bubbleSort") return new BubbleSort();
        return new MergeSort();
    }
}

class AppThatNeedsSorting {
    static void Main(string[] args) {
        SortingAlgorithm algorithm = AlgorithmCreator.FactoryMethod("bubbleSort");
    }
}
```

**public interface SortingAlgorithm**

```java
public interface SortingAlgorithm {
    void Sort();
}
```
Creational Patterns: Builder

• You need to create objects of a certain type.
• You want to decouple the creation of the object from their specific parameters.

• Example:
  – We have an abstract SortingAlgorithm class, and we want to deal with the algorithm in a way that is agnostic of the specific algorithm.
  – We need to create various types of SortingAlgorithms including, e.g. BubbleSort and MergeSort.
  – You want to specify the parameters for the creator as a class, e.g. SortParameters.
Creational Patterns: Builder
public interface SortingAlgorithm {
    void Sort();
}

public class BubbleSort : SortingAlgorithm {
    public void Sort() {
        /* bubble sort method */
    }
}

public class MergeSort : SortingAlgorithm {
    public void Sort() {
        /* merge sort method */
    }
}

public class SortParameters {
    public string type;
    public SortParameters(string type) { this.type = type; }
}

public class AlgorithmCreator {
    public static SortingAlgorithm FactoryMethod(SortParameters parameters) {
        if (parameters.type == "bubbleSort") return new BubbleSort();
        return new MergeSort();
    }
}

class AppThatNeedsSorting {
    static void Main(string[] args) {
        SortParameters sortParams = new SortParameters("bubbleSort");
        SortingAlgorithm algorithm = AlgorithmCreator.FactoryMethod(sortParams);
    }
}
Creational Patterns: Abstract Factory

• We now need to create similar classes, as before.
• We need separate, similar but different, object creators.

• Example:
  – As before, an abstract `SortingAlgorithm` class, but now we also have a `SearchingAlgorithm` class, both (potentially) inheriting on the `Algorithm` class.
  – We need to create various types of `SortingAlgorithms` including, e.g. `BubbleSort` and `MergeSort`.
  – We need to create various types of `SearchingAlgorithms` including, e.g. `DumbSearch` and `BinarySearch`. 
Creational Patterns: Abstract Factory
public interface Algorithm { }

public interface SortingAlgorithm : Algorithm { void Sort(); }
pUBLIC class BubbleSort : SortingAlgorithm { public void Sort() { /*bubble*/ } }
pUBLIC class MergeSort : SortingAlgorithm { public void Sort() { /*merge*/ } }

public interface SearchingAlgorithm : Algorithm { void Search(); }
pUBLIC class DumbSearch : SearchingAlgorithm { public void Search() { /*dumb*/ } }
pUBLIC class BinarySearch : SearchingAlgorithm { public void Search() { /*binary*/ } }

Creational Patterns: Abstract Factory
public interface AlgorithmCreator { Algorithm FactoryMethod(string type); }

public class SortingCreator: AlgorithmCreator {
    public Algorithm FactoryMethod(string type) {
        if (type == "bubbleSort") return new BubbleSort();
        return new MergeSort();
    }
}

public class SearchingCreator : AlgorithmCreator {
    public Algorithm FactoryMethod(string type) {
        if (type == "dumbSearch") return new DumbSearch();
        return new BinarySearch();
    }
}

class AppThatNeedsSorting {
    static void Main(string[] args) {
        AlgorithmCreator sortingCreator = new SortingCreator();
        Algorithm algorithm = sortingCreator.FactoryMethod("bubbleSort");
    }
}
Creational Patterns – Summary

• **Singleton**: limits object creation to only one instance.
• **Prototype**: creates objects by cloning existing objects.
• **Factory method**: method that creates related objects.
• **Builder**: separates construction and representation.
• **Abstract factory**: groups object factories that have a common theme.

Summary of Lecture 1

• Generic motivation of using design patterns.

• Overview of UML diagrams.

• Classification the various types of design patterns.

• Creational design patterns.