Object Orientation with Design Patterns

Lecture 1 : An Introduction

Lecturer: Luke Raeside
Room: E020
Email: luke.raeside@itb.ie

Course Objectives

At the end of this semester the student will:

- Be able to describe **what design patterns are** and what **impacts** they have on object oriented software design
- Understand some of the **more common design patterns** as described by the Gang of Four
- **Apply in the lab** some of the more widely used design patterns
- Be able to identify and apply design patterns in your **projects** where appropriate

Recommended Books

- Java Design Patterns – A Tutorial
  James W. Cooper
- Design Patterns – Elements of Reusable Object-Oriented Software
  Gamma, Helm, Johnson, and Vlissides. (Gang of Four)
- Design Patterns Explained
  Alan Shalloway / James R. Trott
- Design Patterns Java Workbook
  Stephen John Metsker

Web Sites

- http://www.dofactory.com/Patterns/Patterns.aspx
- YouTube – plenty of descriptions about different patterns (but read about the patterns first so that you can critically analyze the video content)

No matter how much you read about something

The best way to really learn is to put it into **practise**

Assessment

Exam 50%	Continuous Assessment (50%) **Weekly Programs in Lab** **50% (500 points)** **2 submissions total (labs 1-5 and 5-6)** Approx 50 points each week with a mix of simple and more challenging exercises

Course Overview

- Object-oriented programming is not just about applying inheritance and polymorphism etc.....it's about **improving software reuse** and **designing systems effectively**

- The fundamentals of object-oriented are the **building blocks** of the OO paradigm, you cannot expect to build a house without knowing how to put the blocks together

- Equally you cannot expect to build skyscrapers and other huge architectural feats without examining the **quality of your design**

Course Overview

- Software is no different, you may have the best code writers in the world but someone has to **examine the quality and robustness** of the software

- It was **this type of thinking** that lead to the development of design patterns.

- As the software industry matures so too must the **approach to building systems,** the software industry is much younger than traditional architecture for building but there's plenty to learn from already (solutions to problems)

Objectives of Today's lecture

- At the end of today's lecture the student will be able to:
  - Understand what is meant by a **design pattern**
  - Understand the **function of design patterns** in modern OOAD\OOP
  - Describe the **MVC** and **Factory pattern** (first examples we meet of design patterns)
  - Go to the lab and do the exercises at the end on Moodle called 'Labsheet 1 MVC and Factory' (i.e. practice!)

What are Design Patterns ?

- One of the main reasons computer science researchers began to recognize design patterns was **to satisfy the need for good, simple, and reusable solutions.**

- The term *design pattern* can sound a little bit formal to the beginner, but in fact a design pattern is just a **convenient way of reusing object-oriented code between projects and between programmers**.

- The idea behind design patterns is simple:

  - **To catalog common characteristics between objects that programmers have often found useful.**

Why study design patterns?

- **Reuse existing**, high quality solutions to **commonly recurring** problems
- Establish **common terminology** to improve communications within software groups/teams
- Shift the level of thinking to a **higher level**
- Decide whether I have the **right design**, not just **one that gets the job done**

Why study design patterns?

- Improve **individual learning** and the group learning

- Facilitate adoption of **improved design alternatives** even if patterns are not directly applied

From Architecture to Software

- **Christopher Alexander** was an architect who began to examine design of buildings and other architectural things

- Alexander was convinced that in architectural design there were repeating themes which lead to what was understood as a **QUALITY** design

- Even though every new design differed from the last...there seemed to be **similarities between quality designs**

  "Architectural structures differ from each other, even if they are the same type"....

- Alexander christened these similarities as **"Patterns"**

- Alexander defined a pattern as : **"a solution to a problem in context"**

From Architecture to Software

Here's another **quote** that's worth examining as a precursor to studying 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**"

From Architecture to Software

- Alexander also stated that a pattern has the following:
  - A **name**
  - A **purpose**...what problem it solves
  - **How** to accomplish the purpose
  - **Constraints and forces** we have to consider in order to accomplish it

From Architecture to Software

- In the 1990's some smart software developers started to wonder if these "patterns" also **existed in software design**

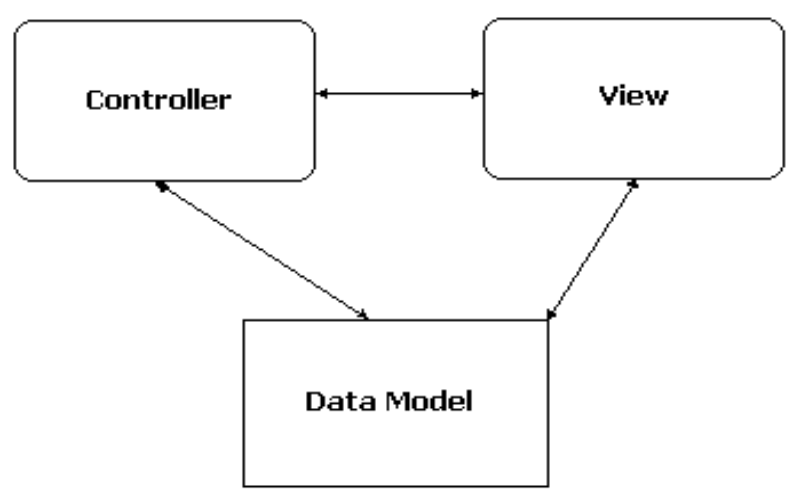
- It was in **1995 that the Gang of Four** published the book "**Design Patterns – Elements of Reusable Object-Oriented Software**" (which by the way doesn't have any Java in it!!!)

From Architecture to Software

- What did these guys accomplish:
  - They applied the idea of **design patterns to software design**
  - Described a structure to catalog and describe software
  - Cataloged **23 patterns** in all
  - Postulated **OO strategies** and approaches based on these design patterns
- Please note these patterns were **not invented** by these guys...
- They were there as a **result of collective experiences...**
- These guys identified what was already there AND **put down on paper**!!!

What are Design Patterns ?

- A common pattern cited in early literature on programming frameworks is **Model-View-Controller (MVC)** for Smalltalk [Krasner and Pope, 1988], which divides the user interface problem into three parts.



- **Data Model**, which contains the **computational parts** of the program
- **View**, which presents the **user interface**
- **Controller**, which **interacts** between the user and the view

What are Design Patterns ?

- Each aspect of the problem is a **separate object**, and each has its **own rules for managing its data.**

- Communication between the user, the graphical user interface, and the data should be **carefully controlled**; this separation of functions accomplishes that.

- Three objects talking to each other using **this restrained set of connections** is an example of a powerful design pattern.

- In other words , a design pattern describes **how objects communicate** without becoming **entangled in each other's data models and methods.**

What are Design Patterns ?

- NOTE: This idea of separation has always been a goal of object-oriented development. **Design patterns just provide an easier way of achieving it !**

- So if you have been trying to keep objects minding their own business chances are that you are already using a common design pattern (shifting responsibility – the "Next Class" analogy)

- It is interesting that the (MVC) pattern was **used throughout Java 1.2 as part of the JFC** (in particular Swing)

- Microsoft introduced the Document View Architecture in Visual C++ 4 as a way for developers to separate application interfaces, views, and data. This is a modified version of the MVC pattern.

What are Design Patterns ?

- More formal recognition of design patterns began in the early 1990's when Erich Gamma described patterns incorporated in the GUI application framework, ET++.

- Programmers began to meet and discuss these ideas. The culmination of these discussions and a number of technical meetings was the publication of the seminal book, ***Design Patterns – Elements of Reusable Software*, by Gamma, Helm, Johnson, and Vlissides**.

- This book became an all-time bestseller and has had a powerful impact on those seeking to understand how to use design patterns.

  – NOTE: This book is recommended reading for this course !

Defining Design Patterns

- Some useful definitions of design patterns have emerged as the literature in this field has expanded:
  - "Design patterns are recurring solutions to design problems you see over and over." *[Smalltalk Companion]*
  - "Design patterns focus on the reuse of architectural design themes." [*Coplien and Schmidt, 1995*]
  - "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it." *[Buschmann and Meunier, et al., 1996]*
  - "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [Gamma, Helm, Johnson, and Vlissides, 1993]

Defining Design Patterns

- Design patterns can exist at many levels, from **very low-level specific solutions to broadly generalized system issues.**
- Patterns are **discovered NOT developed** !
- The 23 patterns included in *Design Patterns* all had several known applications and were on a middle level of granularity where they could easily cross application areas and encompass several objects.
- The authors divided these patterns into three types:
  - **Creational Patterns**
  - **Structural Patterns**
  - **Behavioral Patterns**

Defining Design Patterns

- **Creational Patterns**
  - **create objects for you**, rather than you having to instantiate objects directly. Your program gains 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 and accounting data.

- **Behavioral Patterns**
  - help you to **define the communication between objects in your system** and how the flow is controlled in a complex program.

Pop Quiz

A Design Pattern is:

1. A UML diagram that solves software problems

2. A recurring solution to a design problem that you may see over and over again.

3. A solution to a GUI Java problem.

Pop Quiz

Design Patterns were:

a) Discovered

b) Created

c) Developed

Pop Quiz

A Creational Pattern:

a) Creates programs for you

b) Allows you to create very innovative GUI front ends

c) Create objects for you.

Pop Quiz

A Structural Pattern:

a) Allows you to structure a UML diagram in a specific way.

b) Helps you compose groups of objects into larger structures

c) Structures a Java program in a specific way.

Pop Quiz

A Behavioral Pattern:

a) Helps you to define the communication between objects in your system.

b) Identifies the methods in each class

c) Displays the behaviors of specific classes

CREATIONAL PATTERNS

THE FACTORY PATTERN

The Factory Pattern

- One type of pattern that we see again and again in OO programs is the **factory pattern**.

- **WHAT IS THE FACTORY PATTERN??**

  - A **simple pattern returns an instance of one of several possible classes depending on the data given to it**.

The Factory Pattern

- Usually all objects that it returns have a **common superclass and common methods**, but each performs a slightly different task and is optimized for different kinds of data (**responsibility is shifted down to the subclass)**

- So lets take a look at how the factory pattern works and how we can implement it in Java.

The Factory Pattern

- So for example lets imagine that we want to store lists of numbers.

- Our lists can be either integer lists or floating point lists
  - 1 2 3 4 5 6 7 8 9
  - 1.1 2.2 3.3 4.4 5.5 6.6 7.7

- We have decided that the best way to store these **number lists** would be to develop **two classes**, one that will look after integers and one that will look after floating point numbers. Lets call our classes *IntList* and *DoubleList*.

- Once our application receives a list of numbers we would like to store the list in the appropriate type of object.

The Factory Pattern

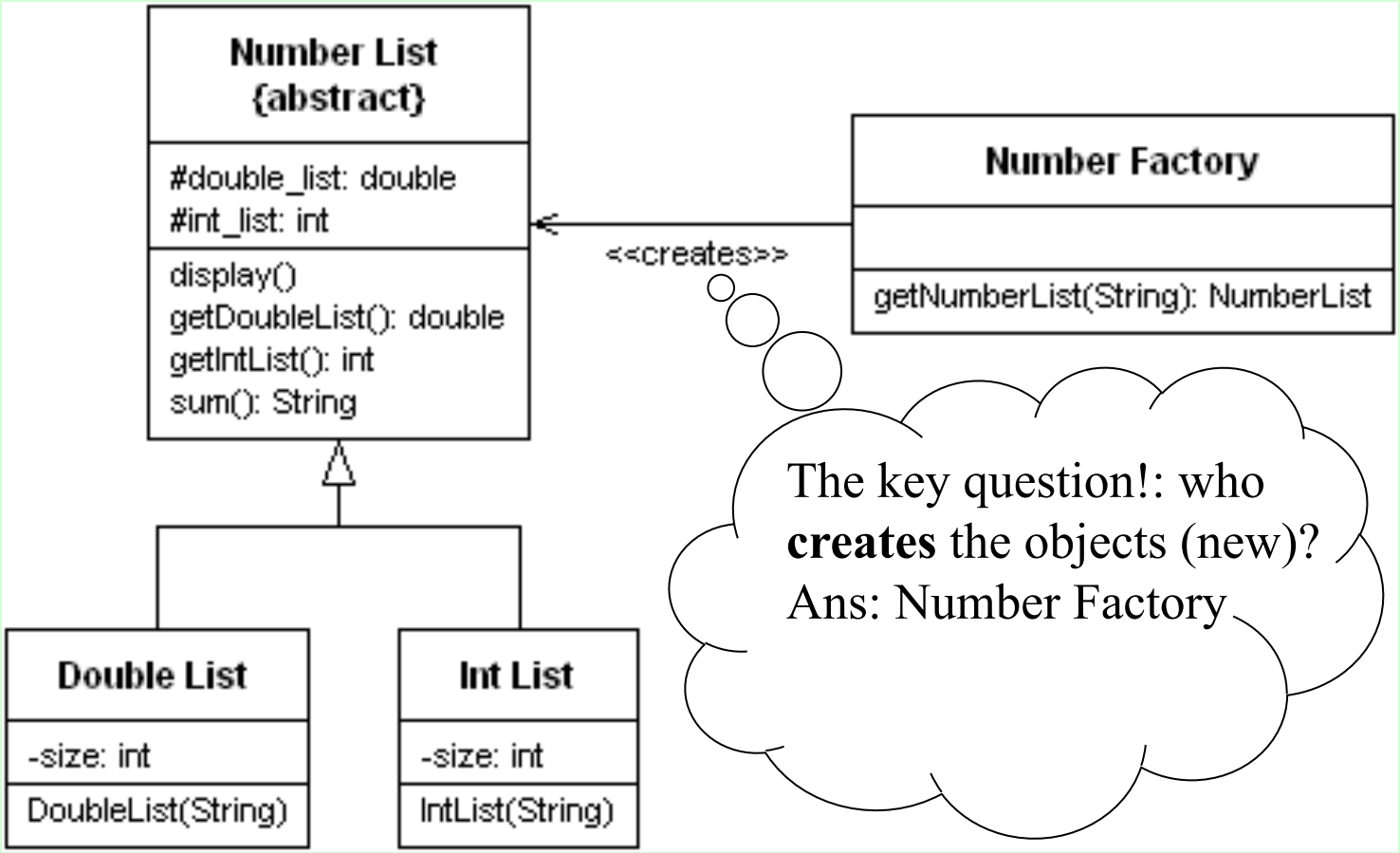
- This means that we will have to do two things:
  - decide which object to create
  - create the object and give it the data

- This may not seem like a big task, but object creation can become a real pain especially if lots of different types of object are required.

- We can use a **simple factory pattern** to solve this problem. The factory pattern takes the burden of **object creation away from the programmer** and makes dealing with the different kinds of number lists quite easy.

The Factory Pattern

- To understand the factory pattern we can use a simple UML diagram.



The Factory Pattern

- From the previous diagram we could see that ***NumberList* is a superclass** and *IntList* and *DoubleList* are derived from it.
- The *NumberFactory* class **decides which of these classes (*IntList*, *DoubleList*) to return** depending on the arguments you give it.
- In order to see how things work let's look at some sample code
- Remember behind each of the design patterns their are **fundamental OOP concepts at work**...so **refresh your knowledge as needed**, or ask questions!!!!
- Techniques like Polymorphism, Override of superclass methods …….

The Factory Pattern

- The first place to start is with the superclass NumberList.
- There are a couple of important points to note about this class.

  – its an **abstract class**
  – we have defined a constructor
  – it has protected attributes
  – it has methods for returning both arrays
  – it has a blank display and sum method

The Factory Pattern

- The most important point is that our two subclasses will **override the display and sum methods** and will have access to the two arrays as they are defined using the **access modifier *protected***.

- Any attribute that's defined as being protected will be directly **available to any class that extends the class it belongs to**.

The Factory Pattern

```
public abstract class NumberList {

  protected int[] intList;
  protected double[] doubleList;

  NumberList() {
    intList = null;
    doubleList = null;
  }

  public int[] getIntList() { return intList; }
  public double[] getDoubleList() { return doubleList; }
  public void display() {}
  public Number sum() { return null; }

}
```

The Factory Pattern

- All this class really does is provide a **common storage place for the arrays** and a **base from which we can extend**.

- Polymorphism plays an important role in almost all commonly used design patterns and recall from last year one of the things needed for polymorphism to work is a base or superclass with methods which can be overridden in a subclass.

- Once we have a **superclass** we can begin to develop the two **subclasses *IntList*** and ***DoubleList*.**

- These will be almost identical apart from the data types they deal with.

The Factory Pattern

```
public class IntList extends NumberList  {
        int size;
        IntList(String list)  {
                size = 0;
                StringTokenizer token = new StringTokenizer(list);
                size = token.countTokens();
                // Allocate some space for the array
                intList = new int[size];
                // Store each list item an the appropriate array
                for(int i = 0; i < size; i++)  {
                        intList[i] = Integer.parseInt(token.nextToken());
                }
        }
```

The Factory Pattern

```
        public Number sum()
        {
                int n = 0;

                for(int i = 0; i < size; i++)
                {
                  n = n + intList[i];
                }
                return new Integer(n);
        }

        public void display()
        {
                System.out.print("Integer List");
                for(int i = 0; i < size; i++)
                        System.out.println("[" + i + "] = " + intList[i]);
        }
}
```

The Factory Pattern

- The DoubleList class is the same except it uses the array of doubles.

```
public class DoubleList extends NumberList {
        int size;
        DoubleList(String list) {
                size = 0;
                StringTokenizer token = new StringTokenizer(list);
                size = token.countTokens();
                // Allocate some space for the array
                doubleList = new double[size];
                // Store each list item an the appropriate array
                for(int i = 0; i < size; i++)  {
                        doubleList[i] = Double.parseDouble(token.nextToken());
                }
        }
```

The Factory Pattern

```
        public Number sum() {
                double n = 0;
                for(int i = 0; i < size; i++) {
                  n = n + doubleList[i];
                }
                return new Double(n);
        }
        public void display() {
                System.out.print("Double List");
                for(int i = 0; i < size; i++)
                        System.out.println("[" + i + "] = " + doubleList[i]);

        }
}
```

The Factory Pattern

- Both the IntList and DoubleList classes could at this stage be used on their own.

- For example:
  - IntList list1 = new IntList(“1 2 3 4 5 6 7 8 9 10”);
  - list1.display();
  - System.out.println(“Sum of Ints = “ + list1.sum());

  - DoubleList list2 = new DoubleList(“1.1 2.2 3.3 4.4 5.5 6.6 7.7”);
  - list2.display();
  - System.out.println(“Sum of Doubles = “ + list2.sum());

- However we still have one more class to write, the factory class which will look after creating objects depend on the data it receives.

The Factory Pattern

- The factory class in the example is actually really simple. All it does is check its argument for the existence of a decimal point.

- If one is found then it returns a new object of type DoubleList, otherwise it returns a new object of type IntList.

```
public class NumberFactory
{
    public NumberList getNumberList(String list)
    {
        // Check for  existence of '.'
        int i = list.indexOf('.');
        if (i != -1)
        {
            return new DoubleList(list);
        }
        else
        {
            return new IntList(list);
        }
    }
}
```

The Factory Pattern

- The factory class has no constructor or attributes, all that is required is a method which will return the appropriate type of object for the given data.

- This means we can **create a factory object** which will **look after creating other objects for use**.

- In order to see how all of this works we can write a test program which will run the simple factory design pattern through its paces.

The Factory Pattern

```
public class NumberFactoryTest
{
    public static void main(String[] args)
    {
        String list1 = new String("1 2 3 4 5 6 7 8 9 10");
        String list2 = new String("1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.1");


        NumberFactory nfactory = new NumberFactory();


        nfactory.getNumberList(list1).display();

        System.out.println("");

        NumberList numberlist2 = nfactory.getNumberList(list2);
        numberlist2.display();

        System.out.println("");
        System.out.println("Sum of list 2 : " + numberlist2.sum());

    }
}
```

nfactory calls the 'new' keyword
Factory creates subclass objects

The Factory Pattern

- Our test program firstly creates two String variables, the first contains a list of integers and the second contains a list of doubles.

- Then we create a new NumberFactory object which will be responsible for creating the appropriate NumberList objects.

- Then we call the getNumberList and pass over the first String variable list1 which contains integers. **The getNumberList method should create and return an IntList object**.

The Factory Pattern

- Then the getNumberList is called for a second time using list2 as an argument. **This time the getNumberList should create and return a DoubleList object.**

- Note: This differs from using polymorphic method calls as we have delegated the instantiation of the type to the factory, with polymorphism the subtype needs to be instantiated

The Factory Pattern

- Both objects can be tested by calling their display and sum methods.

- NOTE : In the first case
  - nfactory.getNumberList(list1).display();
  - Once the display method has been invoked the **returned object will be lost forever** because we have not stored its reference.

- In the second case we use a **reference variable called numberList2** to store the reference returned by getNumberList so that we can call the display and sum methods.

Summary

- We have introduced the concept of Design Patterns
- Given some definitions for Design Patterns
- Described how design patterns can improve the software development process.
- Introduced the simple Factory Pattern
- Implemented a Simple Factory Pattern in Java

Labwork week 2

- See the copy of Labwork 2 in Moodle