

# Object Design & Design Pattern

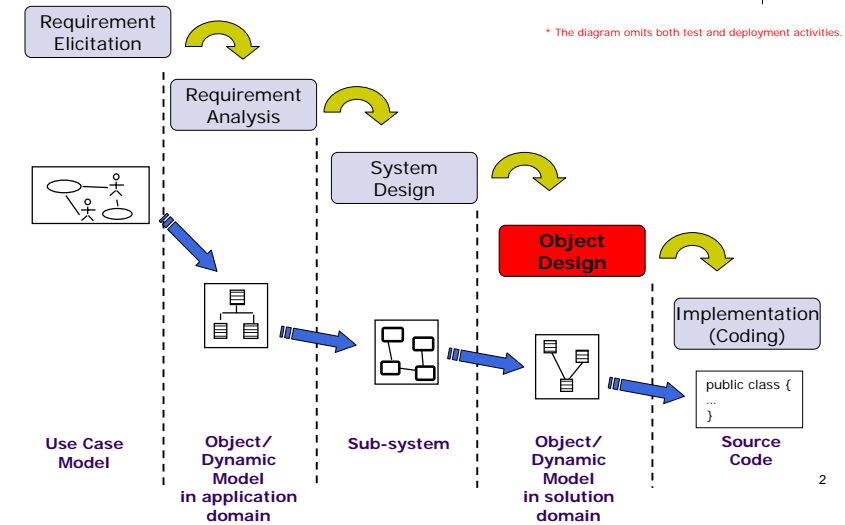
Lecture #09  
Software Engineering and  
Project Management

Instructed by Steven Choy on Nov 27, 2006



1

## Software Development Activities



2

## Object Design Overview



3

## Deliverables of System Analysis

1. Introduction
  - 1.1 Purpose of the system
  - 1.2 Design Goals
  - 1.3 Definition, acronyms and abbreviations
  - 1.4 References
  - 1.5 Overview
2. Software Architecture
  - 2.1 Overview
  - 2.2 Subsystem Decomposition
  - 2.3 Hardware/software mapping
  - 2.4 Persistent data management
  - 2.5 Access control and security
  - 2.6 Global Software Control
  - 2.7 Exception Handling Strategies
  - 2.8 Boundary Conditions
3. Subsystem
4. Glossary

Software Design Document



4

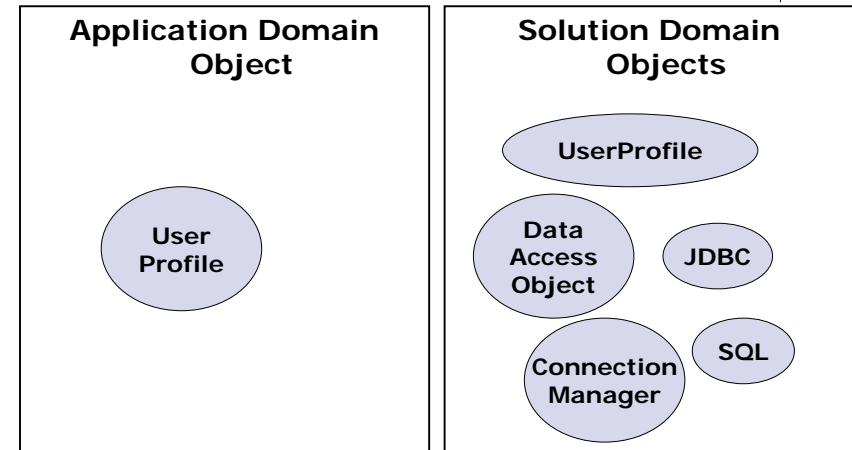
## Application Domain Object vs Solution Domain Object



- Application Domain Objects
  - Represent concepts of the domain that are relevant to the system.
  - They are identified by the application domain specialists and by the end users.
- Solution Domain Objects
  - Represent concepts that do not have a counterpart in the application domain
  - They are identified by the developers

5

## From Domain Objects to Solution Objects



6

## Object Design Activities



- Reuse *(Covered in this lecture)*
  - Use of Inheritance and Composition/Delegation
  - Identify Off-the-shelf framework
  - Design Patterns
- Interface Specification *(Covered later)*
  - Describe interface of classes precisely
- Restructuring *(Covered later)*
  - Transform the object model to increase code reuse or meet other design goals such as readability and maintainability
- Optimization *(Covered later)*
  - Address performance goals such as response time and throughput

7

## What will we discuss today?



- Reuse Concepts
- Use of Inheritance
- Inheritance vs Composition
- Design Patterns *(next lecture)*
  - Creational Pattern
  - Structural Pattern
  - Behavioral Pattern

8

# Reuse Concepts



# Why Reuse?



- Business requirements are always changing. The system you design for today may not be the same for tomorrow.
- You can't say "No, my system is not designed for that business requirement."
- You don't want to rewrite your system when there are some new requirements raise up.
- → The need for reusable and flexible design.

# Reuse of Classes

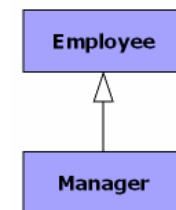


- Two fundamental ways of reuse:
  - The use of inheritance
  - The use of composition

# Inheritance Overview



- Inheritance is one of merits of OO Design
- Extend from a Base Class
  - Add in new operations
  - Overwrite the existing operations
  - Reuse operations in the base class
- Promote code reuse, reduce redundancy and make your class extensible



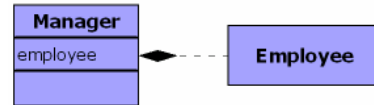
```
class Employee {
// ...
}
class Manager extends Employee {
// ...
}
```

Java Code Sample

# Composition Overview



- An alternative to inheritance for code reuse
- Implementation view: Class A holds an instance of Class B . Class A can call on methods of Class B.
- Example: Manager class holds a reference to Employee class using instance variable



```
class Employee {
// ...
}
class Manager {
// ...
private Employee employee = new Employee();
}
```

Java Code Sample

# Inheritance Problem



- The Ripple Effect: A change in Base Class may lead to change of other subclasses
- Example: Modify an interface in base class may result compilation failure of the subclasses

```
Employee Class:
class Employee {
public double getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
// ...
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

What if the return value of getSalary() is changed?

```
Employee Class:
class Employee {
public SalaryInfo getSalary() {
// ...
}
}
SalaryInfo Class:
class SalaryInfo {
private double salary;
public double getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
// ...
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

# Inheritance Problem Example



- What if the return value of getSalary() method is change?

```
Employee Class:
class Employee {
public double getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
// ...
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

Change interface

```
Employee Class:
class Employee {
public SalaryInfo getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
// ...
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

Compilation error

# The Composition Alternative



- For the inheritance problem, what if composition is used for code reuse?

```
Employee Class:
class Employee {
public double getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
private Employee employee = new Employee();
public double getSalary() {
return employee.getSalary();
}
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

Change interface

```
Employee Class:
class Employee {
public SalaryInfo getSalary() {
// ...
}
}
Manager Class:
class Manager extends Employee {
private Employee employee = new Employee();
public double getSalary() {
SalaryInfo salaryInfo = employee.getSalary();
return salaryInfo.getSalary();
}
}
Sample Program:
class Sample {
public static void main(String[] args) {
Manager manager = new Manager();
double salary = manager.getSalary();
}
}
```

The only fix

Still can compile

## Code Reuse – Inheritance or Composition?



- When using inheritance, make sure the base class and subclass has a **is-a** relationship.
- Generally, favor composition over inheritance for code reuse.
- Composition helps you keep each class encapsulated.

17

## Design Pattern



18

## Pattern in Composition Exam



- Date back to my HKALE, I was once told about how I can get a pass for argumentative writing:

1. For the 1<sup>st</sup> paragraph, rephrase the question.
2. For the 2<sup>nd</sup> paragraph and onwards, list out the points to support your argument and elaborate one point in each paragraph.
3. For the last paragraph, make your conclusion and re-emphasize your proposition.
4. For all paragraphs, avoid grammatical mistakes. Just use simple sentence that you are sure it's grammatically correct.

That's the "pattern" to win a pass!

19

## Back to the History



- Pattern is not firstly adopted by software experts
- In 1970s, Christopher Alexander wrote a number of books about patterns in architecture and civil engineering
- Software engineers inspired by the pattern principle and applied it in software design
- Pattern in software were popularized by the book **"Design Patterns: Elements of Reusable Object-Oriented Software"**, written by Gang of Four or GoF

20



*Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.*

- Christopher Alexander

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

- Christopher Alexander

21

## What's Design Pattern?



- Design patterns are recurring solutions to design problems you see over and over again [Alpert, et al., 1998]
- Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development [Pree, 1994]
- Design patterns are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]

22

## Pattern Characteristics



- Patterns are not created but observed through experience
- Patterns avoid you to re-invent the wheel
- Patterns are reusable artifacts
- Patterns undergo continuous improvement
- Patterns can be used together to solve larger problem

23

## Benefits of Using Patterns



- Reuse proven solutions
  - You do not need to reinvent your wheel
  - The best solution is already there
- Establish common vocabulary
  - Pattern provides developers a common point of reference to communicate design
- Have a higher perspective on analysis and design
  - Free you from dealing with details too early
- Improve flexibility of your code

24

## "Gang of Four" Design Patterns



- The GoF revealed a total of 23 design patterns and categorized them
- Creational Patterns
  - Patterns that are for object creations
  - But rather than just instantiate objects directly, they defines the best ways to create objects
- Structural Patterns
  - Patterns that helps you compose objects into larger structures
- Behavioral Patterns
  - Patterns that concerns with interactions between objects
  - Help developers define the communication between objects in the system

25

## Pattern Categories



### Creational Pattern

<b>Abstract Factory</b>	Creates an instance of several families of classe
<b>Builder</b>	Separates object construction from its representation
<b>Factory Method</b>	Creates an instance of several derived classes
<b>Prototype</b>	A fully initialized instance to be copied or cloned
<b>Singleton</b>	A class of which only a single instance can exist

26

## Pattern Categories



### Structural Pattern

<b>Adapter</b>	Match interfaces of different classes
<b>Bridge</b>	Separates an object's interface from its implementation
<b>Composite</b>	A tree structure of simple and composite objects
<b>Decorator</b>	Add responsibilities to objects dynamically
<b>Façade</b>	A single class that represents an entire subsystem
<b>Flyweight</b>	A fine-grained instance used for efficient sharing
<b>Proxy</b>	An object representing another object

27

## Pattern Categories



### Behavioral Pattern

<b>Chain of Responsibility</b>	A way of passing a request between a chain of objects
<b>Command</b>	Encapsulate a command request as an object
<b>Interpreter</b>	A way to include language elements in a program
<b>Iterator</b>	Sequentially access the elements of a collection
<b>Mediator</b>	Defines simplified communication between classes
<b>Memento</b>	Captures and restores an object's internal state
<b>Observer</b>	A way of notifying change to a number of classes
<b>State</b>	Alter an object's behaviour when its state changes
<b>Strategy</b>	Encapsulates an algorithm inside a class
<b>Template Method</b>	Defers the exact steps of an algorithm to a subclass
<b>Visitor</b>	Defines a new operation to a class without change

28

## Documenting Design Pattern



- A design pattern has four elements:
  - A name
    - Name of the pattern
  - A problem description
    - Describe the problem/situation faced by developers in which the pattern can be used
  - A solution
    - Describe the solution approach and the solution elements in detail
  - A set of consequences
    - Describe the trade-offs of pattern, that the pros and cons of using the pattern

29

## Strategies Behind Design Patterns



- The GoF suggests the following strategies for creating good OO Design:
  - Design to interface
  - Favour composition over inheritance
  - Find what varies and encapsulate it

30

## Design Pattern: Singleton



31

How do you ensure only a single instance of class is instantiated?

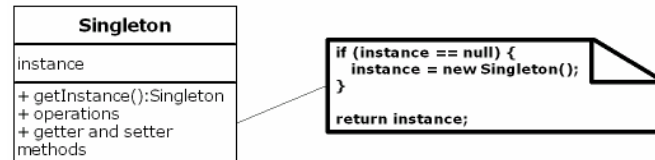


32



# Singleton Pattern

- What did GoF say?
  - Ensure a single instance of a class
  - Provide a global access point to the instance
- How does it work?



33



# Singleton Implementation

```
class Singleton {
    private static Singleton instance = null;
    private Singleton () {
    }

    public Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton ();
        }
        return instance;
    }

    // Other method implementations
}
```

34



# Singleton Example

- You can find plenty of singleton usages in JDK
  - java.util.Calendar
  - java.util.Currency
  - java.text.DateFormat
  - java.security.KeyStore
  - java.security.KeyFactory

35

# Design Pattern: Observer



36



What if a group of objects needs to update themselves when some object changes state?

37



## The Observer Pattern

- This pattern solves a common problem:
  - What if a group of objects needs to update themselves when some object changes state?
- aka Dependents & Publish-Subscribe
  - An object notifies other objects if it changes

38

## Implement in Java

- There are two types of objects used to implement the observer pattern in Java
  - Interface Observer
  - Class Observable

39



## Implement in Java...

- The Observable object keeps track of everybody who wants to be informed when a change happens.
- When someone says "OK, everybody should check and potentially update yourselves,"
  - The Observable object performs the `notifyObservers()` method for each one on the list.
  - The `notifyObservers()` method is part of the base class `Observable`.

40



## Implement in Java...

- An Observer object is some object that implements the Observer interface
- The Observer object will register with an Observable object.
- When a change has been made in the Observable object, the Observable object will call the Observer object's update( ) method.

41



```
public class Person extends Observable {

    private String forename ;
    private String surname ;
    private int age ;

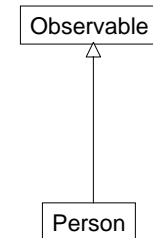
    .....

    public void setForename(String f) {
        forename = f ;
        setChanged();
        notifyObservers();
    }

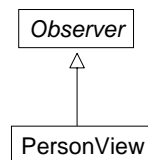
    public void setSurname(String s) {
        .....
    }

    public void setAge(int a) {
        .....
    }

}
```



42



```
public class PersonView extends JFrame
implements ActionListener, Observer {

    ....

    public void update( Observable o, Object arg ) {
        Person aperson = ( Person ) o;
        ....
        al.setText( "Age: " + String.valueOf( aperson.getAge() ) );
    }

}
```

43



```
public class ControllerTest {
    public static void main( String[] args ) {

        Person p = new Person( "Ivor", "Pattern", 23 );
        PersonView pfr = new PersonView( "Person GUI" );
        p.addObserver( pfr );

        pfr.setLocation( 200, 200 );
        pfr.setVisible( true );
        try {
            for( int i = 0; i <= 300; i++ ) {
                p.increaseAge( 1 );
                Thread.sleep( 1000 );
            }
        }
        catch( Exception e ) {
            System.out.println( e );
        }
    }
}
```

44



## References



- GoF Patterns  
<http://www.fluffycat.com/java/patterns.html>
- J2EE Patterns  
<http://java.sun.com/blueprints/patterns/>

## Acknowledgement

Part of the slides  
were originally authored

by **Simon Ng**.

