Software Engineering – Fall 2015
(CSC 4350/6350)
TR. 5:30 pm – 7:15 pm

Rao Casturi
11/03/2015
http://cs.gsu.edu/~ncasturi1
Object Design
Object Design

• Close the gap between the application objects and the off-the-shelf components.
• Identify additional solution objects.
• Refine the existing objects.
Object Design includes
(4 Groups of Activities)

1. Reuse (Component Selection)
2. Service Specification
3. Object Model Restructuring
4. Object Model Optimization
Reuse or off-the-shelf components

- Off-the-Shelf components are identified during system design.
- Class Libraries and additional components are selected for basic data structures and services.
- Design patterns are selected for solving common problems.
- Classes are protected from future changes.
- Buy-versus-Build trade off
Service Specification

• The subsystem services identified during the system design are specified in terms of:
  [Class Interfaces, Operations, arguments, type signatures and exceptions].

• Additional operations and objects needed for data transfer among subsystems are identified.

• The subsystem service specification is also called API (Application Programmer Interface)
Object Model Restructuring

- In this phase the focus is more on the reuse of the code.
- Revisit the N-ary Association of classes and see if we can get it to Binary association
- Look for Merging 2 similar classes, 2 similar subsystems.
- Attribute merging.
- Complex class decomposition for simple inheritance.
- Maintainability, Readability, Understandability
Object Model Optimization

• This phase addresses the performance requirements of the system model.
• Efficiency of algorithms, Memory management is a focus.
• Techniques like simplicity vs multiplicity
• Rearrange Execution order for efficiency
• Adding redundancy / derived attributes for speedup.
Christopher Alexander says, “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 same way ever twice”.

*Objects and interfaces instead of Walls and doors*
Reuse Concepts:
Solution Objects, Inheritance, and Design Patterns

• Application Objects and Solution Objects
• Specification Inheritance and Implementation Inheritance
• Delegation
• The Liskov Substitution Principle
• Delegation and Inheritance in Design Patterns
Application Objects & Solution Objects

• During analysis, we identify entity objects and their relationships, attributes, and operations. Most entity objects are application objects that are independent of any specific system.

• During analysis, we also identify solution objects that are visible to the user, such as boundary and control objects representing forms and transactions defined by the system.

• During system design, we identify more solution objects in terms of software and hardware platforms.

• During object design, we refine and detail both application and solution objects and identify additional solution objects needed to bridge the object design gap.
Specification Inheritance and Implementation Inheritance

• Superclass, Subclass
• The focus of inheritance during object design is to reduce redundancy and enhance extensibility.
• By factoring all redundant behavior into a single superclass, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we have to make changes only once for all subclasses
• By Providing abstract classes interfaces that are used by the application, we can write new specialized behavior by writing new subclasses that comply with the abstract interfaces.
In object-oriented analysis and design, inheritance is used for achieving several goals, in particular modeling taxonomies and reusing behavior from abstract classes.

When modeling taxonomies, the inheritance relationships can be identified either during specializations (when specialized classes are identified after general ones) or during generalizations (when general classes are abstracted out of a number of specialized ones).

When using inheritance for reuse, specification inheritance represents subtyping relationships, and implementation inheritance represents reuse among conceptually unrelated classes.

Source: Object-Oriented Software Engineering – Using UML, Patterns, and Java – Brenda Bruegge and Allen Dutoit
• The use of inheritance for the sole purpose of reusing code is called **implementation inheritance**.
• With implementation inheritance, developers reuse code quickly by subclassing an existing class and refining its behavior.
• A Set implemented by inheriting from a **Hashtable** is an example of **implementation inheritance**.
• Conversely, the classification of concepts into type hierarchies is called **specification inheritance** (also called “interface inheritance”)

---

Inheritance Example

```
/* Implementation of MySet using inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)) {
            put(element, this);
        }
    }

    boolean containsValue(Object element) {
        return containsKey(element);
    }
    /* Other methods omitted */
}

/* Implementation of MySet using delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }

    void put(Object element) {
        if (!containsValue(element)) {
            table.put(element, this);
        }
    }

    boolean containsValue(Object element) {
        return (table.containsKey(element));
    }
    /* Other methods omitted */
}
```
Delegation

• **Delegation** is the alternative to implementation inheritance that should be used when reuse is desired.

• A class is said to delegate to another class *if it implements an operation by resending a message to another class.*

• Delegation makes explicit the dependencies between the reused class and the new class.
The only significant change is the private field table and its initialization in the `MySet()` constructor.

This addresses 2 problems:

- **Extensibility.** The MySet on the right column does not include the `containsKey()` method in its interface and the new field table is private. Hence, we can change the internal representation of MySet to another class (e.g., a List) without impacting any clients of MySet.

- **Subtyping.** MySet does not inherit from Hashtable and, hence, cannot be substituted for a Hashtable in any of the client code. Consequently, any code previously using Hashtables still behaves the same way.

- Delegation is a preferable mechanism to implementation inheritance as it does not interfere with existing components and leads to more robust code. Note that specification inheritance is preferable to delegation in subtyping situations as it leads to a more extensible design.
The Liskov Substitution Principle

• If a client code uses the methods provided by a superclass, then developers should be able to add new subclasses without having to change the client code.
The Liskov Substitution Principle

- Liskov Substitution Principle
  - If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

- Interpretation
  - In object design, the Liskov Substitution Principle means that if all classes are subtypes of their superclasses, all inheritance relationships are specification inheritance relationships. In other words, a method written in terms of a superclass T must be able to use instances of any subclass of T without knowing whether the instances are of a subclass. Consequently, new subclasses of T can be added without modifying the methods of T, hence leading to an extensible system. An inheritance relationship that complies with the Liskov Substitution Principle is called strict inheritance.
Design pattern – 4 Essential Elements

1. A *name* that uniquely identifies the pattern from other patterns.

2. A *problem description* that describes the situations in which the pattern can be used. Problems addressed by design patterns are usually the realization of modifiability and extensibility design goals and nonfunctional requirements.

3. A *solution* stated as a set of collaborating classes and interfaces.

4. A set of *consequences* that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.
### Cataloging of the Design Patterns

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td>Adapter (Class)</td>
<td>Interpreter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template Method</td>
<td></td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Chain of Responsibility</td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td>Composition</td>
<td>Iterator</td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Façade</td>
<td>Memento</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Observer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>State</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor</td>
<td></td>
</tr>
</tbody>
</table>
A.2 Adapter: Wrapping Around Legacy Code

<table>
<thead>
<tr>
<th>Name</th>
<th>Adapter Design Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem description</strong></td>
<td>Convert the interface of a legacy class into a different interface expected by the client, so that the client and the legacy class can work together without changes.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td>An Adapter class implements the ClientInterface expected by the client. The Adapter delegates requests from the client to the LegacyClass and performs any necessary conversion.</td>
</tr>
</tbody>
</table>

```
\begin{tikzpicture}
\node[anchor=east] (t) at (0,0) {Client};
\node[rectangle, below=of t] (r) {ClientInterface};
\node[below=of r] (cl) {Request()};
\node[below=of cl] (l) {ExistingRequest()};
\node[below=of l] (o) {adaptee};
\node[below=of o] (n) {Adapter};
\node[below=of n] (m) {Request()};
\node[below=of m] (c) {LegacyClass};
\path[->] (t) -- (r);
\path[->] (r) -- (cl);
\path[->] (cl) -- (m);
\end{tikzpicture}
```

**Consequences**
- Client and LegacyClass work together without modification of either Client or LegacyClass.
- Adapter works with LegacyClass and all of its subclasses.
- A new Adapter needs to be written for each specialization (e.g., subclass) of ClientInterface.

**Example**
- Sorting instances of an existing String class with an existing sort() method (Section 8.4.2): MyStringComparator is an Adaptor for bridging the gap between the String class and the Comparator interface used by the Array.sort() method.

**Related concept**
- The Bridge (Section A.3) fills the gap between an interface and its implementations.
The **client class** accesses the pattern. In the class diagram of the Adapter pattern this class is simply called Client. Client classes can be either existing classes of a class library or new classes of the system under development. The **pattern interface** is the part of the pattern that is visible to the client class. Often, the pattern interface is realized by an abstract class or an interface. In the Adapter pattern, this class is called ClientInterface. The **implementor class** provides the lower-level behavior of the pattern. In the Adapter pattern, the LegacyClass and the Adapter are implementor classes. In many patterns, a number of collaborating implementor classes are needed to realize the pattern behavior. The **extender class** specializes an implementor class to provide a different implementation or an extended behavior of the pattern. In the Adapter pattern, the subtypes of LegacyClass are extender classes. Note that, often, extender classes represent future classes that developers anticipate.
# Identification of Design Patterns

## Natural language heuristics for selecting design patterns

Design patterns address specific design goals and nonfunctional requirements. Similar to Abbott’s heuristics described in Chapter 5, *Analysis*, key phrases can be used to identify candidate design patterns. Below are examples for the patterns covered in this chapter.

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Design Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Manufacturer independence&quot;</td>
<td>Abstract Factory</td>
</tr>
<tr>
<td>&quot;Platform independence&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Must comply with existing interface&quot;</td>
<td>Adapter</td>
</tr>
<tr>
<td>&quot;Must reuse existing legacy component&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Must support future protocols&quot;</td>
<td>Bridge</td>
</tr>
<tr>
<td>&quot;All commands should be undoable&quot;</td>
<td>Command</td>
</tr>
<tr>
<td>&quot;All transactions should be logged&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Must support aggregate structures&quot;</td>
<td>Composite</td>
</tr>
<tr>
<td>&quot;Must allow for hierarchies of variable depth and width&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Policy and mechanisms should be decoupled.&quot;</td>
<td>Strategy</td>
</tr>
<tr>
<td>&quot;Must allow different algorithms to be interchanged at runtime.&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Object Design
Interface Specification Concepts
Activities

• **Identify missing attributes and operations**
  - During this activity, we examine each subsystem service and each analysis object. We identify missing operations and attributes that are needed to realize the subsystem service. We refine the current object design model and augment it with these operations.

• **Specify visibility and signatures**
  - During this activity, we decide which operations are available to other objects and subsystems, and which are used only within a subsystem. We also specify the **turn type of each operation as well as the number and type of its parameters**. This goal of this activity is to reduce coupling among subsystems and provide a mall and simple interface that can be understood easily by a single developer.

• **Specify contracts**
  - During this activity, we describe in terms of constraints the behavior of the operations provided by each object. In particular, for each operation, we describe the conditions that must be met before the operation is invoked and a specification of the result after the operation returns.
Class Implementor, Class Extender, and Class User

- The **class implementor** is responsible for realizing the class under consideration. Class implementors design the internal data structures and implement the code for each public operation. For them, the interface specification is a work assignment.

- The **class user** invokes the operations provided by the class under consideration during the realization of another class, called the **client class**. For class users, the interface specification discloses the boundary of the class in terms of the services it provides and the assumptions it makes about the client class.

- The **class extender** develops specializations of the class under consideration. Like class implementors, class extenders may invoke operations provided by the class of interest, the class extenders focus on specialized versions of the same services. For them, the interface specification both a specifies the current behavior of the class and any constraints on the services provided by the specialized class.
During analysis, we identified attributes and operations without necessarily specifying their types or their parameters.

The **Type** of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute.

The Type constrains the range of values the parameter or the return value can take. Given an operation, the tuple made out of the types of its parameters and the type of the return value is called the **signature** of the operation.

The **Visibility** of an attribute or an operation is a mechanism for specifying whether the attribute or operation can be used by other classes or not.
A *private attribute* can be accessed only by the class in which it is defined. Similarly, a *private operation* can be invoked only by the class in which it is defined. Private attributes and operations cannot be accessed by subclasses or calling classes. Private operations and attributes are intended for the class implementor only.

A *protected* attribute or operation can be accessed by the class in which it is defined and by any descendant of that class. Protected operations and attributes cannot be accessed by any other class. Protected operations and attributes are intended for the class extender.

A *public* attribute or operation can be accessed by any class. The set of public operations and attributes constitute the public interface of the class and is intended for the class user.

An attribute or an operation with *visibility package* can be accessed by any class in the nearest enclosing package. This visibility enables a set of related classes (for example, forming a subsystem) to share a set of
Questions?
References

• Use Cases Combined with BOOCH, OMT UML Process and Products
  - Putnam P Texel, Charles B Williams

• Object-Oriented Software Engineering Using UML Patterns, and JAVA
  - Bernd Bruegge & Allen H. Dutoit

• Software Engineering 9th Edition
  - Ian Sommerville