Chapter 8, Object Design: Reuse and Patterns

Object Design

- **Purpose of object design:**
  - Prepare for the implementation of the system model based on design decisions
  - Transform the system model (optimize it)
  - Investigate alternative ways to implement the system model
  - Use design goals: minimize execution time, memory and other measures of cost.
  - Object design serves as the basis of implementation.

Terminology: Naming of Design Activities

Methodology: Object-oriented software engineering (OOSE)
- **System Design**
  - Decomposition into subsystems, etc

Methodology: Structured analysis/structured design (SA/SD)
- **Preliminary Design**
  - Decomposition into subsystems, etc
  - Data structures are chosen
- **Detailed Design**
  - Algorithms are chosen
  - Data structures are refined
  - Implementation language is chosen.

Object Design consists of 4 Activities

1. **Reuse: Identification of existing solutions**
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - Design patterns

2. **Interface specification**
   - Describes precisely each class interface

3. **Object model restructuring**
   - Transforms the object design model to improve its understandability and extensibility

4. **Object model optimization**
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Object Design Activities

- **Lecture 17**
  - Select Subsystem

- **Lecture 18**
  - Check Use Cases

Detailed View of Object Design Activities (ctd)

- **Lecture 18**
  - Check Use Cases

- **Restructuring**
  - Revisiting Inheritance
  - Collapsing Classes
  - Realizing associations

- **Optimization**
  - Optimizing access paths
  - Optimizing complex computations
One Way to do Object Design

1. Identify the missing components in the design gap
2. Make a build or buy decision to obtain the missing component

=> Component-Based Software Engineering:
   The design gap is filled with available components ("0 % coding").

- Special Case: COTS-Development
  - COTS: Commercial-off-the-Shelf
  - The design gap is completely filled with commercial-off-the-shelf-components.

=> Design with standard components.

Adapter Pattern

- **Adapter Pattern**: Connects incompatible components.
  - It converts the interface of one component into another interface expected by the other (calling) component
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering)
  - Also known as a wrapper.

Adapter Pattern

![Diagram of Adapter Pattern]

Modeling of the Real World

- Modeling of the real world leads to a system that reflects today’s realities but not necessarily tomorrow’s.
- There is a need for reusable and flexible designs
- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge.

Reuse of Code

- I have a list, but my customer would like to have a stack
  - The list offers the operations Insert(), Find(), Delete()
  - The stack needs the operations Push(), Pop() and Top()
  - Can I reuse the existing list?
- I am an employee in a company that builds cars with expensive car stereo systems
  - Can I reuse the existing car software in a home stereo system?

Reuse of interfaces

- I am an off-shore programmer in Hawaii. I have a contract to implement an electronic parts catalog for DaimlerChrysler
  - How can I and my contractor be sure that I implement it correctly?
- I would like to develop a window system for Linux that behaves the same way as in Vista
  - How can I make sure that I follow the conventions for Vista windows and not those of MacOS X?
- I have to develop a new service for cars, that automatically call a help center when the car is used the wrong way.
  - Can I reuse the help desk software that I developed for a company in the telecommunication industry?
**Reuse of existing classes**

- I have an implementation for a list of elements of type int
  - Can I reuse this list to build
    - a list of customers
    - a spare parts catalog
    - a flight reservation schedule?
- I have developed a class “Addressbook” in another project
  - Can I add it as a subsystem to my e-mail program which I purchased from a vendor (replacing the vendor-supplied addressbook)?
  - Can I reuse this class in the billing software of my dealer management system?

**Customization: Build Custom Objects**

- **Problem:** Close the object design gap
  - Develop new functionality
- **Main goal:**
  - Reuse knowledge from previous experience
  - Reuse functionality already available
- **Composition (also called Black Box Reuse)**
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing objects
- **Inheritance (also called White-box Reuse)**
  - New functionality is obtained by inheritance

**White Box and Black Box Reuse**

- **White box reuse**
  - Access to the development products (models, system design, object design, source code) must be available

- **Black box reuse**
  - Access to models and designs is not available, or models do not exist
    - Worst case: Only executables (binary code) are available
    - Better case: A specification of the system interface is available.

**Identification of new Objects during Object Design**

**Requirements Analysis (Language of Application Domain)**

**Object Design (Language of Solution Domain)**

<table>
<thead>
<tr>
<th>Requirements Analysis (Language of Application Domain)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
</tr>
<tr>
<td>subscribe(subscriber)</td>
</tr>
<tr>
<td>update()</td>
</tr>
<tr>
<td>ConcreteSubject</td>
</tr>
<tr>
<td>getState()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>update()</td>
</tr>
<tr>
<td>ConcreteObserver</td>
</tr>
<tr>
<td>update()</td>
</tr>
</tbody>
</table>

**Application Domain vs Solution Domain Objects**

**Object Design (Language of Solution Domain)**

- The implementation of algorithms may necessitate objects to hold values
- New low-level operations may be needed during the decomposition of high-level operations
  - Example: `EraseArea()` in a drawing program
    - Conceptually very simple,
    - Implementation is complicated:
      - Area represented by pixels
      - We need a `Repair()` operation to clean up objects partially covered by the erased area
      - We need a `Redraw()` operation to draw objects uncovered by the erasure
      - We need a `Draw()` operation to erase pixels in background color not covered by other objects.
Types of Whitebox Reuse

1. Implementation Inheritance
   - Reuse of Implementations
2. Specification Inheritance
   - Reuse of Interfaces

- Programming concepts to achieve reuse
  - Inheritance
  - Delegation
  - Abstract classes and Method Overriding
  - Interfaces

Why Inheritance?

1. Organization (during analysis):
   - Inheritance helps us with the construction of taxonomies to deal with the application domain
   - when talking with the customer and application domain experts we usually find already existing taxonomies

2. Reuse (during object design):
   - Inheritance helps us to reuse models and code to deal with the solution domain
   - when talking to developers

The use of Inheritance

- Description of Taxonomies
  - Used during requirements analysis
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable

- Interface Specification
  - Used during object design
  - Activity: identify the signatures of all identified objects
  - Goal: increase reusability, enhance modifiability and extensibility

Inheritance can be used during Modeling as well as during Implementation

- Starting Point is always the requirements analysis phase:
  - We start with use cases
  - We identify existing objects ("class identification")
  - We investigate the relationship between these objects; "Identification of associations":
    - general associations
    - aggregations
    - inheritance associations.

Inheritance comes in many Flavors

Inheritance is used in four ways:

- Specialization
- Generalization
- Specification Inheritance
- Implementation Inheritance.

Discovering Inheritance

- To "discover" inheritance associations, we can proceed in two ways, which we call specialization and generalization

  - Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
  - Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.
Generalization

- First we find the subclass, then the superclass
- This type of discovery occurs often in science and engineering:
  - Biology: First we find individual animals (Elephant, Lion, Tiger), then we discover that these animals have common properties (mammals).
  - Engineering: What are the common properties of cars and airplanes?

Restructuring of Attributes and Operations is often a Consequence of Generalization

Specialization

- Specialization occurs, when we find a subclass that is very similar to an existing class.
  - Example: A theory postulates certain particles and events which we have to find.
  - Specialization can also occur unintentionally.

Example of a Specialization

CandyMachine is a new product and designed as a subclass of the superclass VendingMachine

A change of names might now be useful: dispenseItem() instead of dispenseBeverage() and dispenseSnack()
**Meta-Model for Inheritance**

![Diagram showing the meta-model for inheritance]

---

**Implementation Inheritance and Specification Inheritance**

- **Implementation inheritance**
  - Also called class inheritance
  - Goal:
    - Extend an application's functionality by reusing functionality from the superclass
    - Inherit from an existing class with some or all operations already implemented

- **Specification Inheritance**
  - Also called subtyping
  - Goal:
    - Inherit from a specification
    - The specification is an abstract class with all operations specified, but not yet implemented.

---

**Implementation Inheritance vs. Specification Inheritance**

- **Implementation Inheritance:** The combination of inheritance and implementation
  - The interface of the superclass is completely inherited
  - Implementations of methods in the superclass ("reference implementations") are inherited by any subclass

- **Specification Inheritance:** The combination of inheritance and specification
  - The interface of the superclass is completely inherited
  - Implementations of the superclass (if there are any) are not inherited.

---

**Delegation instead of Implementation Inheritance**

- **Inheritance:** Extending a Base class by a new operation or overwriting an operation.
- **Delegation:** Catching an operation and sending it to another object.
- **Which of the following models is better?**

![Diagram comparing inheritance and delegation]

---

**Delegation**

- Delegation is a way of making composition as powerful for reuse as inheritance.
- In delegation two objects are involved in handling a request from a Client.
  - The Receiver object delegates operations to the Delegate object.
  - The Receiver object makes sure, that the Client does not misuse the Delegate object.

![Diagram illustrating delegation]

---

**Example for Implementation Inheritance**

- A very similar class is already implemented that does almost the same as the desired class implementation.

  ![Diagram illustrating the example]

---

**Problem with implementation inheritance:**

- The inherited operations might exhibit unwanted behavior.
- Example: What happens if the Stack user calls `Remove()` instead of `Pop()`?
Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
  - Flexibility: Any object can be replaced at run time by another one
  - Inefficiency: Objects are encapsulated
- Inheritance
  - Straightforward to use
  - Supported by many programming languages
  - Easy to implement new functionality
  - Exposes a subclass to details of its super class
  - Change in the parent class requires recompilation of the subclass.

Recall: Implementation Inheritance v. Specification Inheritance

- **Implementation Inheritance**: The combination of inheritance and implementation
  - The Interface of the super class is completely inherited
  - Implementations of methods in the super class ("Reference implementations") are inherited by any subclass
- **Specification Inheritance**: The combination of inheritance and specification
  - The super class is an abstract class
  - Implementations of the super class (if there are any) are not inherited
  - The Interface of the super class is completely inherited

Abstract Methods and Abstract Classes

- **Abstract method**:
  - A method with a signature but without an implementation (also called abstract operation)
- **Abstract class**:
  - A class which contains at least one abstract method is called abstract class
- **Interface**: An abstract class which has only abstract methods
  - An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

Example of an Abstract Method

```
VendingMachine
totalReceipts
collectMoney()
makeChange()
dispenseItem()
```

```
CoffeeMachine
numberOfCups
coffeeMix
heatWater()
addSugar()
addCreamer()
dispenseItem()
```

```
SodaMachine
cansOfBeer
cansOfCola
chill()
dispenseItem()
```

```
CandyMachine
bagsOfChips
numberOfCandyBars
dispenseItem()
```

dispenseItem() must be implemented in each subclass. We do this by specifying the operation as abstract. Abstract operations are written in UML in italics.

Rewritable Methods and Strict Inheritance

- **Rewritable Method**: A method which allows a reimplementation.
  - In Java methods are rewriteable by default, i.e. there is no special keyword.
- **Strict inheritance**
  - The subclass can only add new methods to the superclass, it cannot over write them
  - If a method cannot be overwritten in a Java program, it must be prefixed with the keyword `final`.

```
public class LuxuryCar extends Car {
    public final void drive() {...}
    public final void brake() {...}
    public final void accelerate() {...}
}
```

Strict Inheritance

Superclass:
```
public class Car {
    public final void drive() {...}
    public final void brake() {...}
    public final void accelerate() {...}
}
```

Subclass:
```
public class LuxuryCar extends Car {
    public void playMusic() {...}
    public void ejectCD() {...}
    public void resumeMusic() {...}
    public void pauseMusic() {...}
```
Example: Overwriting a Method

Original Java-Code:
```java
class Device {
  int serialnr;
  public final void help() {....}
  public void setSerialNr(int n) {
    serialnr = n;
  }
}
class Valve extends Device {
  Position s;
  public void on() {
    ....
  }
  public void setSerialNr(int n) {
    serialnr = n;
  }
}
```

New Java-Code:
```java
class Device {
  int serialnr;
  public final void help() {....}
  public void setSerialNr(int n) {
    serialnr = n;
  }
}
class Valve extends Device {
  Position s;
  public void on() {
    ....
  }
  public void setSerialNr(int n) {
    serialnr = n + s.serialnr;
  }
}
```

Rewritable Methods:
Usually implemented with Empty Body
```java
class Device {
  int serialnr;
  public void setSerialNr(int n) {}
}
class Valve extends Device {
  Position s;
  public void on() {} 
  public void setSerialNr(int n) {
    serialnr = n + s.serialnr;
  }
}
```

UML Class Diagram

### Bad Use of Overwriting Methods

One can overwrite the operations of a superclass with completely new meanings.

**Example:**
```java
public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

- We have redefined addition as subtraction and subtraction as addition!!

### Bad Use of Implementation Inheritance

- We have delivered a car with software that allows to operate an on-board stereo system
- A customer wants to have software for a cheap stereo system to be sold by a discount store chain

**Dialog between project manager and developer:**

Project Manager:
- “Reuse the existing car software. Don’t change this software, make sure there are no hidden surprises. There is no additional budget, deliver tomorrow!”

Developer:
- “OK, we can easily create a subclass BoomBox inheriting the operations from the existing Car software!!
- “And we overwrite all method implementations from Car that have nothing to do with playing music with empty bodies!”

### What we do to save money and time

<table>
<thead>
<tr>
<th>Auto</th>
<th>Boombox</th>
</tr>
</thead>
<tbody>
<tr>
<td>engine</td>
<td>musicSystem</td>
</tr>
<tr>
<td>windows</td>
<td>playMusic()</td>
</tr>
<tr>
<td>-</td>
<td>specCD()</td>
</tr>
<tr>
<td>MusicSystem</td>
<td>resumeMusic()</td>
</tr>
<tr>
<td>brake()</td>
<td>pauseMusic()</td>
</tr>
<tr>
<td>accelerate()</td>
<td></td>
</tr>
</tbody>
</table>

**Existing Class:**
```java
public class Auto {
  public void drive() {.....}
  public void brake() {.....}
  public void accelerate() {.....}
  public void playMusic() {.....}
  public void ejectCD() {.....}
  public void resumeMusic() {.....}
  public void pauseMusic() {.....}
}
```

**Boombox:**
```java
public class Boombox extends Auto {
  public void drive() {};
  public void brake() {};
  public void accelerate() {};
}
```
Contraction

- **Contraction**: Implementations of methods in the super class are overwritten with empty bodies in the subclass to make the super class operations "invisible"
- Contraction is a special type of inheritance
- It should be avoided at all costs, but is used often.

Contraction must be avoided by all Means

A contracted subclass delivers the desired functionality expected by the client, but:
- The interface contains operations that make no sense for this class
- What is the meaning of the operation brake() for a BoomBox?

The subclass does not fit into the taxonomy
A BoomBox is not a special form of Auto
- The subclass violates Liskov’s Substitution Principle:
  - I cannot replace Auto with BoomBox to drive to work.

Revised Metamodel for Inheritance

Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.

- The key benefits of frameworks are reusability and extensibility:
  - **Reusability** leverages of the application domain knowledge and prior effort of experienced developers
  - **Extensibility** is provided by hook methods, which are overwritten by the application to extend the framework.

Classification of Frameworks

- Frameworks can be classified by their position in the software development process:
  - Infrastructure frameworks
  - Middleware frameworks

- Frameworks can also be classified by the techniques used to extend them:
  - Whitebox frameworks
  - Blackbox frameworks

Frameworks in the Development Process

- **Infrastructure frameworks** aim to simplify the software development process:
  - Used internally, usually not delivered to a client.

- **Middleware frameworks** are used to integrate existing distributed applications:
  - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].

- **Enterprise application frameworks** are application specific and focus on domains:
  - Example of application domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.
White-box and Black-box Frameworks

• **White-box frameworks:**
  - Extensibility achieved through **inheritance** and dynamic binding.
  - Existing functionality is extended by subclassing framework base classes and overriding specific methods (so-called hook methods)

• **Black-box frameworks:**
  - Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via **delegation**.

Class libraries vs. Frameworks

• **Class Library:**
  - Provide a smaller scope of reuse
  - Less domain specific
  - Class libraries are passive; no constraint on the flow of control

• **Framework:**
  - Classes cooperate for a family of related applications.
  - Frameworks are active; they affect the flow of control.

Components vs. Frameworks

• **Components:**
  - Self-contained instances of classes
  - Plugged together to form complete applications
  - Can even be reused on the binary code level
  - The advantage is that applications do not have to be recompiled when components change

• **Framework:**
  - Often used to develop components
  - Components are often plugged into blackbox frameworks.

Documenting the Object Design

• **Object design document (ODD)** = The Requirements Analysis Document (RAD) plus...
  - ...additions to object, functional and dynamic models (from the solution domain)
  - ... navigational map for object model
  - ... Specification for all classes (use Javadoc)

Documenting Object Design: ODD Conventions

• Each subsystem in a system provides a service
  - Describes the set of operations provided by the subsystem

• Specification of the service operations
  - Signature: Name of operation, fully typed parameter list and return type
  - Abstract: Describes the operation
  - Pre: Precondition for calling the operation
  - Post: Postcondition describing important state after the execution of the operation

• **Use Javadoc and Contracts for the specification of service operations**
  - Contracts are covered in the next lecture.

Package it all up

• Pack up design into discrete units that can be edited, compiled, linked, reused

• Construct physical modules
  - Ideally use one package for each subsystem
  - System decomposition might not be good for implementation.

• **Two design principles for packaging**
  - Minimize coupling:
    - Classes in client-supplier relationships are usually loosely coupled
    - Avoid large number of parameters in methods to avoid strong coupling (should be less than 4-5)
    - Avoid global data

  - Maximize cohesion: Put classes connected by associations into one package.
Packaging Heuristics

- Each subsystem service is made available by one or more interface objects within the package.
- Start with one interface object for each subsystem service.
  - Try to limit the number of interface operations (7±2).
- If an interface object has too many operations, reconsider the number of interface objects.
- If you have too many interface objects, reconsider the number of subsystems.
- Interface objects vs Java interface:
  - **Interface object**: Used during requirements analysis, system design, object design. Denotes a service or API.
  - **Java interface**: Used during implementation in Java (May or may not implement an interface object).

Summary

- **Object design closes the gap between the requirements and the machine**.
- **Object design adds details to the requirements analysis and makes implementation decisions**.
- **Object design activities include**: Identification of Reuse, Identification of Inheritance and Delegation opportunities, Component selection.
- **Interface specification (Next Lecture)**.
- **Object model restructuring**, **Object model optimization**.
- **Object design is documented in the Object Design Document (ODD)**.