SOFTWARE DESIGN PRINCIPLES

Outline

- Basics of design
- Design approaches
- Review of OO concepts
An oxymoron is defined as a phrase in which two words of contradictory meaning are brought together

- Clearly misunderstood
- Exact estimate
- Fully empty
- Seriously funny
- Original copies
- Act naturally
- Small crowd
- Pretty ugly
- Only choice

And the Mother of all...

Happily Married

What is Design?

- The process of creating the description of a solution to a problem.

  Specification document (i.e., SRS) → Design Process → Design document

- The purpose of software design is “To produce a workable (implementable) solution to a given problem.”

- Although methods, processes, and tools can help guide a designer, software design is essentially a creative activity.
Specification vs. Design Documents

- Specification document
  - A description of “what” the software is to do.

- Design document
  - A description of “how” the software is to achieve this.

Design Concepts

- The design should be based on requirements specification.

- The design should be documented (so that it supports implementation, verification, and maintenance.)

- The design should be modular (to support abstraction, parallel implementation, verification, and maintenance.)

- The design should be assessed for quality as it is being created, not after the fact.
Architectural vs. Detailed Design

- Architectural design
  - Decompose system into modules (components or subsystems).
  - Identify interrelationships (e.g., interconnections or interactions) among modules.
  - Also known as system design, general design, logical design, and high-level design.

- Detailed design
  - Design each module separately in detail (e.g., interface, data structures, and algorithms).
  - Also known as modular design, physical design, and low-level design.

Definition of Design Revisited

- The process of identifying a set of modules and interfaces from a set of requirements relying on SE techniques, imagination, experience, and art!
Design Criteria and Principles

- Simple, understandable, modular, independent, high cohesion and low coupling, …
- However, the most important principle is “design for change”
  - The design should allow maintainers to quickly modify the program to meet future needs.
  - Challenging because we may not know what the future needs are.
  - Why so important?

Outline

- Basics of design
- Design approaches
- Review of OO concepts
Approaches to Design

- Many techniques and approaches
  - Action-oriented (function-oriented)
  - Data-oriented
  - Object-oriented

- Common characteristics
  - Decomposes the system into elementary parts.
  - Applies the divide and conquer strategy to eventually build the system.

Action-Oriented Design

- Focus on the actions (functions or operations) that the system perform.

- Example?
  - E.g., DFD, Structure Design (functional decomposition).
Data-Oriented Design

- Data is considered first, i.e., the structure of data is determined first and then the procedures are designed to conform to the structure of the data.
- Example?
  - E.g., Entity-relationship diagram (ERD)

Object-Oriented Design

- Combines features of action-oriented and data-oriented designs.
- Balanced view on actions and data, as objects incorporate both actions and data.
- Seamless development from specification, design, and to implementation.
- E.g., UP, CRC Approach.
General Approach of OOD

- Objects (and classes) play a key role in OOD.
- General OOD Process
  1. Identify classes (data and operations).
  2. Group classes into subsystems.
  3. Refine subsystems.

Design Models

- A design model [Booch 99]
  - is an abstraction of reality
  - helps us understand complex systems
  - provides a blueprint for a system
  - includes elements that have broad effect on a system and omits minor elements that are not relevant to the level of abstraction of the module
Design Views

- **Architectural Design (high-level design)**
  - A high-level modular structure for the software is created.
  - Modules are identified, data and functionality are modeled, and module relationships are described.

- **Data Design**
  - The information analysis models are translated into data structure design.

- **Interface Design**
  - A description of how the software communicates with other systems and with human users is developed.

- **Temporal Design**
  - The events and signals, state transition, processes, and timing conditions and constraints are incorporated into the design model.

- **Detailed Design (procedural design)**
  - The algorithms and data structures defined to satisfy the required functionality of each module.

Review of OO Concepts

- **Object**
  - Encapsulated data and operations (or methods)
  - Has responsibilities (or services) to clients
    - Knowledge an object maintains.
    - Actions an object can perform.
Classes vs. Objects

- An object is an instance of a class.
- A class is a type, which is a set of objects having similar behavior.
- A class is a compile-time description, whereas, an object is a runtime entity.

Message, Method, and Signature

- Objects communicate with each other by sending messages.
- A message consists of
  - Name and arguments, e.g.,
    printer.printFile(myFile);
- A message is implemented by a method, the step-by-step algorithm executed in response to receiving a message.
- A signature is a formal specification of I/O of a method, i.e., name, parameter types, and return type, e.g.,
  printFile: File -> Status
Inheritance

- **Definition**
  - The ability of one class, called a **subclass**, to define the behavior and data structure of its instances as a superset of the definition of another class, called a **superclass**.

- **Benefits**
  - Allows incremental definition of classes; i.e., only need to define the difference.
  - Allows to organize classes into a **class hierarchy**.

Polymorphism

- **Definition**
  - The ability of two or more classes of objects to respond to the same message, each in its own way.

- **Benefits**
  - Allow to recognize and exploit similarities between different classes of objects

- **Examples**
  - Subtyping (subclassing) polymorphism
    
    ```java
    shape.draw(); // shape can be a Circle, Triangle, Rectangle, Square, etc.
    ```
  - Parametric polymorphism
    ```java
    3 + 4;
    3.0 + 4.0;
    “3” + “4”;
    ```
### Abstract Classes

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes not intended to produce instances of themselves.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow to factor behavior common to many subclasses.</td>
</tr>
<tr>
<td>Can provide partial implementations.</td>
</tr>
</tbody>
</table>

### Design and Abstraction

<table>
<thead>
<tr>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a design approach that emphasizes essential external features and behavior, and obscures details about internal structure and logic.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information hiding</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a concept for “abstracting” or hiding the details of a module not needed by a user of a module.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>is collecting all of the essential features and behavior of a data object into a single entity.</td>
</tr>
</tbody>
</table>
Types of Abstraction (Example of each)

- **procedural abstraction**
  - provide an interface for invoking procedures/functions without requiring knowledge of logic and implementation
  - e.g., \( y = \text{SQRT}(X); \)

- **data abstraction**
  - *abstract data type* (ADT) a set of data values and associated data operations
  - e.g., `int t1, t2;`

- **object abstraction**
  - An object is an entity (often from the real world) that has a defined identity, a state, and behavior.
  - e.g., `Student stud1 = new Student("Maria Lopez", "111-11-1111", 18);`

Encapsulation

- Encapsulation is the grouping of related ideas into one unit, which can thereafter be referred to by a single name.

- OO encapsulates the attributes and operations of an object.

- Consider the Bank Account object type.

![BankAccount](image)

- `withdraw`
- `deposit`
- `balance`
- `getBalance`
Information Hiding

- **Information hiding** is the use of encapsulation to restrict access to certain information and implementation detail that are internal to an object.

- A **public interface** for the object defines how external entities can interact with the object.
  - Design decisions are hidden, so that changes to the design can be made without affecting an object’s interaction with external entities.

<table>
<thead>
<tr>
<th>Student</th>
<th>access types</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: String</td>
<td>- private</td>
</tr>
<tr>
<td>- idNumber: String</td>
<td>+ public</td>
</tr>
<tr>
<td>- age: int</td>
<td># protected</td>
</tr>
<tr>
<td>- gpa: float</td>
<td></td>
</tr>
<tr>
<td>- major: Degree</td>
<td></td>
</tr>
<tr>
<td>+studentReport(): String</td>
<td></td>
</tr>
<tr>
<td>+changeMajor()</td>
<td></td>
</tr>
</tbody>
</table>

Object Class Relationships

- **Aggregation (Composition):** one class is made up (composed) of other classes. Example?
  - (e.g., a Window is made up of 4 Panels)

- **Inheritance:** one class is derived from another (inherits attributes and operations). Example?
  - (e.g., Temperature Sensor is a subclass of Sensor)

- **Association:** a relationship that is neither aggregation or inheritance. Example?
  - (e.g., “The Controller displays an Aircraft Record.”).
Aggregation Example

Student
- name: String
- idNumber: String
- age: int
- gpa: float
- major: Degree
+ studentReport(): String
+ changeMajor()

Class
- section
- courseID
- instructor: Teacher
- students: List
+ drop(s: Student)
+ add(s: Student)

Teacher
- name: String
- age: int
- sex: String
- title: Rank
- officeHours: String

Student
- name: String
- idNumber: String
- age: int
- gpa: float
- major: Degree
+ studentReport(): String
+ changeMajor()

Inheritance Example

Person
- name: String
- age: int
- gender: String
- friends: Pet[]

Student
- idNumber: String
- gpa: float
- major: degree
+ studentReport(): String
+ changeMajor()

Teacher
- title: Rank
- officeHours: String
**Association Example**

- Associations can be uni-directional (using an arrow head) or bi-direction (no arrows)

  ![Diagram of associations](image)

- The number of objects involved in an association relationship can be designated in various ways:
  - 1 to 1
  - 1 to many (0 or more)
  - many to many
  - numerically specified

**Open/Closed Principle**

- Software should be open for extension, but closed for modification
The Open-Closed Principle (OCP) states that we should attempt to design modules that never need to be changed.

To extend the behavior of the system, we add new code. We do not modify old code.

Modules that conform to the OCP meet two criteria:
- Open For Extension - The behavior of the module can be extended to meet new requirements.
- Closed For Modification - the source code of the module is not allowed to change.

How can we do this?
- Abstraction
- Polymorphism
- Inheritance
- Well-Defined Interfaces
- Design patterns such as template methods

Consider the following method that totals the price of each part in the specified array of parts of some class:

```java
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for (Part p: parts) {
        total += p.getPrice();
    }
    return total;
}
```

If Part is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of parts without having to be modified!

It conforms to the OCP.
Open/Closed Principle

Suppose the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price.

```java
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for (Part p : parts) {
        if (p instanceof Motherboard)
            total += (1.45 * p.getPrice());
        else if (p instanceof Memory)
            total += (1.27 * p.getPrice());
        else
            total += p.getPrice();
    }
    return total;
}
```

Does not conform to OCP. `totalPrice()` must be changed whenever Accounting changes pricing policy.

What could we do instead?

A better idea is to have a `PricePolicy` class which can be used to provide different pricing policies:

```java
public class Part {
    private double price;
    private PricePolicy pricePolicy;
    public void setPricePolicy(PricePolicy pricePolicy) {
        this.pricePolicy = pricePolicy;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public double getPrice() {
        return pricePolicy.getPrice(price);
    }
}
```

Part -------------- PricePolicy

Open/Closed Principle
Open/Closed Principle

```java
/**
 * Class PricePolicy implements a given price policy.
 */
public class PricePolicy {
    private double factor;
    public PricePolicy (double factor) { this.factor = factor; }
    public double getPrice(double price) {
        return price * factor;
    }
}
```

Pricing policies can be set dynamically by changing the PricePolicy object. Multiple pricing policy classes may exist.

OO Development – A Beginning

- How does one start OO development?
  - Typically one might start with a statement of a customer need (e.g., a one page description of a product to be developed).
    - In real-world this might be followed by study, analysis and specification of the requirements for the product.
  - Using object-oriented analysis (OOA) and the product requirements, one might next identify and specify the key objects/classes from the application (or business) domain.
Initial Object/Class Analysis

- Examine documents for nouns, adjectives, and verbs
  - Nouns can indicate classes (e.g., “Sensor”, “Window”, “Date”, “Deviation”, “Time”)
    - Or a noun might represent an attribute
    - Adjectives can indicate an inheritance relationship or an attribute (e.g., “Temperature Sensor”, “Task Window”, “Math Course”)
  - Verbs might indicate operations/methods (e.g., “read Sensor”, “display Window”, “flash Window”, “get Time”)
    - Verbs can also represent a relationship between classes. (e.g., “Each course is assigned an instructor”)
    - Make sure the operation in a class operates on objects of the class. (Typically giving information about the state of an object or changing the state of an object)

Summary

- Design represents a solution to a problem.
- Design is a creative process.
- Design should use abstraction and modularity.
- Design Components should be independent.
- Design should support verification and maintenance.
- There are two popular design methodologies:
  - structured design
  - object-oriented design
Reading Assignment

- Read Chapter 4 of “Effective Java”, by Joshua Bloch
- *Added to SmartCloud*

Quiz 2