MC322 - Object Oriented Programming
Lesson 10.1
Design Patterns

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Introduction to Design Patterns in Object-Oriented Programming

- **What Are Design Patterns?**
  - Design patterns are reusable solutions to common problems encountered in software design, providing templates and guidelines for effective problem-solving in different contexts.
  - They embody best practices derived from proven solutions, making them valuable for developing efficient and maintainable object-oriented systems.
- **Design Patterns and Reuse:**
  - Patterns fit seamlessly into the concept of reusable software development because they offer a way to document and apply strategies that have been successful across different projects.
  - This reusability aligns perfectly with the object-oriented paradigm's focus on modularity, encapsulation, and code reuse.

- **Benefits of Design Patterns:**
  - **Shared Knowledge**: By providing standard solutions, patterns facilitate knowledge sharing among developers, ensuring that proven techniques are consistently applied across projects.
  - **Best Practices**: Patterns highlight best practices for solving recurring challenges, helping developers avoid common pitfalls and saving valuable time in the design process.
  - **Improved Communication**: Patterns provide a common vocabulary that developers can use to describe solutions, making design discussions more precise and collaborative.
- **Key Takeaway:**
  - The influence of design patterns on object-oriented development is significant. They bridge the gap between theory and practical application, allowing developers to implement robust, scalable, and maintainable software based on successful precedents.

Why Design Patterns?

The Concept of Design Patterns

- The concept of design patterns did not necessarily start with the need for reusable software.
- The seminal work on design patterns is about constructing buildings and cities.
- Christopher Alexander noted in **A Pattern Language: Towns, Buildings, Construction**:

  > *"Each pattern describes a problem that 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 the solution a million times over, without ever doing it the same way twice."*

- Four Elements of a Pattern: Name, Problem, Solution, Consequences.

The Four Elements of a Pattern

- **Pattern Name**: A handle to describe a design problem, its solutions, and consequences in just a word or two.
  - Naming a pattern increases our design vocabulary and lets us design at a higher level of abstraction.
  - Having this vocabulary allows easier communication with colleagues, in documentation, and even for personal use.
  - Good pattern names make it easier to discuss designs and their tradeoffs with others.
  - Finding suitable names has been one of the hardest parts of developing our catalog.

The Four Elements of a Pattern

- **The Problem**: Describes when to apply the pattern and provides context.
  - Explains the design problem, such as how to represent algorithms as objects.
  - May identify class or object structures that indicate inflexible design.
  - Sometimes includes conditions that must be met before the pattern is applicable.

The Four Elements of a Pattern

- **The Solution**: Describes the elements that constitute the design, including their relationships, responsibilities, and collaborations.
  - Doesn't specify a particular concrete design or implementation because a pattern serves as a flexible template.
  - Provides an abstract description of a design problem and how a general arrangement of elements can solve it.
  - The arrangement typically includes classes and objects to form a generalized solution.

The Four Elements of a Pattern

- **The Consequences**: Results and trade-offs of applying the pattern.
  - Consequences are crucial for evaluating design alternatives and understanding the costs and benefits.
  - Software consequences often relate to space and time trade-offs, language, and implementation issues.
  - Patterns affect flexibility, extensibility, and portability, particularly in object-oriented design.
  - Explicitly listing the consequences aids in understanding and evaluating them.

Smalltalk's Model/View/Controller

Historical Perspective on MVC

- The Model/View/Controller (MVC) pattern, introduced in Smalltalk, is often used to illustrate the origins of design patterns.
- The MVC paradigm was initially developed for creating user interfaces in Smalltalk.
- Smalltalk was one of the first popular object-oriented languages.

Smalltalk

Smalltalk is the result of several great ideas that emerged from Xerox PARC, including the use of a mouse and a windowing environment. It is a wonderful language that provided the foundation for subsequent object-oriented languages.

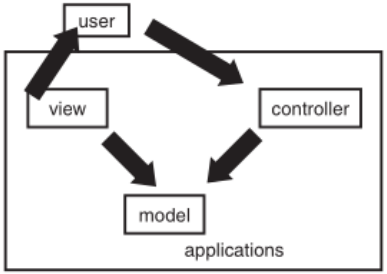
One of the criticisms of C++ is that it isn't truly object-oriented, whereas Smalltalk is. Although C++ had a larger following in the early days of OO, Smalltalk has always had a dedicated core group of supporters. Java is also mostly OO and has embraced the C++ developer base.

MVC Components Defined by Design Patterns

- The Model is the application object.
- The View is the screen presentation.
- The Controller defines how the user interface reacts to user input.

- **Previous Paradigms**: The Model, View, and Controller were often combined into a single entity.
- **MVC Paradigm**:
  - Separates these components into distinct interfaces.
  - Allows changing the user interface by modifying only the View.

Figure 15.1 illustrates what the MVC design looks like.



Constructing an object.

Separation of Interfaces and Implementation

- **Interface vs. Implementation**:
  - Object-oriented development focuses on separating the interface from the implementation as much as possible.
  - We should also aim to keep distinct interfaces separate.
  - Avoid combining unrelated interfaces that are not linked to solving the problem at hand.
- **MVC Paradigm**:
  - One of the early pioneers in interface separation.
  - Explicitly defines interfaces between components to solve the common problem of creating user interfaces.
  - Provides a structure for connecting user interfaces to the business logic and data.

Advantages and Drawbacks of MVC

- Following **MVC concept** and separate the user interface, business logic, and data:
  - Your system will be more flexible and robust.
  - Changing the GUI won't affect business logic or data.
  - Adjusting business logic won't require GUI changes.
  - Modifying how data is stored won't affect GUI or business logic, assuming interfaces between the three don't change.

MVC Example

Consider a GUI with a list of phone numbers as an example involving a list box. The list box is the view, the phone list is the model, and the controller is the logic binding the list box to the phone list.

MVC Drawbacks

Although the MVC is a great design, it can become complex due to the required upfront design attention. Object-oriented design has the challenge of balancing good and cumbersome design. The key question is how much complexity should be included.

Types of Design Patterns

Design Patterns: Categorization and Historical Context

- **23 Patterns Grouped into Three Categories**:
  - Most examples are in C++, with some in Smalltalk.
  - The book's publication period reflects the popularity of C++ and Smalltalk.
- **Historical Context**:
  - Published in 1995, during the Internet revolution and Java's rise.
  - The value of design patterns spurred interest, leading to many books.
  - Later publications were written largely in Java due to its growing popularity.

Categories of Patterns

- **Language Irrelevance**:
  - The specific programming language is irrelevant, as the book is focused on design.
  - Patterns can be implemented in any language.
- **Three Categories of Patterns**:
  - **Creational Patterns**: Create objects for you, providing flexibility in object instantiation.
  - **Structural Patterns**: Compose groups of objects into larger structures like complex UIs or accounting data.
  - **Behavioral Patterns**: Define communication and flow between objects in complex programs.

Creational Patterns

Creational Patterns Categories

- **Creational Patterns** consist of:
  - **Abstract Factory**
  - **Builder**
  - **Factory Method**
  - **Prototype**
  - **Singleton**

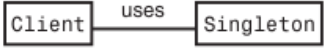
Scope: This chapter aims to describe what a design pattern is rather than cover each pattern in the GoF book.

Example Pattern: We'll focus on a single pattern from each category. Let's consider an example of a creational pattern by looking at the Singleton pattern.

The Singleton Design Pattern

Definition: The Singleton pattern is a creational pattern that ensures a class has only a single instance and provides a global point of access to that instance.

- **Purpose**: Regulates object creation to one instance per class.
- **Example Use Case**:
  - A website counter object tracks the hits on a page.
  - The counter object should not be re-instantiated each time a page is loaded.
  - A single instance is created with the first hit and reused to increment the count afterward.



The singleton model.

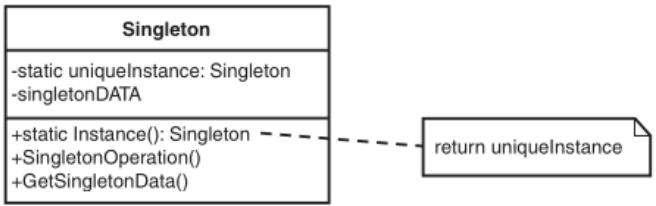
Taking Care of Business

Remember, one of the most important **object-oriented rules** is that an object should take care of itself. This means that issues regarding the life cycle of a class should be handled within the class itself, not delegated to language constructs like `static`.

UML Model for Singleton

Figure shows the UML model for the Singleton pattern as presented in **Design Patterns**.

- The **uniqueinstance** property is a static singleton object.
- The **Instance()** method provides access to the unique instance.
- Other properties and methods support the business logic of the class.



The singleton model.

Singleton Pattern Code Example

Access and Instance Control:

- Any class needing the singleton instance should use the **getInstance()** method.
- The constructor controls object creation like any OO design.
- The **getInstance()** method handles instantiation:
  - Checks if the singleton instance is `null`.
  - Instantiates a new object if required.
  - Returns the singleton instance.

Java Code Example (Classic Singleton):

```
public class ClassicSingleton {
    private static ClassicSingleton instance = null;
    protected ClassicSingleton() {}
    public static ClassicSingleton getInstance() {
        if (instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }
}
```

Multiple References to a Singleton

Managing Multiple References:

- Each reference in the application pointing to the singleton must be managed properly.
- Multiple references can coexist but should refer to the same singleton instance.

```
public class Singleton {
    public static void main(String[] args) {
        Counter counter1 = Counter.getInstance();
        System.out.println("Counter: " + counter1.getCounter());
        Counter counter2 = Counter.getInstance();
        System.out.println("Counter: " + counter2.getCounter());
    }
}
```

Explanation of the Example:

- Both `counter1` and `counter2` point to the same Counter instance.
- The constructor is not directly used; object creation is managed by the `getInstance()` method.

Proving Singleton References

Purpose: Demonstrate that both `counter1` and `counter2` refer to the same instance.

```
package Counter;

public class Singleton {
    public static void main(String[] args) {
        Counter counter1 = Counter.getInstance();
        counter1.incrementCounter();
        counter1.incrementCounter();
        System.out.println("Counter: " + counter1.getCounter());

        Counter counter2 = Counter.getInstance();
        counter2.incrementCounter();
        System.out.println("Counter: " + counter2.getCounter());
    }
}
```

Explanation:

- `counter1` increments twice before displaying its value.
- `counter2` then increments once and displays its value.
- Both references output the same, cumulative count, proving that they're pointing to the same instance.

Structural Patterns

Structural Design Patterns

Purpose of Structural Patterns:

- Structural patterns are used to create larger structures from groups of objects.
- These patterns help separate interfaces from implementations.

Members of the Structural Category:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Adapter Pattern:

- One of the most important design patterns.
- Effectively separates the implementation from the interface.
- Facilitates compatibility between different interfaces.

Adapter Pattern Example

Adapter Pattern Overview:

- Allows creating a different interface for an existing class.
- Provides a class wrapper to adapt functionality.
- Enables compatibility with existing interfaces or newer designs.

Java Wrapper Example:

- Primitives like `int` aren't objects in Java, but sometimes need object behavior.
- The `Integer` class wraps a primitive `int`, providing object-like functionality.

Advantages of Wrapping:

- Treats the original primitive as an object.
- Provides the benefits of object-oriented programming, like conversions.

Code Example:

```
// Primitive integer
int myInt = 10;

// Wrap in an Integer object
Integer myIntWrapper = new
     Integer(myInt);

// Convert to string
String myString =
     myIntWrapper.toString();
```

Adapter Pattern: Mail Client Example

Mail Client Scenario:

- A mail tool interface provides a robust set of features but needs an API adjustment.
- The objective is to change the API to retrieve mail without altering existing functionality.

```
package MailTool;

public class MailTool {
    public MailTool() {
    }

    public int retrieveMail() {
        System.out.println("You've Got Mail");
        return 0;
    }
}
```

Adapter Pattern Solution:

- The adapter pattern wraps the existing mail tool class with a new interface.
- This ensures compatibility with existing code while providing the desired API adjustment.

Adapter Pattern: Wrapping a Mail Tool

Original Method:

- The `retrieveMail()` method in the original mail client displays a greeting: `"You've Got Mail"`.

New Interface Requirement:

- Need to change the interface from `retrieveMail()` to `getMail()` in all company clients.

Adapter Pattern Benefits:

- `MyMailTool` wraps `MailTool`, allowing `getMail()` to internally call `retrieveMail()`.
- This ensures compatibility while maintaining a unified interface.

Mail Interface Definition:

```
package MailTool;
interface MailInterface {
    int getMail();
}
```

Adapter Implementation:

```
package MailTool;
class MyMailTool implements
     MailInterface {
    private MailTool yourMailTool;

    public MyMailTool() {
        yourMailTool = new MailTool();
    }

    public int getMail() {
        return
            yourMailTool.retrieveMail();
    }
}
```

Behavioral Patterns

Behavioral Patterns: Iterator Example

Behavioral Patterns Categ:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Iterator Pattern Overview:

- Provides a way to sequentially access elements of a collection without exposing its internal structure.
- Implemented by several programming languages, offering a consistent way to navigate through data structures.
- Used to iterate through lists, arrays, and other collections.

Key Features:

- Separates traversal logic from collection logic, providing flexibility.
- Supports different types of traversal, like forward or backward iteration.
- Allows multiple traversals simultaneously on the same collection.

The Iterator Design Pattern

Iterator Pattern Overview:

- Provides a standard mechanism for traversing a collection like a vector or list.
- Facilitates accessing each item of a collection sequentially.
- Offers information hiding to keep the internal structure secure.
- Allows multiple iterators to coexist without interference.

Key Features of Java Iterator:

- The `iterate()` method uses an enhanced for loop to traverse and print elements.
- The iterator pattern makes this traversal easy and secure.

Java Iterator Implementation

```
public class Iterator {
    public static void main(String args[]) {
        // Instantiate an ArrayList
        ArrayList<String> names = new ArrayList<>();
        // Add values to the ArrayList
        names.add("Joe");
        names.add("Mary");
        names.add("Bob");
        names.add("Sue");

        // Now iterate through the names
        System.out.println("Names:");
        iterate(names);
    }

    private static void iterate(ArrayList<String> arl) {
        for (String listItem : arl) {
            System.out.println(listItem.toString());
        }
    }
}
```

Better example: Guru:link

Antipatterns

Antipatterns

Definition:

- Collections of past negative experiences where design solutions went wrong.
- Serve as practices to avoid, opposite to proactive design patterns.

Antipattern Characteristics:

- Most software projects face failure due to poor design decisions.
- Highlight problematic approaches, often leading to outright project cancellations.
- Derived from reactionary, flawed experiences in software development.

Facets of Antipatterns (Koenig, 1995):

- **Bad Solutions**: Describe ineffective approaches that lead to adverse situations.
- **Recovery Paths**: Outline how to escape these situations and reach a better solution.

Reference: Johnny Johnson's article, "Creating Chaos," discusses the prevalence of project cancellations.

The Utility of Antipatterns

Why Antipatterns Are Useful:

- They help identify root causes of design issues that have already occurred.
- The concept of root-cause analysis allows the examination of failed design patterns.
- Antipatterns emerge from the failure of previous solutions, providing valuable hindsight.

Scott Ambler's Reuse Patterns and Antipatterns:

- **Robust Artifact Pattern**:
  - Well-documented and thoroughly tested item that meets general needs.
  - Multiple examples show how to work with it.
  - Easy to understand and work with, making it more reusable.
- **Reuseless Artifact Antipattern**:
  - An artifact mistakenly declared reusable but never actually reused.
  - Requires reworking to become a robust artifact that meets wider requirements.

Continuous Refactoring: Antipatterns promote revising and refactoring designs to find workable solutions.

Conclusion

Chapter Summary

Concept of Design Patterns:

- Design patterns are rooted in everyday life and should shape how we think about object-oriented design.
- Solutions are often derived from real-life situations, providing practical relevance.

Further Exploration:

- This chapter covered design patterns briefly but highlighted their importance.
- To delve deeper into design patterns and their practical applications, consult the recommended books listed at the chapter's end.

More Patterns: refactoring.guru

MC322 - Object Oriented Programming
Lesson 10.1
Design Patterns

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP