UML and Structural Design Patterns: Comprehensive Study Notes

Based on the lecture slides for BCS1430, Spring 2025

Contents

1 Introduction

This document provides a detailed review of **UML (Unified Modeling Language)** and the **Structural Design Patterns** from the lecture materials. It includes:

- Key UML diagram types and their purposes
- Essential UML elements: classes, relationships, use cases, etc.
- Structural patterns: Adapter, Composite, Decorator, Facade, Proxy (and references to others)
- Extra tips, memory hooks, and best practices

2 Unified Modeling Language (UML) Overview

2.1 What is UML?

UML is a standardized visual language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It is both:

1. A **notation**: graphical syntax (diagrams, symbols).
2. A **metamodel**: a formal specification of elements and their relationships.

Key Benefits

- **Communication**: Helps teams and stakeholders understand system structure and behavior.
- **Specification/Blueprint**: Clear architecture and requirements.
- **Documentation**: Eases maintenance and future modifications.

2.2 Main Diagram Categories

UML is often divided into **Structural** and **Behavioral** diagrams:

- **Structural diagrams**: Class, Object, Component, Composite Structure, Deployment, Package.
- **Behavioral diagrams**: Use Case, Activity, State Machine, plus the family of *Interaction* diagrams (Sequence, Communication, Interaction Overview, Timing).

3 Use Case Diagrams (Behavioral)

3.1 Purpose

- Show how **actors** (people or other systems) interact with the system to achieve goals.
- High-level system functionality from an end-user perspective.
- Useful early in the analysis phase to capture requirements.

Main Elements

Actors External entities (users, other systems). Represented by stick figures or labeled icons.

Use Cases Oval shapes naming the user goals. E.g., *"Rent Car"*, *"Pay for Order"*.

System Boundary A rectangle encloses the set of use cases, showing the scope of the system.

Relationships
- `<<include>>`: Common sub-use-case that is unconditionally used by multiple use cases.
- `<<extend>>`: Optional or conditional behavior extending a main use case.
- `Generalization`: Actors or use cases can inherit from each other.

Memory Hook

Use Case = user's objective. Always ask: *"What is the user trying to accomplish?"*

4 Class Diagrams (Structural)

4.1 Purpose

- Depict the **static** structure: classes, attributes, operations, and relationships.
- Provide a blueprint for system implementation.

4.2 Key Notation

Class: shown as a box with up to three compartments:

1. Class Name
2. Attributes (name, type, visibility)
3. Operations (methods, parameters, visibility)

Example Class Diagram Syntax in LaTeX

```
+---------------------+
|     Car             |
+---------------------+
| - color: String     |
| - make:  String     |
+---------------------+
| + drive(): void     |
| + park(): void      |
+---------------------+
```

4.3 Relationships

Association A generic link between two classes.

Multiplicity e.g. `1..*`, `0..*`, specifying how many instances of one class relate to another.

Generalization (Inheritance): `Car` inherits from `Vehicle`.

Aggregation/Composition Special forms of association for *whole–part* relationships (e.g. a Car has Wheels).

5 Object Diagrams (Structural)

5.1 Purpose

- A **snapshot** at run-time, showing instances (objects) and links between them.
- Helps illustrate example configurations or test scenarios.

5.2 Notation

- Objects typically denoted with their name and class (e.g., `myCar:Car`).
- Links show the references between objects (like association instances).

6 Interaction Diagrams (Behavioral)

6.1 Sequence Diagrams

- Show the order of messages between objects over time.
- **Lifeline**: dashed vertical line from the object/actor's rectangle.
- **Activation**: narrow rectangles indicating a focus of control.

- **Messages**: arrows labeled with method calls, optionally showing parameters/return values.

Memory Hook

Think of them like a *vertical timeline* of how objects talk to each other.

7 Design Patterns Overview

7.1 What is a Design Pattern?

- A **general, reusable solution** to a recurring design problem in software.
- Patterns have *names*, *intents*, *applicability*, *participants*, *structure*, and *consequences*.

7.2 Pattern Classification

- **Creational**: Ways to create objects (e.g., Factory Method, Abstract Factory, Singleton).
- **Structural**: Ways to combine/structure classes and objects.
- **Behavioral**: Ways for objects to communicate or assign responsibilities.

Memory Hook

Use design patterns to *avoid reinventing the wheel*. They provide standard solutions and a shared vocabulary (*e.g. "Use a Decorator here!"*).

8 Structural Design Patterns

These patterns focus on **how classes and objects are composed** to form larger structures, while maintaining flexibility and efficiency.

8.1 Adapter Pattern

Intent: Convert the interface of a class into another interface clients expect, so classes that otherwise couldn't work together can collaborate.

Key Idea Wrap an incompatible object in an "adapter" that translates calls.

When to Use You have a legacy/third-party class whose interface differs from what your client code needs.

Structure Example

```
  Client ----> (Target interface) ----> Adapter ----> Adaptee

// The adapter implements the Target's interface
// internally calling the Adaptee's methods
```

Quick Code Example (Java-ish Pseudocode)

```
interface AnalyticsStatsProvider {
    // The target interface
    JSON getStats();
}

class StockMarketApp {
    // Existing class with an incompatible interface
    public XML getMarketData() { ... }
}

class StockMarketAdapter implements AnalyticsStatsProvider {
    private StockMarketApp app;

    public StockMarketAdapter(StockMarketApp app) {
        this.app = app;
    }

    @Override
    public JSON getStats() {
        // Convert the XML from app.getMarketData() into JSON
        XML xmlData = app.getMarketData();
        JSON converted = convertXMLtoJSON(xmlData);
        return converted;
    }
}
```

Memory Hook: *"Adapter = plug converter."* Like using a travel adapter for different outlets.

8.2 Composite Pattern

Intent: Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions uniformly.

Key Idea Use a *Component* interface that both leaf and composite classes implement.

When to Use You have hierarchical data, or you want to deal with single objects and entire sub-trees in a uniform manner.

Structure

```
              Component
               /    \
     Leaf <----      ----> Composite
```

Typical Methods

- `operation()`: shared method each node or composite implements.
- `add(Component c)` / `remove(Component c)`: relevant only in **Composite**.

Memory Hook: *"Composite = trees"*. If you want to treat a single item and a group of items the same way, it's Composite.

8.3 Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically. A flexible alternative to subclassing for extending functionality.

Key Idea Place the original object inside a "wrapper" that adds behavior before/after delegating to the real object.

When to Use You want to add functionality without affecting other instances, or you have too many subclasses for every possible feature.

Structure

```
            Component Interface
             |              ^
             v              |
         ConcreteComponent  |
             ^              |
             | (has a)      |
         Decorator ---------+
```

Memory Hook: *"Decorator = wrapping gifts"*. You wrap an object in multiple layers for new features.

8.4 Facade Pattern

Intent: Provide a unified interface to a set of interfaces in a subsystem, making it easier to use.

Key Idea A *Facade* class wraps complex subsystems behind a simple interface.

When to Use You want to shield the complexity of a subsystem from the client or **decouple** the subsystem from the rest of the code.

Structure Example

```
Client ------ Facade ------> SubsystemA
                       ---> SubsystemB
                       ---> SubsystemC
```

Memory Hook: *“Facade = front desk at a hotel”*. It provides a single point of contact to many behind-the-scenes services.

8.5 Proxy Pattern

Intent: Provide a surrogate or placeholder for another object to control access to it.

- **Remote Proxy** for distributed objects.
- **Virtual Proxy** for lazy loading or caching.
- **Protection Proxy** for access control.

Structure

```
Client ----> Proxy ----> RealSubject
```

Memory Hook

“Proxy = stand-in”. The proxy *represents* the real object and can add extra logic before letting you access it.

9 Extra Key Points & Memory Hooks

- **UML is about communication**: The *exact* UML notation can be relaxed if it helps you convey the design more clearly.
- **Patterns are language-independent**: They solve conceptual problems, not just code-level issues.
- **Combine patterns carefully**: Patterns can overlap or be used together (e.g., Composite + Iterator, Decorator + Composite).
- **When in doubt, KISS (Keep It Simple)**: Don’t overuse patterns. Use them only when they *truly* solve a problem.
- **Practice with examples**: Build small UML diagrams. Write mini snippet implementations of patterns to memorize.

Design Pattern Memory Tip:

- **Adapter** = *"Convert interface A into interface B."*
- **Composite** = *"Treat single and group same way (tree)."*
- **Decorator** = *"Add features dynamically (wrap)."*
- **Facade** = *"Unified interface for a complex subsystem."*
- **Proxy** = *"Access control or placeholder for RealSubject."*

10 Suggested Study Approach & Tips

10.1 Study Flow

1. **Revisit UML Diagrams**:
    - Sketch a simple system with Class and Use Case diagrams.
    - Practice an example Sequence Diagram for one scenario in that system.
2. **Patterns Breakdown**:
    - **Read the intent** of each pattern carefully.
    - Check at least one **code example** of each pattern.
    - Try to link each pattern to a **real-world analogy**.
3. **Quizzes or Flashcards**:
    - *"Which pattern suits bridging two incompatible interfaces?"* → ***Adapter***
    - *"Which pattern to add behavior to an instance at runtime?"* → ***Decorator***

10.2 Extra Materials

- **Reading**: *"Design Patterns: Elements of Reusable Object-Oriented Software"* (Gang of Four).
- **Online UML Tools**: e.g., PlantUML, Lucidchart, StarUML.
- **Diagrams Practice**: Draw UML for a known system (like a simple e-commerce site).

11 Conclusion

- **UML** provides the standard language to model your system's structure and behavior.
- **Structural Patterns** help you build maintainable, flexible, object-oriented architectures.
- **Knowing UML + Patterns** is crucial for clear communication and robust design.

End of Document

Compiled with insights from the BCS1430 lecture slides (Spring 2025).