Ultimate One-Stop Study Document

Refactoring, Code Smells, and UML

Based on Lectures by Dr. Ashish Sai
BCS1430

February 18, 2025

Contents

Chapter 1

Introduction and Overview

Software development is a craft where maintaining clarity, flexibility, and simplicity is as important as implementing functionality. In these lectures, you learned two core topics:

1. **Refactoring and Code Smells:** Techniques to improve code quality without altering external behavior.
2. **Unified Modeling Language (UML):** A standardized visual language for specifying, constructing, and documenting software systems.

This document is designed as your ultimate, one-stop reference to deeply understand these topics and to help you in exam preparation and daily programming practice.

Memory Hook: Key idea: Great software is built on clear design and clean code.

Chapter 2

Part I: Refactoring and Code Smells

2.1 What is Refactoring?

Refactoring is the systematic process of restructuring existing code without changing its external behavior. It is a disciplined way to clean up your code and improve its internal structure.

2.1.1 Why Refactor?

The primary motivations for refactoring are:

- **Enhanced Readability:** Clearer code is easier to understand and maintain.
- **Reduced Technical Debt:** Choosing quick fixes can lead to higher future costs; refactoring addresses this debt.
- **Facilitated Future Modifications:** Clean code makes it easier to add features or fix bugs.
- **Improved Testing:** Smaller, well-defined methods are easier to test.

Memory Hook: Remember: "Clean code is easier to change."

2.2 When to Refactor

Refactoring is not a one-time task; it should be integrated into your development cycle:

- **Before Adding New Features:** Clean up existing code to ensure new code integrates well.
- **During Bug Fixes:** Simplify code around the bug to isolate issues.
- **In Code Reviews:** Incorporate improvements when reviewing others' code.
- **Regularly:** Small, continuous refactoring avoids large-scale redesigns.

Memory Hook: Think of refactoring as regular maintenance—like oil changes for your car.

2.3 Refactoring Techniques and Examples

2.3.1 Extract Method

The **Extract Method** technique involves taking a block of code that performs a distinct function and moving it into its own method with a descriptive name.

Example: Video Store Calculation

Consider a method that calculates the total amount for a customer's rentals:

```
class Customer {
    private List<Rental> rentals;

    public double calculateTotalAmount() {
        double totalAmount = 0;
        for (Rental rental : rentals) {
            double amount = 0;
            switch (rental.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    amount = 2;
                    break;
                case Movie.NEW_RELEASE:
                    amount = 3;
                    break;
                case Movie.CHILDRENS:
                    amount = 1.5;
                    break;
                default:
                    amount = 0;
                    break;
            }
            totalAmount += amount;
        }
        return totalAmount;
    }
}
```

Refactor by extracting the calculation into a separate method:

```
public double amountFor() {
    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            return 2;
        case Movie.NEW_RELEASE:
            return 3;
        case Movie.CHILDRENS:
            return 1.5;
        default:
            return 0;
    }
}
```

Now, `calculateTotalAmount()` becomes more readable:

```java
public double calculateTotalAmount() {
    double totalAmount = 0;
    for (Rental rental : rentals) {
        totalAmount += rental.amountFor();
    }
    return totalAmount;
}
```

Memory Hook: Always test after extraction to ensure that external behavior does not change.

2.3.2 Replace Temp with Query

This technique advises replacing temporary variables with method calls that return the same value, ensuring data is always current and reducing side effects.

2.3.3 Introduce Parameter Object

When a method has many parameters that are always passed together, bundle them into a single object. For example:

```java
public void createReservation(Date start, Date end, Guest guest, Room room) {
    // ...
}
```

can be refactored to:

```java
public class ReservationRequest {
    private Date start, end;
    private Guest guest;
    private Room room;
    // Constructors, getters, setters...
}

public void createReservation(ReservationRequest request) {
    // ...
}
```

Memory Hook: Grouping related parameters clarifies the method's purpose and reduces errors.

2.3.4 Replace Conditionals with Polymorphism

Switch statements or complex if-else chains can often be replaced by using polymorphism. For instance, instead of:

```
switch (movie.getPriceCode()) {
    case Movie.REGULAR: amount = 2; break;
    case Movie.NEW_RELEASE: amount = 3; break;
    // ...
}
```

Refactor by creating subclasses for each movie type, each implementing its own `getCharge()` method:

```
public abstract class Movie {
    private String title;
    public Movie(String title) { this.title = title; }
    public String getTitle() { return title; }
    public abstract double getCharge();
}

public class RegularMovie extends Movie {
    public RegularMovie(String title) { super(title); }
    public double getCharge() { return 2; }
}

// Similarly for NewReleaseMovie and ChildrensMovie
```

Memory Hook: Polymorphism reduces conditionals, leading to code that is easier to extend and maintain.

2.4 Code Smells: Recognizing the Red Flags

Code smells are indications that your code may require refactoring. Here are several common types:

2.4.1 Bloaters

Bloaters are methods or classes that have grown too large. They are hard to understand, maintain, and often hide bugs.

- **Long Method:** A method with too many lines of code.
- **Large Class:** A class that has taken on too many responsibilities.

Memory Hook: When you see a 100+ line method, it's likely a bloater waiting for refactoring.

2.4.2 Duplicated Code

When the same or similar code appears in multiple places, maintenance becomes a challenge. If one change is required, multiple spots need to be updated.

Memory Hook: DRY ("Don't Repeat Yourself") is a key principle here.

2.4.3 Long Parameter Lists

Methods that require many parameters are confusing and error-prone. Use parameter objects to simplify.

Memory Hook: Fewer parameters mean easier-to-read methods.

2.4.4 Primitive Obsession

Overuse of primitive data types (e.g., strings, integers) for complex concepts may signal that a small class should be introduced.

Memory Hook: For example, instead of passing a string to represent a phone number, create a `Phone` class with its own validation and formatting methods.

2.4.5 Data Clumps

When groups of data (e.g., first name, last name, address) always appear together, they should be encapsulated into their own class.

Memory Hook: This not only simplifies method signatures but also clarifies the domain.

2.4.6 Feature Envy and Message Chains

- **Feature Envy:** A method that heavily uses data or methods of another class might belong in that class.
- **Message Chains:** Long chains like `order.getCustomer().getAddress().getZipCode()` indicate tight coupling.

Memory Hook: Refactor by moving methods or hiding the chain behind a single call.

2.5 Refactoring Recap

Summary: Refactoring is a set of techniques designed to improve the design, clarity, and maintainability of code without altering its observable behavior. Code smells serve as red flags for areas that need refactoring. Techniques such as extracting methods, introducing parameter objects, and leveraging polymorphism are essential tools.

Chapter 3

Part II: Unified Modeling Language (UML)

3.1 Introduction to UML

The **Unified Modeling Language (UML)** is a standardized language that provides a suite of graphical notation techniques to create visual models of object-oriented software systems. Developed by the Object Management Group (OMG), UML is used by software designers to:

- **Specify** the structure and behavior of systems.
- **Visualize** system components and interactions.
- **Construct** detailed blueprints for software.
- **Document** design decisions for future reference.

Memory Hook: UML is the "blueprint" language that helps translate abstract ideas into a tangible design.

3.2 UML in Analysis and Design

UML supports both the analysis and design phases:

- **Analysis:** Capture the problem domain by modeling actors, requirements, and interactions (e.g., use case diagrams).
- **Design:** Plan the system architecture with detailed models such as class diagrams, component diagrams, and deployment diagrams.

Memory Hook: Early UML modeling helps you catch design flaws before writing a single line of code.

3.3 Types of UML Diagrams

UML diagrams are categorized into three main types:

3.3.1 Structural Diagrams

These depict the static structure of a system:

- **Class Diagrams:** Illustrate classes, attributes, operations, and relationships.
- **Object Diagrams:** Provide snapshots of object instances and their relationships at a given moment.
- Other diagrams include Component, Composite Structure, Deployment, Package, and Profile diagrams.

Memory Hook: Class diagrams are the most common; they help you "see" the blueprint of your system.

3.3.2 Behavioral Diagrams

These capture dynamic behavior:

- **Use Case Diagrams:** Show interactions between actors (users, systems) and use cases (system functions).
- **Activity Diagrams:** Represent workflows and control flows.
- **State Machine Diagrams:** Model the states of objects and transitions between states.

Memory Hook: Behavioral diagrams help clarify "what the system does" from a user's perspective.

3.3.3 Interaction Diagrams

These are a subset of behavioral diagrams focusing on object interactions:

- **Sequence Diagrams:** Detail the order of messages exchanged between objects over time.
- Additional diagrams include Communication, Interaction Overview, and Timing diagrams.

Memory Hook: Sequence diagrams are particularly useful for understanding the flow of complex operations.

3.4 Use Case Diagrams

Use case diagrams are written narratives that describe how various actors interact with the system. They typically include:

- **Actors:** Users or other systems that interact with your system.
- **Use Cases:** Specific tasks or functions that the system performs.
- **Relationships:** Such as `include` (mandatory sub-flows), `extend` (optional/alternative flows), or generalization.

Example: An ATM system use case might include:

- Actor: Customer.
- Use Cases: Withdraw Cash, Check Balance, Deposit Funds.
- Relationships: `Include` the authentication process for each use case.

Memory Hook: When writing use cases, focus on clear, non-technical language to capture the user's journey.

3.5 Class Diagrams

Class diagrams are the core of UML modeling. They represent the static structure of a system:

- **Classes:** Denote entities with attributes (data) and operations (methods).
- **Attributes:** Listed with visibility symbols: `+` (public), `-` (private), `#` (protected).
- **Operations:** Methods that the class can perform, with parameters and return types.
- **Relationships:** Include associations (e.g., one-to-one, one-to-many), generalizations (inheritance), dependencies, aggregations, and compositions.

Example: A simplified `Car` class diagram might look like:

```
 Car
- color:  String
- make:  String
- model:  String
+ startEngine():  boolean
+ drive(direction:  String, speed:  int):  String
+ park():  void
```

Memory Hook: Notice how class diagrams capture both the data (attributes) and behaviors (methods) of classes.

3.6 Sequence Diagrams and Other Interaction Diagrams

Sequence diagrams illustrate how objects interact in a time-ordered fashion:

- **Lifelines:** Vertical dashed lines representing the existence of an object.
- **Activation Bars:** Narrow rectangles on a lifeline showing when an object is active.
- **Messages:** Horizontal arrows indicating communication. Solid arrows are synchronous (blocking), and dashed arrows represent return messages.

Memory Hook: Sequence diagrams are great for visualizing the order of events in processes such as order processing or login sequences.

3.7 UML Recap and Best Practices

Summary: UML is a versatile toolkit for modeling software systems. Use structural diagrams to capture system architecture and behavioral diagrams to capture interactions. Use case diagrams, class diagrams, and sequence diagrams are particularly useful for planning, communicating, and documenting your design.

Memory Hook: A well-modeled system is like a detailed map—it guides you and your team through the development journey.

Chapter 4

Part III: Final Summary, Practical Tips, and Further Reading

4.1 Overall Summary

This ultimate study guide has covered two major areas from Week 3:

1. **Refactoring and Code Smells:**
    - **Refactoring** is the art of restructuring code to improve clarity, reduce technical debt, and facilitate future modifications.
    - Techniques include Extract Method, Replace Temp with Query, Introduce Parameter Object, and leveraging Polymorphism.
    - **Code smells**—such as bloated methods, duplicated code, long parameter lists, primitive obsession, and feature envy—signal where refactoring is needed.
2. **Unified Modeling Language (UML):**
    - UML provides a standardized set of diagrams for modeling both the static structure and dynamic behavior of systems.
    - Key diagrams include Use Case, Class, and Sequence Diagrams.
    - UML helps in both analysis (understanding requirements) and design (planning the solution).

Memory Hook: Overall Takeaway: High-quality software emerges from clean code and clear design.

4.2 Practical Tips for Effective Refactoring and Modeling

4.2.1 Tips for Refactoring

- **Make small, incremental changes** and test after each change.
- **Focus on one smell at a time.** Prioritize areas where code is most complex or error-prone.
- **Keep the external behavior unchanged.** Use unit tests to ensure functionality is preserved.
- **Use meaningful names** for extracted methods to clarify their purpose.

4.2.2 Tips for Using UML Effectively

- **Start with high-level diagrams.** Use use case diagrams to capture overall requirements before diving into detailed class diagrams.
- **Iterate your design.** UML diagrams should evolve as your understanding deepens.
- **Use simple, clear language.** Diagrams should be understandable by both technical and non-technical stakeholders.
- **Validate your models.** Compare your UML diagrams with actual code or prototypes to ensure accuracy.

Memory Hook: Always remember: A good model is one that both guides the design and communicates it clearly.

4.3 Additional Resources and Further Reading

To deepen your understanding, consider reviewing:

- **Refactoring: Improving the Design of Existing Code** by Martin Fowler.
- **Clean Code: A Handbook of Agile Software Craftsmanship** by Robert C. Martin.
- **UML Distilled: A Brief Guide to the Standard Object Modeling Language** by Martin Fowler.
- Online resources such as the Refactoring Guru website.

4.4 Final Memory Hooks and Exam Preparation

- **For Refactoring:** Always ask, *"How can I make this code simpler and more modular?"*
- **For Code Smells:** Look for signs like long methods or duplicated code as indicators for refactoring.
- **For UML:** Ask, *"How does this diagram help me understand the system?"* Use diagrams as communication tools.

- **For Both:** Consistency, simplicity, and clarity are the cornerstones of maintainable software.

Summary: Your ultimate goal is to create software that is not only functional but also easy to maintain, extend, and communicate. Refactoring and UML are two sides of the same coin in achieving high-quality software design.

Memory Hook: Keep this document handy for exam revision and as a reference during your projects.

Chapter 5

Appendix

5.1 Sample UML Diagram Description

Class Diagram Example:
Imagine a system for managing a video store. The primary classes might include:

- `Customer` with attributes like `name` and a list of `Rental`s.
- `Rental` which contains a `Movie`.
- `Movie` which is abstract, with concrete subclasses such as `RegularMovie`, `NewReleaseMovie`, and `ChildrensMovie` implementing a `getCharge()` method.

Such a diagram would show associations (a Customer has many Rentals) and inheritance (Movie is extended by its subclasses).

5.2 Sample Sequence Diagram Description

Sequence Diagram Example:
Consider a use case for processing a customer order in an e-commerce system. The diagram would typically involve:

- An actor (Customer) sending a `placeOrder()` request.
- The `Order` object processing the request and interacting with a `PaymentService` to authorize payment.
- Messages flowing back and forth in a time-ordered fashion, with activation boxes indicating when objects are processing a message.

Chapter 6

Conclusion

In this ultimate study document, we have combined detailed explanations, code examples, and memory hooks to cover every essential aspect of refactoring, code smells, and UML. Whether you are revising for an exam or using this as a reference for your own projects, the key concepts here will guide you toward writing cleaner code and designing better systems.

Final Takeaway: Clear, modular, and well-modeled software is the foundation for robust, scalable, and maintainable systems. Embrace refactoring to improve your code continuously, and use UML to articulate your design ideas with clarity.