LE/CSSD 2101 4.00

Object-Oriented Programming

Lab 2 - Polymorphic Bookstore Management System

Prerequisites: Basic OOP Knowledge + Review of Lab 1

Submission: Source Code Report + 20 Minutes Video Reflection

Learning Objectives:

- Master the four pillars of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction
- Implement and understand SOLID principles in practice
- Design and implement a polymorphic class hierarchy
- Create immutable objects with defensive programming techniques
- Develop unit tests with high coverage
- Apply design patterns including Template Method and Strategy
- Practice interface segregation and dependency inversion
- Understand dynamic binding and method dispatch in Java

Lab Goals: Implement a bookstore management system that demonstrates professional software engineering practices. The system manages various types of materials including books, magazines, audiobooks, and videos through a polymorphic hierarchy, showcasing the power of object-oriented design principles.

Department of Electrical Engineering and Computer Science

York University - Fall 2025

Contents

1 Lab 2 Summary

In this lab, you will work with an advanced bookstore management system that demonstrates professional software engineering practices. The system manages various types of materials—including books, magazines, audiobooks, videos, and newly introduced digital resources—through a polymorphic hierarchy that showcases the power and elegance of object-oriented design principles.

1.1 System Architecture

The bookstore system is architected around several core components. At the foundation lies the **Model Layer**, with an abstract `Material` class and its concrete implementations. The **API Layer** defines the contracts through the `BookstoreAPI` and `MaterialStore` interfaces, while the **Implementation Layer** provides concrete classes such as `BookstoreArrayList` and `MaterialStoreImpl`. Supporting utilities handle array-based operations, and a robust **Test Suite**—driven by JUnit 5—ensures reliability with 70% or higher code coverage. Together, these components establish a platform where students can practice designing, extending, and testing real-world systems.

1.2 Codebase Exploration

The lab begins with a structured exploration of the existing codebase. Students will analyze the abstract and concrete class hierarchy, examine how interfaces shape contracts, and identify examples of defensive programming techniques that safeguard system integrity. A key deliverable in this stage is the generation of UML diagrams using IntelliJ IDEA Ultimate's built-in diagramming tools. These diagrams serve not only as visual documentation but also as a critical aid in understanding inheritance hierarchies, associations, aggregations, and composition relationships within the system.

1.3 Feature Implementation

Building on this foundation, students will extend the system by implementing new features. The most significant enhancement is the introduction of the `EBook` class, which extends `Material` and implements the `Media` interface. This new type introduces properties unique to digital content—file format, file size, DRM status, and word count—and enforces immutability and validation. By integrating the EBook into the polymorphic hierarchy, students experience firsthand how new functionality can be added without disrupting existing code, embodying the Open/Closed Principle.

1.4 Enhanced Search Capabilities

Beyond new material types, students will also enhance the system's search capabilities. By implementing query methods that support temporal filters, multi-creator searches, custom predicates, and comparator-based sorting, they learn how well-designed APIs can future-proof systems. This exercise emphasizes the balance between flexibility and clarity in interface design, while also reinforcing the role of abstraction in creating reusable and extensible software components.

1.5 Design Pattern Integration

To deepen their grasp of design patterns, students will implement the **Visitor Pattern** to calculate shipping costs across different material types. Unlike hardcoded conditionals, the Visitor cleanly separates algorithmic behavior from data structures, ensuring that the system is open to extension while closed to modification. Physical items incur weight-based costs, magazines use flat-rate shipping, and digital resources remain cost-free, offering a realistic and well-structured application of design principles.

1.6 Testing and Quality Assurance

Testing and quality assurance form the backbone of the third phase of this lab. Students will design and run unit tests to validate the correctness of their `EBook` implementation, including creation, validation, discount application, and reading time calculations. They will also explore performance testing by benchmarking search operations across different data structures, such as `ArrayList` versus `HashMap`. These experiments illuminate the trade-offs between time complexity and memory efficiency in practical contexts, helping students develop a performance-aware engineering mindset.

1.7 Design Principles and Reflection

In the final stage, students will explicitly apply **SOLID principles** and additional design patterns. Through written analysis and code-based examples, they will articulate how their implementation embodies single responsibility, open/closed design, and dependency inversion. They will then extend the system further by choosing and implementing one of three advanced design patterns: Factory, Observer, or Decorator. This exercise integrates theory and practice, encouraging students to make informed design decisions that balance clarity, maintainability, and extensibility.

The lab culminates with a **20-minute reflection video**, in which each student demonstrates technical and conceptual mastery. Students must explain polymorphism through the `Material` hierarchy, illustrate dynamic binding, discuss the benefits of immutability, and compare alternative design strategies. Reflection questions further prompt them to consider interface design, defensive programming, performance trade-offs, and the relative value of different design principles. This capstone element ensures that students not only produce a working system but also articulate the reasoning behind their engineering choices—an essential skill for both professional practice and lifelong learning.

1.8 Learning Outcomes

By completing this lab, you will be able to:

- **Understand OOP:** Master the four pillars of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction
- **SOLID Principles:** Implement and understand SOLID principles in practice
- **Design Patterns:** Apply Template Method, Strategy, and Visitor patterns effectively
- **Polymorphic Design:** Design and implement complex polymorphic class hierarchies

- **Testing:** Develop unit tests with high coverage using JUnit 5
- **Performance Analysis:** Analyze and compare different data structure implementations
- **Interface Design:** Practice interface segregation and dependency inversion
- **Dynamic Binding:** Understand method dispatch and runtime polymorphism in Java

Why These Advanced Concepts Matter

Polymorphism allows objects of different types to be treated uniformly through a common interface, enabling flexible and extensible code that can handle new types without modifying existing code.
SOLID Principles provide guidelines for writing maintainable, scalable, and robust object-oriented code. These principles help prevent common design problems and make code easier to understand, test, and modify.
Design Patterns are proven solutions to common software design problems. Understanding and applying patterns like Template Method, Strategy, and Visitor helps create more flexible and maintainable systems.
Interface Segregation ensures that clients are not forced to depend on interfaces they don't use, leading to more cohesive and focused interfaces that are easier to implement and maintain.
Dependency Inversion promotes loose coupling by depending on abstractions rather than concrete implementations, making the system more flexible and testable.
Testing with high coverage ensures code reliability and serves as living documentation. Advanced testing techniques help catch edge cases and prevent regressions.

1.9 OOP Concepts Covered in This Lab

This lab provides hands-on experience with the following foundational and advanced object-oriented programming concepts:

1.9.1 Foundational OOP Concepts

- **Abstraction:** Hiding complex implementation details behind simple interfaces
- **Encapsulation:** Bundling data and methods together while controlling access
- **Inheritance:** Creating new classes based on existing ones to promote code reuse
- **Polymorphism:** Using a single interface to represent different underlying forms

1.9.2 Class Types & Structures

- **Abstract Classes:** Base classes that cannot be instantiated directly
- **Concrete Classes:** Fully implemented classes that can be instantiated
- **Interfaces:** Contracts defining what methods implementing classes must provide

- **POJOs:** Plain Old Java Objects serving as simple data carriers
- **Static Members:** Class-level state and behavior shared across all instances
- **Final Classes/Methods:** Restricting extension and method overriding
- **Inner/Nested Classes:** Classes defined within other classes for encapsulation

1.9.3 Object Lifecycle & Behavior

- **Constructors:** Special methods for object initialization
- **Method Overloading:** Multiple methods with same name but different parameters
- **Method Overriding:** Subclasses providing specific implementations
- **Immutability:** Objects whose state cannot be changed after creation
- **Exception Handling:** Managing and responding to runtime errors

1.9.4 Object Relationships

- **Association:** General relationship between objects
- **Aggregation:** "Has-a" relationship where objects can exist independently
- **Composition:** Strong "part-of" relationship with shared lifecycle
- **Dependency:** One object depends on another for its functionality

1.10 Core Design Quality Attributes

Design Quality Fundamentals

Cohesion: Degree to which elements of a module/class belong together and serve a single, well-defined purpose.

- **High cohesion** → module does one thing well (e.g., UserRepository only handles persistence)
- **Low cohesion** → module mixes unrelated responsibilities (e.g., class handling DB, business logic, and UI)
- High cohesion enhances readability, reusability, and maintainability

Coupling: Degree of interdependence between modules/classes.

- **Low/loose coupling** → components communicate via well-defined interfaces and can be modified independently
- **High/tight coupling** → components are tightly linked; changes in one ripple through others
- Goal: Minimize coupling while maintaining meaningful interactions

Modularity: Decomposition of a system into smaller, self-contained units with clear boundaries.

- Facilitates parallel development, easier debugging, and scalability
- High cohesion and loose coupling are great criteria for good modularity

1.11 SOLID Principles

SOLID Principles in Practice

The SOLID principles formulated by Robert C. Martin are five key object-oriented design principles to improve maintainability and flexibility:

1. **Single Responsibility Principle (SRP)**
   - A class should have one, and only one, reason to change
   - Example: Don't mix logging logic with business logic in the same class
2. **Open/Closed Principle (OCP)**
   - Software entities should be open for extension but closed for modification
   - Achieved by abstraction and polymorphism
   - Example: Instead of editing PaymentProcessor for new payment methods, extend it with new subclasses
3. **Liskov Substitution Principle (LSP)**
   - Subtypes must be substitutable for their base types without breaking functionality
   - Example: If Rectangle is a base class, Square as a subclass should not violate expected behavior
4. **Interface Segregation Principle (ISP)**
   - Clients should not be forced to depend on interfaces they don't use
   - Example: Instead of monolithic IMachine interface with print(), scan(), fax(), split into role-specific interfaces
   - A simple printer should only implement IPrinter, not be forced to implement unused scan() and fax() methods
5. **Dependency Inversion Principle (DIP)**
   - Depend on abstractions, not concretions
   - High-level modules should not depend on low-level modules; both depend on abstractions
   - Example: Service depends on ILogger interface, not on concrete FileLogger implementation

1.12 GRASP Principles

GRASP: Responsibility Assignment Guidelines

General Responsibility Assignment Software Patterns (GRASP) introduced by Craig Larman provide guidelines for assigning responsibilities in OOP design. They're more granular than SOLID, focusing on who does what in a system:

1. **Information Expert:** Assign responsibility to the class with the necessary information
   - Example: Order calculates its own total, since it has access to its line items
2. **Creator:** A class that contains/aggregates/records another should be responsible for creating it
   - Example: Order creates OrderLine
3. **Controller:** A controller object handles system events, coordinating work between UI and domain objects
   - Example: OrderController handles "place order" request
4. **Low Coupling:** Assign responsibilities to reduce dependency between classes
5. **High Cohesion:** Assign responsibilities to keep classes focused and manageable
6. **Polymorphism:** Use polymorphism to handle variations in behavior
   - Example: Different TaxCalculator implementations for different regions
7. **Pure Fabrication:** Create an artificial class when responsibilities don't fit existing domain objects
   - Example: PersistenceManager class that doesn't represent a domain concept but cleanly handles DB logic
8. **Indirection:** Assign responsibility to an intermediate object to mediate between components
   - Example: Service locator or middleware
9. **Protected Variations:** Shield elements from variations in other elements via stable interfaces
   - Example: Define abstraction for payment processing so clients aren't affected if payment providers change

1.13 General Design Principles

Additional Design Principles

Law of Demeter (Principle of Least Knowledge): Objects should only communicate with their immediate friends
Separation of Concerns (SoC): Different aspects of functionality should be handled by different modules
DRY (Don't Repeat Yourself): Avoid code duplication by extracting common functionality
KISS (Keep It Simple, Stupid): Prefer simple solutions over complex ones
YAGNI (You Aren't Gonna Need It): Don't add functionality until it's actually needed
Composition over Inheritance: Favor object composition over class inheritance for code reuse

2 Lab Overview

In this lab, you will work with a slightly more advanced bookstore management system compare to lab 1 that demonstrates professional software engineering practices. The system manages various types of materials including books, magazines, audiobooks, and videos through a polymorphic hierarchy. The goal is that we showcase our gradual understanding of the principles of OOP and how to apply them in a real-world application.

2.1 System Architecture

Our new bookstore system consists of several key components:

Component	Description
Model Layer	Abstract `Material` class along with its concrete implementations
API Layer	`BookstoreAPI` and `MaterialStore` interfaces
Implementation	`BookstoreArrayList` and `MaterialStoreImpl`
Utilities	`BookArrayUtils` for static array operations
Test Suite	many JUnit tests to provide 70% codebase coverage

3 Part 1: Understanding the Codebase (10 points)

3.1 Deliverable: Code Exploration and Analysis (5 points)

You have been provided with the codebase for the bookstore system. Please navigate to the `src` directory and explore the codebase structure, understand it and explain it in your own words.

Deliverable

Answer the following questions in your submission:

1. How many concrete classes extend the abstract `Material` class?
2. What design pattern is demonstrated in the `getDiscountedPrice()` method?
3. List all interfaces implemented in the system and their purposes.
4. Identify three examples of defensive programming in the `Book` class.
5. What is the worst case time complexity of finding a book by ISBN in `BookstoreArrayList`?

3.2 Deliverable: UML Generation and Analysis (5 points)

UML diagrams are a great way to visualize the structure of the codebase. In this lab, we will use the **Diagrams plugin** to generate UML diagrams. As you may know, IntelliJ Ultimate Edition IDE includes the **Diagrams plugin**, which is enabled by default and allow for UML diagrams to be generated:

- To generate a UML class diagram:
  - In the **Project** tool window, right-click on a package, directory, or class.
  - Choose **Diagrams** → **Show Diagram**.
  - Use the shortcut: `Ctrl+Alt+Shift+U` (Windows/Linux) or `Option+Shift+Cmd+U` (macOS).
  - Select **Java Class Diagram**, IntelliJ will render the UML diagram.
- Reference: IntelliJ IDEA Help: Class Diagrams.

Deliverable

1. Draw (or screenshot) the inheritance hierarchy for the `Material` class
2. Identify all aggregation and composition relationships
3. Explain why `Media` is an interface rather than a class

4 Part 2: Implementing New Features (45 points)

4.1 Deliverable: Add a New Material Type (15 points)

Create a new material type called `EBook` that:

- Extends `Material`
- Implements the `Media` interface
- Has properties: author, fileFormat (PDF/EPUB/MOBI), fileSize, DRM-enabled, wordCount
- Provides a 15% discount for DRM-free books
- Estimates reading time based on average reading speed (250 words/minute)
- Validates fileFormat is one of the allowed types
- Ensures fileSize is positive

```
1  public class EBook extends Material implements Media {
2      private final String author;
3      private final String fileFormat; // PDF, EPUB, MOBI
4      private final double fileSize;   // in MB
5      private final boolean drmEnabled;
6      private final int wordCount;
7      private final MediaQuality quality;
8
9      // Constructor with validation
10     // Implement getDiscountRate() - return 0.15 if !drmEnabled
11     // Implement getReadingTimeMinutes() - wordCount / 250
12     // Implement all Media interface methods
13     // Override getDescription() from Material
14 }
```

4.2 Deliverable: Enhanced Search Functionality (20 points)

Add the following search methods to `MaterialStore`:

```
// Find materials published in the last N years
List<Material> findRecentMaterials(int years);

// Find materials by multiple creators (OR condition)
List<Material> findByCreators(String... creators);

// Find materials with custom filtering
List<Material> findWithPredicate(Predicate<Material> condition);

// Get materials sorted by custom comparator
List<Material> getSorted(Comparator<Material> comparator);
```

4.3 Deliverable: Visitor Pattern Implementation (10 points)

Implement the Visitor pattern to calculate shipping costs:

```
public interface MaterialVisitor {
    void visit(PrintedBook book);
    void visit(Magazine magazine);
    void visit(AudioBook audioBook);
    void visit(VideoMaterial video);
    void visit(EBook ebook);
}

public class ShippingCostCalculator implements MaterialVisitor {
    // Physical items: £0.50 per 100g
    // Digital items: £0 (instant download)
    // Magazines: £2 flat rate
}
```

5 Part 3: Testing and Quality Assurance (15 points)

5.1 Deliverable: Enhanced Unit Testing (10 points)

Write unit tests for your `EBook` class:

```
@Test
public void testEBookCreation() {
    EBook ebook = new EBook("978-0134685991", "Effective Java",
        "Joshua Bloch", 45.99, 2018, "EPUB", 2.5,
        false, 90000, MediaQuality.HIGH);
    assertNotNull(ebook);
    assertEquals("Effective Java", ebook.getTitle());
}

@Test
public void testEBookValidation() {
```

```
    // Test invalid file format
    assertThrows(IllegalArgumentException.class, () ->
        new EBook(..., "TXT", ...)); // Should only accept PDF/EPUB/MOBI

    // Test negative file size
    assertThrows(IllegalArgumentException.class, () ->
        new EBook(..., -1.0, ...));
}

@Test
public void testDRMDiscount() {
    EBook drmFree = new EBook(..., false, ...); // DRM disabled
    assertEquals(0.15, drmFree.getDiscountRate(), 0.001);

    EBook withDRM = new EBook(..., true, ...); // DRM enabled
    assertEquals(0.0, withDRM.getDiscountRate(), 0.001);
}

@Test
public void testReadingTimeEstimation() {
    EBook book = new EBook(..., 50000, ...); // 50,000 words
    assertEquals(200, book.getReadingTimeMinutes()); // 50000/250 = 200
}
```

Run tests and generate coverage report:

```
mvn test
mvn jacoco:report
# Open target/site/jacoco/index.html
```

5.2 Deliverable: Exploring Performance Testing Concepts (5 points)

Create a performance test that:

1. Adds 10,000 materials to the store
2. Measures search performance for different operations
3. Compares ArrayList vs HashMap implementations
4. Documents your findings with execution times

```
@Test
public void performanceComparison() {
    MaterialStore arrayStore = new MaterialStoreImpl();
    // HashMap implementation (if created)

    long startTime = System.nanoTime();
    for (int i = 0; i < 10000; i++) {
        arrayStore.addMaterial(createRandomMaterial(i));
    }
    long addTime = System.nanoTime() - startTime;

    startTime = System.nanoTime();
```

```
13      Material found = arrayStore.findById("ID-5000");
14      long searchTime = System.nanoTime() - startTime;
15
16      System.out.println("ArrayList Add 10K: " + addTime/1_000_000 + " ms");
17      System.out.println("ArrayList Search: " + searchTime/1_000_000 + " ms");
18  }
```

6 Part 4: Design Patterns and Principles (30 points)

6.1 Deliverable: SOLID Principles Analysis (10 points)

For each SOLID principle, provide:

1. **S**ingle Responsibility: One example from the codebase
2. **O**pen/Closed: How the Material hierarchy demonstrates this
3. **L**iskov Substitution: An example of proper substitution
4. **I**nterface Segregation: Why Media is separate from Material
5. **D**ependency Inversion: How the API layer demonstrates this

6.2 Deliverable: Design Pattern Implementation (20 points)

Read the following design pattern concepts and explain them in your own words, select one of the following patterns to implement:

Design Pattern Concepts Explained

6.2.1 Factory Pattern

The **Factory Pattern** is a creational design pattern that provides an interface for creating objects without specifying their exact class. In the context of the bookstore system, the `MaterialFactory` class serves as a centralized creation point for different types of materials (books, magazines, audiobooks, videos, ebooks). This pattern encapsulates the object creation logic and provides a uniform interface for creating various material types. The factory receives a type parameter and a map of properties, then returns the appropriate material instance based on the type. This approach promotes loose coupling by removing the need for client code to know the specific concrete classes being instantiated. The factory pattern is particularly valuable when you have a complex object creation process, when you want to centralize object creation logic, or when you need to support multiple product families. In the bookstore context, it allows the system to easily add new material types without modifying existing client code, following the Open/Closed Principle of SOLID design principles.

6.2.2 Observer Pattern

The **Observer Pattern** is a behavioral design pattern that defines a one-to-many dependency between objects, where when one object changes state, all its dependents are automatically notified and updated. In the bookstore system, the `InventoryObserver` interface defines the contract for objects that need to be notified about inventory changes. The pattern allows multiple observers (such as inventory managers, reorder systems, or analytics modules) to subscribe to inventory events without the inventory system needing to know about their specific implementations. When materials are added, removed, or have their prices changed, all registered observers are automatically notified through the defined interface methods. This pattern promotes loose coupling between the subject (inventory system) and observers, making the system more flexible and extensible. The observer pattern is particularly useful in scenarios where you need to maintain consistency between related objects, implement event-driven architectures, or support multiple views of the same data. It follows the Dependency Inversion Principle by depending on abstractions (the observer interface) rather than concrete implementations.

6.2.3 Decorator Pattern

The **Decorator Pattern** is a structural design pattern that allows behavior to be added to objects dynamically without altering their structure. In the bookstore system, the `MaterialDecorator` abstract class extends the base `Material` class and contains a reference to a `Material` object that it decorates. This pattern enables the addition of features like gift wrapping, express shipping, special packaging, or premium handling to existing materials without modifying their original classes. Each decorator wraps the original material and adds its own functionality while maintaining the same interface as the base material. This creates a flexible alternative to inheritance, allowing you to mix and match different features at runtime. For example, you could have a book with gift wrapping and express shipping by chaining multiple decorators together. The decorator pattern is particularly valuable when you need to add responsibilities to objects dynamically and transparently, when extension by subclassing is impractical, or when you want to avoid feature explosion in the class hierarchy. It follows the Open/Closed Principle by being open for extension (new decorators) but closed for modification (existing classes remain unchanged).

6.2.4 Pattern to Implement

Choose ONE of the following patterns to implement:

Option A: Factory Pattern

```
1  public class MaterialFactory {
2      public static Material createMaterial(String type, Map<String, Object> properties) {
3          // Implementation
4      }
5  }
```

Option B: Observer Pattern

```
1  public interface InventoryObserver {
2      void onMaterialAdded(Material material);
```

```
    void onMaterialRemoved(Material material);
    void onPriceChanged(Material material, double oldPrice, double newPrice);
}
```

Option C: Decorator Pattern

```
public abstract class MaterialDecorator extends Material {
    protected Material decoratedMaterial;
    // Add gift wrapping, express shipping, etc.
}
```

7 Video Reflection Requirements (20 minutes)

Important Note

Create a 20-minute video reflection in your own words explaining the codes and how they work together. You should address ALL of the following points:

7.1 Technical Discussion

1. Explain polymorphism using examples from the Material hierarchy
2. Demonstrate dynamic binding with a code walkthrough
3. Show how the Template Method pattern works in `Material.getDiscountedPrice()`
4. Explain the benefits of immutability in the domain models
5. Discuss the trade-offs between ArrayList and HashMap implementations

7.2 Reflection Questions

Answer these reflection questions thoughtfully:

1. **Abstraction Understanding**: How does the abstract `Material` class enforce a contract for its subclasses? What would happen if we made `Material` a concrete class instead?
2. **Polymorphism in Practice**: Describe a real-world scenario where you would use polymorphism similar to this bookstore system. How would it improve code maintainability?
3. **Interface Design**: Why is the `Media` interface valuable even though `AudioBook` and `VideoMaterial` already extend `Material`? What principle does this demonstrate?
4. **Defensive Programming**: Identify three defensive programming techniques used in the codebase. How do they prevent bugs and improve reliability?
5. **Testing Strategy**: Why is it important to test both valid and invalid inputs? Give an example of a boundary condition test from the codebase.

6. **Design Patterns**: Which design pattern from the lab do you find most useful? How would you apply it in your own projects?
7. **Performance Considerations**: What are the performance implications of using `ArrayList` vs `HashMap` for the bookstore? When would you choose each?
8. **SOLID Principles**: Which SOLID principle do you think is most important for maintainable code? Provide an example from your implementation.
9. **Code Quality**: What makes code "clean"? Identify three characteristics of clean code demonstrated in this lab.
10. **Learning Reflection**: What was the most challenging concept in this lab? How did you overcome the challenge, and what resources did you use?

8 Submission Checklist

☐ **Compilation Success:** All Java source files compile without errors. Please ensure your code compiles successfully to avoid any grading issues. Lab submissions with compilation errors will not be accepted.

☐ **Test Suite:** Enhanced test cases that pass (`mvn test` shows BUILD SUCCESS) with no failures or errors. Please ensure your test suite runs completely to avoid any grading issues. Lab submissions with failing tests will not be accepted.

☐ **Code Coverage:** Achieve $> 70\%$ code coverage (`mvn jacoco:report`). This demonstrates thorough testing of your implementation.

☐ **Documentation:** Javadoc generated without warnings. You can use your IDE to generate Javadoc or use `mvn javadoc:javadoc`. Ensure all public methods are properly documented.

☐ **UML Diagrams:** UML diagrams generated without errors. You can use your IDE to generate UML diagrams. Please make sure to throughly explain the details and symbols used in the diagrams.

☐ **Reflection:** README.md contains thoughtful answers to all reflection questions. Please refer to Section 7.2 for the exact questions to address.

☐ **Repository:** Submit a zip file of your entire Git repository, properly organized with a .gitignore file. Include all source code, tests, and configuration files.

☐ **Video Presentation:** Record a clear video with all team members visible and actively participating. Each member should explain their contributions and demonstrate understanding of the codebase.

☐ **Portability:** Ensure no hardcoded paths or system-specific code. Your code should be portable and run on any system with the required Java and Maven versions.

☐ **Code Quality:** Maintain consistent code formatting throughout the project. Use your IDE's auto-formatting features to ensure uniformity.

☐ **Clean Compilation:** Minimize compiler warnings (unused imports, raw types, etc.). While some warnings may be acceptable, strive for clean compilation output.

9 Additional Learning Resources

- **Installing and Learning Java:**
  - Official Java SE Downloads and Installation Guide: https://www.oracle.com/java/technologies/javase-downloads.html
  - Java Tutorials (Oracle): https://docs.oracle.com/javase/tutorial/
  - Java Collections Framework Reference: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html
- **Maven Build Tool:**
  - Maven Installation Guide: https://maven.apache.org/install.html
  - Maven Getting Started Guide: https://maven.apache.org/guides/getting-started/
  - Maven In-Depth Guides: https://maven.apache.org/guides/
- **Using IntelliJ IDEA:**
  - IntelliJ IDEA Download and Setup: https://www.jetbrains.com/idea/download/
  - IntelliJ IDEA Documentation: https://www.jetbrains.com/help/idea/
  - Quick Start with Maven in IntelliJ: https://www.jetbrains.com/help/idea/maven-support.html
- Oracle: *Java Code Conventions - Naming Conventions* https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
- Oracle Java Tutorials: *Naming a Package* https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html
- **Testing with JUnit:**
  - JUnit 5 User Guide: https://junit.org/junit5/docs/current/user-guide/
  - JUnit 5 API Reference: https://junit.org/junit5/docs/current/api/
- **Java Documentation (Javadoc):**
  - Official Javadoc Tool Guide: https://docs.oracle.com/en/java/javase/17/docs/specs/javadoc/doc-comment-spec.html
  - Generating Javadoc with Maven: https://maven.apache.org/plugins/maven-javadoc-plugin/
  - Best Practices for Javadoc: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Good luck, and enjoy exploring OOP concepts!

10 Sample Solution

This section provides detailed answers to all questions and tasks from Lab 2, with references to the actual source code implementation.

10.1 Part 1: Understanding the Codebase - Solutions

10.1.1 Code Exploration and Analysis

1. **How many concrete classes extend the abstract `Material` class?**

   Based on the codebase analysis, there are **5 concrete classes** that extend the abstract `Material` class:

   - `PrintedBook` - extends `Material`
   - `Magazine` - extends `Material`
   - `AudioBook` - extends `Material`
   - `VideoMaterial` - extends `Material`
   - `EBook` - extends `Material`

2. **What design pattern is demonstrated in the `getDiscountedPrice()` method?**

   The `getDiscountedPrice()` method demonstrates the **Template Method Pattern**. Here's the implementation:

```java
// Template method for calculating discounted price from Material.java
public final double getDiscountedPrice() {
    return price * (1.0 - getDiscountRate());
}

// Hook method for customization
public double getDiscountRate() {
    return 0.0;
}
```

   The Template Method Pattern defines the skeleton of an algorithm in a base class, allowing subclasses to override specific steps without changing the algorithm's structure. The `getDiscountedPrice()` method provides the template (final method), while `getDiscountRate()` serves as a hook that subclasses can override.

3. **List all interfaces implemented in the system and their purposes.**

   The system implements several interfaces:

   - `Comparable<Material>` - Enables natural ordering of materials by title
   - `Media` - Defines contract for media materials (audio/video) with duration, format, quality
   - `MaterialVisitor` - Visitor pattern interface for adding operations to material hierarchy
   - `BookstoreAPI` - Defines bookstore operations (CRUD, search, analytics)
   - `MaterialStore` - Defines polymorphic material store operations

4. **Identify three examples of defensive programming in the `Book` class.**

   Three examples of defensive programming in the `Book` class:

```java
// 1. Input validation in constructor
public Book(String isbn, String title, String author, double price, int year) {
    this.isbn = validateIsbn(isbn);
    this.title = validateStringField(title, "Title");
    this.author = validateStringField(author, "Author");
    this.price = validatePrice(price);
    this.year = validateYear(year);
}

// 2. ISBN validation with checks
private String validateIsbn(String isbn) {
    if (isbn == null) {
        throw new NullPointerException("ISBN cannot be null");
    }
    String cleaned = isbn.replaceAll("[^0-9X]", "");
    if (!isValidIsbn(cleaned)) {
        throw new IllegalArgumentException("Invalid ISBN format: " + isbn);
    }
    return cleaned;
}

// 3. Price validation with NaN and infinity checks
private double validatePrice(double price) {
    if (price < 0.0) {
        throw new IllegalArgumentException("Price cannot be negative: " + price);
    }
    if (Double.isNaN(price) || Double.isInfinite(price)) {
        throw new IllegalArgumentException("Price must be a valid number: " + price);
    }
    return price;
}
```

5. **What is the worst case time complexity of finding a book by ISBN in `BookstoreArrayList`?**

   The worst case time complexity is **O(n)** where n is the number of books in the inventory. This is because `BookstoreArrayList` uses a linear search through the ArrayList:

```java
@Override
public Book findByIsbn(String isbn) {
    if (isbn == null || isbn.trim().isEmpty()) {
        return null;
    }

    for (Book book : inventory) {  // O(n) linear search
        if (book.getIsbn().equals(isbn)) {
            return book;
        }
    }
    return null;
}
```

10.1.2 UML Generation and Analysis

1. **Draw the inheritance hierarchy for the `Material` class**

   The inheritance hierarchy shows:

   - `Material` (abstract base class)
   - `PrintedBook` extends `Material`
   - `Magazine` extends `Material`
   - `AudioBook` extends `Material` implements `Media`
   - `VideoMaterial` extends `Material` implements `Media`
   - `EBook` extends `Material` implements `Media`

2. **Identify all aggregation and composition relationships**

   - **Composition**: `MaterialStoreImpl` contains `List<Material>` - materials cannot exist without the store
   - **Aggregation**: `BookstoreArrayList` contains `List<Book>` - books can exist independently
   - **Association**: `MaterialVisitor` visits `Material` objects

3. **Explain why `Media` is an interface rather than a class**

   `Media` is an interface because:

   - It defines a contract for media-specific behavior (duration, format, quality)
   - Multiple inheritance - classes can extend `Material` AND implement `Media`
   - Interface Segregation Principle - separates media concerns from material concerns
   - Flexibility - allows different media types to have different implementations

10.2 Part 2: Implementing New Features - Solutions

10.2.1 EBook Implementation

The `EBook` class implementation demonstrates polymorphism, immutability, and defensive programming:

```
1  public class EBook extends Material implements Media {
2      private static final Set<String> VALID_FORMATS = Set.of("PDF", "EPUB", "MOBI");
3      private static final int WORDS_PER_MINUTE = 250;
4      private static final double DRM_FREE_DISCOUNT = 0.15;
5
6      private final String author;
7      private final String fileFormat;
8      private final double fileSize;
9      private final boolean drmEnabled;
10     private final int wordCount;
11     private final MediaQuality quality;
12
```

```
    public EBook(String id, String title, String author, double price, int year,
                 String fileFormat, double fileSize, boolean drmEnabled,
                 int wordCount, MediaQuality quality) {
        super(id, title, price, year, MaterialType.E_BOOK);
        this.author = validateStringField(author, "Author");
        this.fileFormat = validateFileFormat(fileFormat);
        this.fileSize = validateFileSize(fileSize);
        this.drmEnabled = drmEnabled;
        this.wordCount = validateWordCount(wordCount);
        this.quality = Objects.requireNonNull(quality, "Media quality cannot be null");
    }

    // Polymorphic discount calculation
    @Override
    public double getDiscountRate() {
        return drmEnabled ? 0.0 : DRM_FREE_DISCOUNT;
    }

    // Reading time estimation
    public int getReadingTimeMinutes() {
        return wordCount / WORDS_PER_MINUTE;
    }

    // Media interface implementation
    @Override
    public int getDuration() {
        return getReadingTimeMinutes();
    }

    @Override
    public String getFormat() {
        return fileFormat;
    }

    @Override
    public boolean isStreamingOnly() {
        return false; // E-books are downloadable
    }
}
```

10.2.2 Enhanced Search Functionality

The enhanced search methods demonstrate functional programming and polymorphism:

```
// Implementation in MaterialStoreImpl.java
@Override
public List<Material> findRecentMaterials(int years) {
    if (years < 0) {
        throw new IllegalArgumentException("Years cannot be negative: " + years);
    }

    int currentYear = java.time.Year.now().getValue();
    int cutoffYear = currentYear - years;

    return inventory.stream()
        .filter(material -> material.getYear() >= cutoffYear)
        .collect(Collectors.toList());
```

```
}

@Override
public List<Material> findByCreators(String... creators) {
    if (creators == null || creators.length == 0) {
        return new ArrayList<>();
    }

    Set<String> creatorSet = Arrays.stream(creators)
        .filter(Objects::nonNull)
        .map(String::trim)
        .filter(s -> !s.isEmpty())
        .collect(Collectors.toSet());

    return inventory.stream()
        .filter(material -> creatorSet.contains(material.getCreator()))
        .collect(Collectors.toList());
}

@Override
public List<Material> findWithPredicate(Predicate<Material> condition) {
    if (condition == null) {
        throw new NullPointerException("Predicate cannot be null");
    }

    return inventory.stream()
        .filter(condition)
        .collect(Collectors.toList());
}

@Override
public List<Material> getSorted(Comparator<Material> comparator) {
    if (comparator == null) {
        throw new NullPointerException("Comparator cannot be null");
    }

    return inventory.stream()
        .sorted(comparator)
        .collect(Collectors.toList());
}
```

10.2.3 Visitor Pattern Implementation

The Visitor pattern implementation for shipping cost calculation:

```
// Visitor interface
public interface MaterialVisitor {
    void visit(PrintedBook book);
    void visit(Magazine magazine);
    void visit(AudioBook audioBook);
    void visit(VideoMaterial video);
    void visit(EBook ebook);
}

// Concrete visitor implementation
public class ShippingCostCalculator implements MaterialVisitor {
    private static final double PHYSICAL_ITEM_RATE = 0.50; // per 100g
```

```java
    private static final double MAGAZINE_FLAT_RATE = 2.00;
    private static final double DIGITAL_ITEM_RATE = 0.00;

    private double totalShippingCost = 0.0;

    @Override
    public void visit(EBook ebook) {
        // E-books are always digital - no shipping cost
        totalShippingCost += DIGITAL_ITEM_RATE;
    }

    @Override
    public void visit(PrintedBook book) {
        // Assume average book weight of 500g
        double weightInHundredGrams = 5.0;
        double cost = weightInHundredGrams * PHYSICAL_ITEM_RATE;
        totalShippingCost += cost;
    }

    @Override
    public void visit(Magazine magazine) {
        totalShippingCost += MAGAZINE_FLAT_RATE;
    }

    @Override
    public void visit(AudioBook audioBook) {
        if (audioBook.getQuality() == MediaQuality.PHYSICAL) {
            // Physical CD - assume 100g
            totalShippingCost += 1.0 * PHYSICAL_ITEM_RATE;
        } else {
            // Digital download
            totalShippingCost += DIGITAL_ITEM_RATE;
        }
    }

    @Override
    public void visit(VideoMaterial video) {
        if (video.getQuality() == MediaQuality.PHYSICAL) {
            // Physical DVD - assume 150g
            totalShippingCost += 1.5 * PHYSICAL_ITEM_RATE;
        } else {
            // Digital download
            totalShippingCost += DIGITAL_ITEM_RATE;
        }
    }

    public double getTotalShippingCost() {
        return totalShippingCost;
    }

    public void reset() {
        totalShippingCost = 0.0;
    }
}
```

10.3 Part 3: Testing and Quality Assurance - Solutions

10.3.1 Unit Testing

Cover unit tests for the `EBook` class:

```java
@Test
void testEBookCreation() {
    assertNotNull(validEBook);
    assertEquals("Effective Java", validEBook.getTitle());
    assertEquals("Joshua Bloch", validEBook.getCreator());
    assertEquals("EPUB", validEBook.getFileFormat());
    assertEquals(2.5, validEBook.getFileSize(), 0.001);
    assertTrue(validEBook.isDrmEnabled());
    assertEquals(90000, validEBook.getWordCount());
    assertEquals(MediaQuality.HIGH, validEBook.getQuality());
}

@Test
void testEBookValidation() {
    // Test invalid file format
    assertThrows(IllegalArgumentException.class, () ->
        new EBook("978-0134685991", "Test Book", "Test Author",
                45.99, 2018, "TXT", 2.5, false, 50000, MediaQuality.HIGH));

    // Test negative file size
    assertThrows(IllegalArgumentException.class, () ->
        new EBook("978-0134685991", "Test Book", "Test Author",
                45.99, 2018, "PDF", -1.0, false, 50000, MediaQuality.HIGH));

    // Test negative word count
    assertThrows(IllegalArgumentException.class, () ->
        new EBook("978-0134685991", "Test Book", "Test Author",
                45.99, 2018, "PDF", 2.5, false, -100, MediaQuality.HIGH));
}

@Test
void testDRMDiscount() {
    // DRM-free book should get 15% discount
    assertEquals(0.15, drmFreeEBook.getDiscountRate(), 0.001);
    assertEquals(39.99 * 0.85, drmFreeEBook.getDiscountedPrice(), 0.01);

    // DRM-enabled book should get no discount
    assertEquals(0.0, validEBook.getDiscountRate(), 0.001);
    assertEquals(45.99, validEBook.getDiscountedPrice(), 0.01);
}

@Test
void testReadingTimeEstimation() {
    // 50,000 words / 250 words per minute = 200 minutes
    assertEquals(200, drmFreeEBook.getReadingTimeMinutes());

    // 90,000 words / 250 words per minute = 360 minutes
    assertEquals(360, validEBook.getReadingTimeMinutes());
}
```

10.3.2 Performance Testing

Performance comparison between ArrayList and HashMap implementations:

```java
@Test
public void performanceComparison() {
    MaterialStore arrayStore = new MaterialStoreImpl();

    // Test adding 10,000 materials
    long startTime = System.nanoTime();
    for (int i = 0; i < 10000; i++) {
        arrayStore.addMaterial(createRandomMaterial(i));
    }
    long addTime = System.nanoTime() - startTime;

    // Test searching for a material
    startTime = System.nanoTime();
    Optional<Material> found = arrayStore.findById("ID-5000");
    long searchTime = System.nanoTime() - startTime;

    System.out.println("ArrayList Add 10K: " + addTime/1_000_000 + " ms");
    System.out.println("ArrayList Search: " + searchTime/1_000_000 + " ms");

    // Results show:
    // ArrayList Add 10K: ~50 ms
    // ArrayList Search: ~0.1 ms (O(n) but small dataset)
    // HashMap would be O(1) for search but O(n) for add
}
```

10.4 Part 4: Design Patterns and Principles - Solutions

10.4.1 SOLID Principles Analysis

1. **Single Responsibility Principle (SRP)**

   Example: `BookArrayUtils` class has a single responsibility - providing utility methods for array operations on Book objects.

2. **Open/Closed Principle (OCP)**

   The Material hierarchy demonstrates OCP by being open for extension (new material types like EBook) but closed for modification (existing Material class doesn't change).

3. **Liskov Substitution Principle (LSP)**

   Any Material subclass can be substituted for Material without breaking functionality. For example, EBook can be used wherever Material is expected.

4. **Interface Segregation Principle (ISP)**

   Media interface is separate from Material because not all materials are media. This prevents classes from depending on methods they don't use.

5. **Dependency Inversion Principle (DIP)**

   The API layer (MaterialStore interface) depends on abstractions (Material), not concrete implementations, allowing for flexible implementations.

10.4.2 Design Pattern Implementation

Factory Pattern Implementation:

```java
public class MaterialFactory {

    public static Material createMaterial(String type, Map<String, Object> properties) {
        if (type == null) {
            throw new NullPointerException("Material type cannot be null");
        }
        if (properties == null) {
            throw new NullPointerException("Properties cannot be null");
        }

        String normalizedType = type.trim().toUpperCase();

        switch (normalizedType) {
            case "BOOK":
            case "PRINTED_BOOK":
                return createPrintedBook(properties);
            case "MAGAZINE":
                return createMagazine(properties);
            case "AUDIO_BOOK":
            case "AUDIOBOOK":
                return createAudioBook(properties);
            case "VIDEO":
            case "VIDEO_MATERIAL":
                return createVideoMaterial(properties);
            case "EBOOK":
            case "E_BOOK":
                return createEBook(properties);
            default:
                throw new IllegalArgumentException("Unsupported material type: " + type);
        }
    }

    private static EBook createEBook(Map<String, Object> properties) {
        String id = getRequiredString(properties, "id");
        String title = getRequiredString(properties, "title");
        String author = getRequiredString(properties, "author");
        double price = getRequiredDouble(properties, "price");
        int year = getRequiredInteger(properties, "year");
        String fileFormat = getRequiredString(properties, "fileFormat");
        double fileSize = getRequiredDouble(properties, "fileSize");
        boolean drmEnabled = getRequiredBoolean(properties, "drmEnabled");
        int wordCount = getRequiredInteger(properties, "wordCount");
        Media.MediaQuality quality = getRequiredMediaQuality(properties, "quality");

        return new EBook(id, title, author, price, year, fileFormat, fileSize,
                         drmEnabled, wordCount, quality);
    }
}
```

10.5 Polymorphism Demonstration

The system demonstrates polymorphism through the `PolymorphismDemo` class:

```
private static void demonstratePolymorphicBehavior(MaterialStore store) {
    System.out.println("1. POLYMORPHIC BEHAVIOR");
    System.out.println("------------------------");

    List<Material> materials = store.getAllMaterials();

    for (Material material : materials) {
        System.out.println("\nMaterial Type: " + material.getType().getDisplayName());
        System.out.println("Title: " + material.getTitle());
        System.out.println("Creator: " + material.getCreator());
        System.out.println("Display Info: " + material.getDisplayInfo());
        System.out.println("Original Price: $" + String.format("%.2f", material.getPrice()));
        System.out.println("Discount Rate: " + (material.getDiscountRate() * 100) + "%");
        System.out.println("Discounted Price: $" + String.format("%.2f",
        ↪  material.getDiscountedPrice()));
    }
}
```

10.6 Defensive Programming Examples

The system implements defensive programming:

```
// Input validation in Material base class
protected String validateId(String id) {
    if (id == null) {
        throw new NullPointerException("ID cannot be null");
    }
    if (id.trim().isEmpty()) {
        throw new IllegalArgumentException("ID cannot be blank");
    }
    return id.trim();
}

protected double validatePrice(double price) {
    if (price < 0.0) {
        throw new IllegalArgumentException(
            "Price cannot be negative. Provided: " + price);
    }
    if (Double.isNaN(price) || Double.isInfinite(price)) {
        throw new IllegalArgumentException(
            "Price must be a valid number. Provided: " + price);
    }
    return price;
}

// EBook-specific validation
private String validateFileFormat(String format) {
    if (format == null) {
        throw new NullPointerException("File format cannot be null");
    }
    String upperFormat = format.trim().toUpperCase();
    if (!VALID_FORMATS.contains(upperFormat)) {
        throw new IllegalArgumentException(
            String.format("Unsupported file format: %s. Supported formats: %s",
                format, VALID_FORMATS));
    }
```

```
    return upperFormat;
}
```

10.7 Immutability Implementation

The system implements immutability through final fields and no setters:

```
// Immutable EBook class design
public class EBook extends Material implements Media {
    // All fields are final - cannot be changed after construction
    private final String author;
    private final String fileFormat;
    private final double fileSize;
    private final boolean drmEnabled;
    private final int wordCount;
    private final MediaQuality quality;

    // Constructor validates and sets all fields
    public EBook(String id, String title, String author, double price, int year,
                 String fileFormat, double fileSize, boolean drmEnabled,
                 int wordCount, MediaQuality quality) {
        super(id, title, price, year, MaterialType.E_BOOK);
        this.author = validateStringField(author, "Author");
        this.fileFormat = validateFileFormat(fileFormat);
        this.fileSize = validateFileSize(fileSize);
        this.drmEnabled = drmEnabled;
        this.wordCount = validateWordCount(wordCount);
        this.quality = Objects.requireNonNull(quality, "Media quality cannot be null");
    }

    // Only getter methods - no setters
    public String getFileFormat() { return fileFormat; }
    public double getFileSize() { return fileSize; }
    public boolean isDrmEnabled() { return drmEnabled; }
    public int getWordCount() { return wordCount; }
}
```

10.8 Recap

We used our polymorphic bookstore management system to demonstrate:

1. **Polymorphism**: Through inheritance hierarchy and interface implementation
2. **Abstraction**: Using abstract base classes and interfaces
3. **Design Patterns**: Factory, Visitor, and Template Method patterns
4. **Defensive Programming**: Detailed input validation and error handling
5. **Immutability**: Immutable object design with defensive copying
6. **Detailed Testing**: Unit tests covering all functionality
7. **SOLID Principles**: Single responsibility, open/closed, and dependency inversion

8. **Performance**: Efficient data structures and algorithms