Exercise 10: Designing for Reuse

Due: December 4, 2025

📋 Assignment Overview

Points: 22 points (20 technical + 2 reflection) (22/20 possible)
Estimated Time: 2-2.5 hours
Submission: Submit via D2L as a single markdown file

🎯 DOWNLOAD THE TEMPLATE: Use `Exercise_10_template.md` **as your starting point. It includes all sections with helpful hints and example code blocks. Rename this to `Exercise_10_FamilyName_GivenName.md**

Learning Objectives:

- Analyze code for SOLID principle violations
- Refactor conditional code to use polymorphism
- Design a generic, reusable component
- Choose composition vs. inheritance appropriately
- Develop metacognitive awareness of your learning process

📝 Instructions

This assignment has **five parts** that directly reinforce the concepts from today's lecture. Complete all parts and submit your work as specified below.

Use the provided template file (`Exercise_10_template.md`) to structure your submission. It includes helpful guidance and code block examples.

The reflection is required. Submissions without the reflection will receive 0 points.

Part 1: Analyzing Code for SOLID Violations (5 points)

Instructions

Examine the following code and identify which SOLID principles are violated. For each violation, explain:

1. **Which principle is violated** (SRP, OCP, LSP, ISP, or DIP)
2. **Why it's a violation** (be specific)
3. **How you would fix it** (brief description)

Code to Analyze

```
public class OrderProcessor {
  private String dbConnectionString =
"jdbc:mysql://localhost:3306/orders";

  public void processOrder(Order order) {
    // Validate order
    if (order.getItems().isEmpty()) {
      System.out.println("Error: Order has no items");
      return;
    }
    if (order.getCustomer() == null) {
      System.out.println("Error: Order has no customer");
      return;
    }

    // Calculate total
    double total = 0;
    for (Item item : order.getItems()) {
      if (item.getType().equals("BOOK")) {
        total += item.getPrice() * 0.9; // 10% discount on books
      } else if (item.getType().equals("ELECTRONICS")) {
        total += item.getPrice() * 0.95; // 5% discount on
electronics
      } else if (item.getType().equals("CLOTHING")) {
        total += item.getPrice(); // No discount
      } else if (item.getType().equals("FOOD")) {
        total += item.getPrice() * 0.85; // 15% discount on food
```

```
      }
    }
    order.setTotal(total);

    // Process payment
    if (order.getPaymentMethod().equals("CREDIT_CARD")) {
      // Credit card processing
      System.out.println("Processing credit card payment");
    } else if (order.getPaymentMethod().equals("PAYPAL")) {
      // PayPal processing
      System.out.println("Processing PayPal payment");
    } else if (order.getPaymentMethod().equals("BITCOIN")) {
      // Bitcoin processing
      System.out.println("Processing Bitcoin payment");
    }

    // Save to database
    try {
      Connection conn =
DriverManager.getConnection(dbConnectionString);
      String sql = "INSERT INTO orders VALUES (?, ?, ?)";
      PreparedStatement stmt = conn.prepareStatement(sql);
      stmt.setInt(1, order.getId());
      stmt.setString(2, order.getCustomer().getName());
      stmt.setDouble(3, total);
      stmt.executeUpdate();
      conn.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }

    // Send confirmation email
    String emailBody = "Dear " + order.getCustomer().getName() +
",\n\n";
    emailBody += "Your order #" + order.getId() + " has been
processed.\n";
    emailBody += "Total: $" + total + "\n\n";
    emailBody += "Thank you for your business!";
    // Email sending logic here
    System.out.println("Sending email: " + emailBody);
```

```
    // Generate PDF invoice
    System.out.println("Generating PDF invoice for order #" +
order.getId());

    // Update inventory
    for (Item item : order.getItems()) {
      System.out.println("Reducing inventory for item: " +
item.getName());
    }
  }
}
```

You should identify at least 3 violations. There may be more!

Part 2: Refactoring Conditional Code to Use Polymorphism (6 points)

Instructions

Refactor the pricing logic from Part 1 to use polymorphism instead of conditionals. The original code uses type-checking to apply different discounts.

Requirements

1. Create an appropriate inheritance hierarchy or interface structure
2. Eliminate the if/else chain for pricing
3. Use proper OO design principles (especially OCP)

Original Code to Refactor

```
// Calculate total with type-based discounts
double total = 0;
for (Item item : order.getItems()) {
  if (item.getType().equals("BOOK")) {
    total += item.getPrice() * 0.9; // 10% discount on books
  } else if (item.getType().equals("ELECTRONICS")) {
    total += item.getPrice() * 0.95; // 5% discount on electronics
```

```
    } else if (item.getType().equals("CLOTHING")) {
      total += item.getPrice(); // No discount
    } else if (item.getType().equals("FOOD")) {
      total += item.getPrice() * 0.85; // 15% discount on food
    }
  }
```

Deliverables

Create the following classes/interfaces in your solution:

1. **Base class or interface** - Define the abstraction
2. **At least 3 concrete subclasses** - Book, Electronics, and one more of your choice
3. **Client code** - Show how to use your new design to calculate totals

---

Part 3: Designing a Generic, Reusable Component (5 points)

Instructions

Design and implement a generic `Cache<K, V>` class that can store key-value pairs of any type with the following functionality:

Requirements

1. **Generic type parameters:** Use `<K, V>` where K is the key type and V is the value type
2. **Methods to implement:**
   - `void put(K key, V value)` - Store a value with a key
   - `V get(K key)` - Retrieve a value by key (return null if not found)
   - `boolean contains(K key)` - Check if a key exists
   - `void remove(K key)` - Remove a key-value pair
   - `int size()` - Return number of items in cache
   - `void clear()` - Remove all items
3. **Capacity limit:** The cache should have a maximum capacity (passed to constructor). When full, the cache should remove the **oldest** item before adding a new one (FIFO - First In, First Out)

4. **Type safety:** Your implementation must be type-safe (no unchecked warnings)

Example Usage

Your cache should work like this:

```
// Cache of strings to integers (e.g., username to user ID)
Cache<String, Integer> userCache = new Cache<>(3); // capacity of 3
userCache.put("alice", 101);
userCache.put("bob", 102);
userCache.put("charlie", 103);

System.out.println(userCache.get("alice")); // 101
System.out.println(userCache.size()); // 3

userCache.put("diana", 104); // This removes "alice" (oldest)
System.out.println(userCache.contains("alice")); // false
System.out.println(userCache.get("diana")); // 104

// Cache of integers to products
Cache<Integer, Product> productCache = new Cache<>(100);
productCache.put(12345, new Product("Laptop", 999.99));
Product p = productCache.get(12345);
```

Deliverables

1. **Cache.java** - Your complete implementation with JavaDoc comments
2. **CacheDemo.java** - A demo class showing at least 3 different uses of your cache with different types

Grading Criteria

- **2 points:** Correct generic implementation with type parameters
- **2 points:** All methods work correctly, including FIFO eviction
- **1 point:** Demo showing reusability with multiple types

Hints

- Make sure to handle edge cases (empty cache, capacity of 0, null keys)

Part 4: Choosing Composition vs. Inheritance (4 points)

Instructions

For each scenario below, decide whether to use **composition** or **inheritance** and justify your choice. Your justification should reference specific principles we learned (is-a vs. has-a, flexibility, substitutability, etc.).

Scenarios

Scenario 1: Restaurant Ordering System

You're building a system for a restaurant chain. Different restaurants have different operational models:

- **Fine Dining:** Requires reservations, has dress code validation, multi-course meal sequencing, sommelier wine pairing
- **Fast Casual:** Walk-in only, simple ordering, quick preparation tracking, loyalty points
- **Food Truck:** GPS location tracking, limited menu that changes daily, mobile payment only, weather-based hours

All restaurants need to:

- Take orders
- Process payments
- Track inventory
- Generate receipts

You're considering:

- **Inheritance:** Create `Restaurant` base class with `FineDiningRestaurant`, `FastCasualRestaurant`, `FoodTruckRestaurant` subclasses, each implementing all their unique features
- **Composition:** Create a `Restaurant` class that composes different service objects: `ReservationSystem`, `LocationTracker`, `MenuManager`, `PaymentProcessor`, `DressCodeValidator`, etc., where each restaurant configures which services it needs

Your answer:

- Which option would you choose? (Composition or Inheritance)
- Why? (Justify in 3-4 sentences using concepts from lecture)
- The chain wants to add a new "Ghost Kitchen" model (delivery only, no physical dining, operates multiple virtual brands from one location). How does your chosen design handle this?

Scenario 2: Smart Home Device Management

You're designing a smart home system that controls various devices:

- **Smart Thermostat:** Temperature control, schedule programming, learning user preferences, weather integration, energy usage tracking
- **Smart Lighting:** On/off, dimming, color changes, motion detection, sunrise/sunset automation
- **Smart Lock:** Lock/unlock, temporary access codes, entry logs, battery monitoring, tamper alerts
- **Smart Camera:** Live streaming, motion detection, cloud recording, facial recognition, two-way audio

Some devices need internet connectivity, some work offline. Some have battery backup, some are wired only. Some support voice control, some don't.

You're considering:

- **Inheritance:** Create device type hierarchy: `SmartDevice` → `ConnectedDevice` / `OfflineDevice` → `BatteryPowered` / `WiredDevice` → specific devices
- **Composition:** Create `SmartDevice` base class where each device composes the capabilities it needs: `NetworkInterface`, `PowerSource`, `VoiceHandler`, `SecurityModule`, `AutomationRules`, etc.

Your answer:

- Which option would you choose? (Composition or Inheritance)
- Why? (Justify in 3-4 sentences using concepts from lecture)
- What happens when you need to add a device that works both online AND offline depending on network availability? How does each option handle this?

Scenario 3: Academic Course Registration System

You're designing a university course registration system. Different courses have different requirements:

- **Lab Courses:** Require safety training completion, equipment checkout system, lab partner matching, hazardous material tracking
- **Online Courses:** Require proctoring software validation, timezone conversion for deadlines, discussion forum participation tracking, video streaming reliability check
- **Hybrid Courses:** Need both physical classroom + online components, attendance tracking in both modes, flexible deadline policies
- **Independent Study:** One-on-one faculty meetings, custom learning contracts, milestone-based grading, no fixed schedule

All courses need:

- Enrollment management (add/drop)
- Grade recording
- Prerequisite checking
- Credit hour calculation

You're considering:

- **Inheritance:** Create `Course` base class with subclasses for each course type (`LabCourse`, `OnlineCourse`, `HybridCourse`, `IndependentStudy`), each implementing their specific requirements
- **Composition:** Create a flexible `Course` class that composes various requirement validators, tracking systems, and delivery modes: `SafetyTraining`, `ProctoringSoftware`, `AttendanceTracker`, `LearningContract`, etc.

Your answer:

- Which option would you choose? (Composition or Inheritance)
- Why? (Justify in 3-4 sentences using concepts from lecture)
- A new "Competency-Based Course" is proposed where students progress by demonstrating skills rather than attending classes—it could be online, in-person, or hybrid, and students move at their own pace. How well does each option accommodate this?

Grading Criteria

- **1 point per scenario:** Correct choice with solid justification
- **1 point overall:** Consistent application of principles across all scenarios

---

Submission Guidelines

What to Submit

Submit a **single markdown file** named `Exercise_10_FamilyName_GivenName.md` via D2L.

Your file should include all five parts in order:

1. Part 1: SOLID Analysis
2. Part 2: Refactoring
3. Part 3: Generic Cache
4. Part 4: Design Decisions
5. Part 5: Reflection

Download the assignment template: Exercise_10_template - Use this as your starting point!

Formatting Requirements

- **Single file:** All parts in one markdown document
- **Code blocks:** Use fenced code blocks with language specification:

  ```
  ```java
  public class Example {
    // your code here with 2-space indentation
  }
  ```
  ```

- **Code quality:** Code should be semantically correct (proper syntax, logical structure, good names) but does not need to be fully compilable (e.g., you can omit imports, constructors, or minor implementation details if the concept is clear)
- **Headers:** Use markdown headers (##, ###) to organize your responses
- **Written responses:** Clear, complete sentences
- **Your info:** Include your name and date at the top of the file

Late Policy

[Your course late policy here]

Grading Rubric

Part	Points	Criteria
Part 1: SOLID Analysis	5	Correct identification and explanation of violations
Part 2: Refactoring	6	Working polymorphic design eliminating conditionals
Part 3: Generic Cache	5	Correct generic implementation with demos
Part 4: Design Decisions	4	Justified choices based on principles
Reflection	2	Thoughtful metacognitive analysis
TOTAL	**22**	

Detailed Grading

Part 1 (5 points):

- 2 points: Identifies SRP violation with clear explanation
- 2 points: Identifies OCP violation with clear explanation
- 1 point: Identifies at least one other violation (DIP, ISP, or additional SRP/OCP)

Part 2 (6 points):

- 2 points: Interface or abstract class with proper abstraction
- 2 points: Concrete implementations that eliminate conditionals
- 1 point: Demonstrates Open/Closed Principle in client code
- 1 point: Code quality and formatting

Part 3 (5 points):

- 1 point: Correct generic type parameters declared
- 1 point: All methods implemented correctly
- 1 point: FIFO eviction works when at capacity
- 1 point: Type-safe usage throughout
- 1 point: Demo shows reusability with 3+ different type combinations

Part 4 (4 points):

- 1 point per scenario: Choice is justified with specific principles
- 1 point: Answers demonstrate understanding across different contexts

Reflection (2 points):

- 1 point: Thoughtful analysis of learning process and challenges
- 1 point: Actionable insights about personal growth as a programmer

---

Academic Integrity

This is an **individual** assignment. You may:

- Refer to lecture slides and textbook
- Use Java documentation
- Ask clarifying questions in office hours or discussion board

You may NOT:

- Share code with other students
- Use AI tools to generate solutions (you may use them to check syntax only)
- Copy code from online sources

Remember: The goal is to practice these concepts. Shortcuts now hurt your learning and your exam performance.

Hints and Tips

For Part 1 (SOLID Analysis):

- Look for classes doing too many things (SRP)
- Look for if/else chains on types (OCP)
- Look for concrete dependencies (DIP)
- There are multiple violations - find at least 3

For Part 2 (Refactoring):

- Start with the interface/abstract class first
- Each product type should be its own class
- The discount calculation should be in each subclass
- No if/else statements should remain in the client code

For Part 3 (Generic Cache):

- Start with `public class Cache<K, V>`
- For FIFO eviction: get the first key, remove it, then add new item
- Test with at least 3 different type combinations

For Part 4 (Composition vs. Inheritance):

- Consider: Can you mix and match behaviors? (Composition better)
- Consider: Is it a true "is-a" relationship? (Inheritance might be OK)
- Consider: Will you need flexibility later? (Usually composition)
- Reference specific principles: LSP, OCP, flexibility, etc.

Part 5: Metacognitive Reflection (2 points - REQUIRED)

Purpose

Developing metacognitive skills—the ability to think about your own thinking—is essential for becoming an excellent programmer. Research shows that students who reflect on their learning process learn more effectively and retain knowledge longer.

Instructions

Create a file called `reflection.md` and respond to the following prompts. Each response should be 3-5 sentences of thoughtful, honest reflection.

1. Learning Process Analysis

Prompt: Which part of this assignment was most challenging for you? Describe not just WHAT was hard, but WHY it was difficult. What specific knowledge or skill were you missing? How

did you work through the challenge?

What we're looking for: Honest analysis of your struggle, identification of the gap in understanding, description of your problem-solving approach.

2. Conceptual Connections

Prompt: Choose one concept from this assignment (generics, SOLID principles, composition vs. inheritance, or polymorphism). Explain how this concept connects to something you already knew or have experienced in previous programming. What's similar? What's different? How does this new understanding change how you think about code?

What we're looking for: Evidence of integration with prior knowledge, depth of understanding beyond surface level.

3. Application to Real Projects

Prompt: Think about a project you want to build (personal project, startup idea, or career goal). Identify a specific design decision you would need to make in that project. Which principle from this assignment would guide that decision? Be concrete—describe the actual scenario and explain your reasoning.

What we're looking for: Ability to transfer learning to new contexts, practical application thinking.

4. Growth Mindset

Prompt: What did you learn about yourself as a programmer through this assignment? What's one specific skill or habit you want to develop further based on what you learned? What concrete action will you take to develop it?

What we're looking for: Self-awareness, actionable plans for improvement, growth orientation.

Example (DO NOT COPY)

Here's an example of good vs. insufficient reflection:

Insufficient:

> "Part 2 was hard. I eventually figured it out by looking at examples. I would use these concepts in my projects."

Good:

> "Part 2 was challenging because I kept thinking procedurally—my instinct was to write the discount calculation in one place with if/else statements. The difficulty was recognizing that I was thinking about 'operations on data' instead of 'objects with behavior.' I worked through it by first writing the procedural version, then asking myself 'what varies here?' (the discount calculation), then realizing each product type should know how to calculate its own discount. This shift from 'what should I do to this data?' to 'what should this object do?' is a fundamental mindset change that I'm still developing."

Grading Criteria

- **2 points:** Thoughtful, specific, honest reflection that demonstrates metacognitive thinking
- **1 point:** Surface-level responses that show some reflection but lack depth
- **0 points:** Generic responses that could apply to any assignment, or missing reflection

Note: There are no "wrong" reflections—only superficial ones. Honest struggles are valued. The goal is to develop self-awareness about your learning process.

Learning Outcomes

By completing this assignment, you will have practiced:

✅ Analyzing code for design principle violations
✅ Refactoring procedural code to object-oriented design
✅ Creating generic, reusable components
✅ Making informed design decisions between composition and inheritance
✅ Applying SOLID principles in practical contexts
✅ Reflecting on your learning process to develop metacognitive skills

These skills are **essential** for professional software development and will appear again on exams and future assignments. The reflection component helps you develop self-awareness about how you learn, which research shows improves long-term retention and problem-solving ability.

Good luck! 🚀