Howard University
School of Engineering &Architecture
Department of Electrical Engineering & Computer Science

Large Scale / Object – Oriented Programming
Spring 2025

Final Exam

100 pts.

Name :_____Oshione Adams_____

No collaboration is allowed on this exam, anything else is fair game.

Make sure you validate that all submission to your repo were successful

Question #1 (30 pts.)

Description:

You are testing a ShoppingCart class that allows users to add and remove items, compute total cost, and apply discount codes. Your task is to **write JUnit test cases** that verify the behavior of this class under various conditions.

```
package org.howard.edu.lspfinal.question1;

import java.util.HashMap;
import java.util.Map;

/**
 * Represents a simple shopping cart that allows adding items,
 * applying discount codes, and calculating the total cost.
 */
public class ShoppingCart {
    private Map<String, Double> items = new HashMap<>();
    private double discountPercentage = 0.0;

    /**
     * Adds an item to the shopping cart.
     *
     * @param itemName  the name of the item
     * @param price     the price of the item (must be non-negative)
     * @throws IllegalArgumentException if price is negative
     */
    public void addItem(String itemName, double price) {
        if (price < 0) {
            throw new IllegalArgumentException("Price cannot be negative.");
        }
        items.put(itemName, price);
    }

    /**
     * Calculates and returns the total cost of the cart,
     * applying any discounts currently in effect.
     *
     * @return total cost after applying discount
     */
    public double getTotalCost() {
        double total = 0.0;
        for (double price : items.values()) {
```

```
            total += price;
        }
        double discountAmount = total * (discountPercentage / 100.0);
        return total - discountAmount;
    }

    /**
     * Applies a valid discount code to the shopping cart.
     * Supported codes:
     * - "SAVE10": 10% discount
     * - "SAVE20": 20% discount
     *
     * @param code  the discount code
     * @throws IllegalArgumentException if the code is invalid
     */
    public void applyDiscountCode(String code) {
        switch (code) {
            case "SAVE10":
                discountPercentage = 10.0;
                break;
            case "SAVE20":
                discountPercentage = 20.0;
                break;
            default:
                throw new IllegalArgumentException("Invalid discount code.");
        }
    }

    /**
     * Returns the current discount percentage applied to the cart.
     * Useful for testing.
     *
     * @return the discount percentage (0.0 if no discount applied)
     */
    public double getDiscountPercentage() {
        return discountPercentage;
    }
}
```

Requirements:

Write a JUnit test class named `ShoppingCartTest` under the package `org.howard.edu.lspfinal.question1`.

You must:

1. Test adding items:
    - Valid items (name + price)
    - Invalid inputs (empty name, negative/zero price)
2. Test removing items:
    - Removing existing items
    - Attempting to remove non-existent items
3. Test total cost:
    - With and without discounts
    - With no items in the cart
4. Test applying discount codes:
    - Valid codes ("SAVE10", "SAVE20")
    - Invalid codes (e.g., "SAVE50", "")
5. Test cart size updates correctly after adding/removing

Deliverables:

A JUnit test class named `ShoppingCartTest.java` and implementation class `ShoppingCart under` package `org.howard.edu.lspfinal.question1`.

- All tests should compile and run without error

Grading Criteria:

Test for adding valid item **[3 pts.]**
Test for adding item with 0 price (expect exception) **[3 pts.]**
Test for adding item with negative price (expect exception) **[3 pts.]**
Test for applying "SAVE10"**[3 pts.]**
Test for applying "SAVE20" **[3 pts.]**
Test for applying invalid code (expect exception) **[3 pts.]**
Test total cost without discount **[4 pts.]**
Test total cost with discount **[5 pts.]**
Test total cost with empty cart **[3 pts.]**

Use the **@DisplayName** annotation to name your JUnit test cases according to the above.

Question #2 (30 pts.)

Description:

You are building a **Task Tracker** system for a team to manage and monitor their work. Each **Task** has the following properties:

1. A unique **name** (e.g., "Fix Bug #204")
2. An integer **priority** (lower number = higher priority)
3. A **status**: "TODO", "IN_PROGRESS", or "DONE"

To make this system reliable and user-friendly, it must handle invalid operations gracefully by using **custom checked exception classes**.

You are to design and implement a TaskManager system in Java that includes:

1. **Add a New Task**
    - A task name must be **unique** (no duplicates).
    - If a task with the same name already exists, throw a custom exception called DuplicateTaskException.
    - This exception must include a helpful message, such as "Task 'Fix Bug #204' already exists."
2. **Retrieve a Task by Name**
    - If no such task exists, throw a custom exception called TaskNotFoundException.
    - The message should clearly indicate the missing task, e.g., "Task 'Deploy App' not found."
3. **Update the Status of an Existing Task**
    - Valid statuses are "TODO", "IN_PROGRESS", and "DONE".
    - If the task does not exist, throw a TaskNotFoundException.
4. **Print All Tasks Grouped by Status**
    - For each status, print the list of tasks currently assigned to it.

Requirements:

1. Choose the appropriate design/data structure to implement your TaskManager.
2. Define and use two custom exception classes:
    a. `DuplicateTaskException`
    b. `TaskNotFoundException`

Sample Driver:

```
/**
 * Driver class for testing the TaskManager system.
```

```
*/
public class Driver {
package org.howard.edu.lspfinal.question2;

public static void main(String[] args) {
        TaskManager manager = new TaskManager();

        try {
            manager.addTask("Fix Bug #204", 1, "TODO");
            manager.addTask("Write Docs", 3, "TODO");
            manager.addTask("Setup CI/CD", 2, "IN_PROGRESS");

            // Duplicate task
            manager.addTask("Fix Bug #204", 4, "DONE");
        } catch (DuplicateTaskException e) {
            System.out.println("Error: " + e.getMessage());
        }

        try {
            Task t = manager.getTaskByName("Write Docs");
            System.out.println("Retrieved: " + t);
        } catch (TaskNotFoundException e) {
            System.out.println("Error: " + e.getMessage());
        }

        try {
            manager.updateStatus("Write Docs", "DONE");
            manager.updateStatus("Nonexistent Task", "IN_PROGRESS");
        } catch (TaskNotFoundException e) {
            System.out.println("Error: " + e.getMessage());
        }

        // Print all tasks grouped by status
        manager.printTasksGroupedByStatus();
    }
}
```

Sample Output:

```
Error: Task 'Fix Bug #204' already exists.
Retrieved: Task{name='Write Docs', priority=3, status='TODO'}
Error: Task 'Nonexistent Task' not found.
Tasks grouped by status:
TODO:
  Task{name='Fix Bug #204', priority=1, status='TODO'}
IN_PROGRESS:
```

```
  Task{name='Setup CI/CD', priority=2, status='IN_PROGRESS'}
DONE:
  Task{name='Write Docs', priority=3, status='DONE'}
```

Deliverables:

1. Use Sample Driver as your driver, call it Driver.java. Make sure your driver produces the same output from the supplied driver code.
2. **Javadoc comments** for all public classes and methods
3. All code should be uploaded to your repo. Solution should be under **package org.howard.edu.lspfinal.question2**

Grading Criteria:

1. **15 points**. For correctly implementing addTask, getTaskByName, updateStatus, and printTasksGroupedByStatus. Task names are unique, and status updates work as expected.
2. **5 points**. For proper implementation of custom exceptions
3. **5 points**: For clear, concise, and correct Javadoc for all public classes and methods.
4. **5 points**: correct implementation of the provided driver.

Question #3 (30 pts.)

You are designing part of a reporting subsystem for a business application. Different departments in the company—like Sales, Inventory, and others—need to generate reports. While each report has its own specific behavior, they all follow the same general workflow:

1. Load the data
2. Format the data
3. Print the final report

However, how these steps are implemented varies from one report to another.

Your task is to design and implement a solution that supports this structure while allowing future developers to add new types of reports with minimal duplication and maximum code reuse.

Your Task:

Design and implement a solution in Java that models this reporting system using object-oriented principles. The core reporting workflow should be reusable and extensible. Your design should:

- Capture the common structure of report generation in a base class.
- Allow each type of report to define its own implementation for the steps in the workflow.
- Demonstrate your design with at least two different kinds of reports that behave differently in each step.
- Show how a client might use your framework to create and generate each kind of report. Create a **Driver.java** class that demonstrates the correct behavior by producing the expected output (see below).

Additional Notes:

- Use your knowledge of **the Template Method design pattern** to structure your solution.
- Your design will be evaluated on clarity, adherence to object-oriented principles, and correct use of the pattern.
- All code should be uploaded to your repo. Solution should be under **package org.howard.edu.lspfinal.question3**

Sample Output from Driver.java:

Loading sales data...
Formatting sales data...
Printing sales report.

Loading inventory data...
Formatting inventory data...
Printing inventory report.

Grading Criteria:

Template Method Structure **[15 pts.]**
Working Driver with Output **[10 pts.]**
Code Quality & Clarity **[5 pts.]**

Question #4 (10 pts.)

During our discussions of design patterns over the last two weeks we've discussed many reasons why incorporating them into your software design can be advantageous. Briefly discuss one instance where using a design pattern may not be beneficial. Maximum points will be earned on this question for providing discussion presented during lecture and not on information obtained from the internet.

Solution should be under **package org.howard.edu.lspfinal.question4**. Embed answer into this document and upload to your repo. E-mail to bwoolfolk@whiteboardfederal.com if you have difficulties uploading to your repo. Make sure you get confirmation that I received it.

Grading Criteria:

10 points. For correctly identifying discussion from lecture
5 points. For identifying a reasonable answer but it does not correlate to class/lecture discussion.

Design patterns are an important aspect of software design, helping to solve common problems in a reusable manner. However, there are instances where using them may not

be beneficial. One key situation we discussed in class is applying a design pattern to a simple problem where it isn't necessary. This often results in overengineering, which can complicate the codebase and make it harder to maintain.

For example, we examined the Singleton pattern in class, which is commonly used to restrict a class to a single instance. While it can be useful for managing shared resources, such as loggers or database connections, we also noted how its misuse can lead to tight coupling and poor testability. When everything depends on a single instance, it becomes more challenging to replace or mock it during testing, which contradicts important object-oriented design principles we've discussed, such as "encapsulate what varies" and "design to an interface, not an implementation."

Another point that arose in our lectures is that some design patterns can increase complexity without necessarily enhancing flexibility or scalability. During our introduction to design patterns, we recognized that while they may promote flexibility, they often do so at the expense of making the design more complicated. Additionally, there's a risk that newer developers might not understand the pattern being used, especially if applied to a situation where a straightforward solution would have sufficed.

In our class examples, we emphasized the KISS principle (Keep It Simple, Stupid). This principle reminds us not to use design patterns merely to demonstrate our knowledge. Instead, they should only be applied when they genuinely simplify a recurring problem or enhance the system's ability to scale or adapt. If a pattern introduces more boilerplate code or complexity than the original problem, it negates the aim of good design.

In summary, design patterns are valuable tools, but they should be used intentionally. Applying them solely for the sake of using patterns can lead to more rigid and less maintainable code. As we learned, a good designer knows not just how to use design patterns but also when to refrain from using them. Design patterns are an important aspect of software design, helping to solve common problems in a reusable manner. However, there are instances where using them may not be beneficial. One key situation we discussed in class was applying a design pattern to a simple problem where it wasn't necessary. This often results in overengineering, which can complicate the codebase and make it harder to maintain.

References

Question 1: ShoppingCart + JUnit Testing

[1] JUnit Team, "JUnit 5 User Guide," *JUnit*, 2024. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/
[2] Oracle, "Creating Your Own Exception Classes," *The Java Tutorials*, Oracle Corporation, 2023. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/creating.html
[3] B. Woolfolk, "Lecture: Interfaces and Patterns," *Howard University LSP Course*, 2024. [Lecture Slides].

Question 2: TaskManager System with Exceptions

[2] Oracle, “Creating Your Own Exception Classes,” *The Java Tutorials*, Oracle Corporation, 2023. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/creating.html
[3] B. Woolfolk, “Lecture: Interfaces and Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[4] B. Woolfolk, “Lecture: Introduction to Design Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].

Question 3: Reporting System – Template Method Pattern

[3] B. Woolfolk, “Lecture: Interfaces and Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[4] B. Woolfolk, “Lecture: Introduction to Design Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[5] B. Woolfolk, “Lecture: Template, Singleton, Facade Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Boston, MA, USA: Addison-Wesley, 1994.

Question 4: When Not to Use a Design Pattern (Written Response)

[4] B. Woolfolk, “Lecture: Introduction to Design Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[5] B. Woolfolk, “Lecture: Template, Singleton, Facade Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].
[3] B. Woolfolk, “Lecture: Interfaces and Patterns,” *Howard University LSP Course*, 2024. [Lecture Slides].