Tutor Marked Exercises 4 Part 1, Questions, Design Document, Testing Plan

Unit 8,9 Learning Objective Questions:

1. Characteristics and Advantages of Swing (TIJ pages 1304–1305)

Swing is a part of Java's **Java Foundation Classes (JFC)** that provides a rich set of GUI components. Key characteristics include:

- **Lightweight:** Components are not tied to native OS widgets. Swing components are written in java allowing them to be lightweight and platform-independent, ensuring consistent behavior across different operating systems.
- **Pluggable Look and Feel:** Allows customization of UI appearance. Swing allows developers to change the appearance of applications dynamically.
- **MVC Architecture:** Separates data (Model), UI (View), and user interaction (Controller).
- **Event-Driven Programming:** Uses event listeners for user interactions.
- Supports advanced components like tables, trees, and sliders.
- Swing uses a single-threaded painting model, where all GUI updates are handled on the Event Dispatch Thread (EDT). This design requires careful management to ensure thread safety but provides a consistent approach to handling GUI events.

---

2. What is a Deadlock? Conditions for Deadlock (TIJ pages 1223, 1227–1228)

A **deadlock** occurs when two or more threads are stuck waiting for resources held by each other, preventing further execution. Any situation where a set of processes or threads are blocked because each process is holding a resource and waiting for another resource held by another process. As a result, none of the processes can proceed, leading to a standstill.

Conditions for Deadlock:

1. **Mutual Exclusion:** Resources cannot be shared; only one thread can use a resource at a time.
2. **Hold and Wait:** A thread holding a resource waits for another resource.
3. **No Preemption:** Resources cannot be forcibly taken from a thread.
4. **Circular Wait:** A cycle of dependencies exists between threads.

Example of Deadlock:

```
class DeadlockExample {
    static final Object lock1 = new Object();
    static final Object lock2 = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock 
1...");
                try { Thread.sleep(50); } catch (InterruptedE
xception e) {}
                synchronized (lock2) {
                    System.out.println("Thread 1: Acquired lo
ck2");
                }
            }
        });
```

```
        Thread t2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock 2...");
                try { Thread.sleep(50); } catch (InterruptedException e) {}
                synchronized (lock1) {
                    System.out.println("Thread 2: Acquired lock1");
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

This example demonstrates two threads waiting on each other, leading to deadlock.

Difficulties Encountered During Development

During the development of this assignment, several challenges were encountered across different areas of Java programming, including concurrency, reflection, event-driven programming, synchronization, and object-oriented design principles. Below are the primary difficulties faced and how they impacted the development process.

1. Java Concurrency and Multithreading

Implementing a **multithreaded event-driven system** introduced various challenges, primarily related to **thread lifecycle management** and **synchronization**.

Challenges Faced:

- Understanding the `Runnable` **interface** and how to correctly implement it for custom event-handling classes.
- Managing the **thread lifecycle**—starting, pausing (suspending), resuming, and stopping threads dynamically based on system state.
- Synchronizing **concurrent access to shared resources** to prevent **race conditions**, especially when multiple events were modifying the system state simultaneously.
- Choosing the right **thread-safe collections** to store and manage event states efficiently while ensuring **event execution order**.

Lessons Learned & Solutions:

- Implemented **Java's built-in thread synchronization mechanisms**, such as `synchronized` blocks and `ReentrantLock`, to coordinate shared state access.
- Used **ConcurrentHashMap** and **CopyOnWriteArrayList** to ensure thread-safe modifications without excessive locking.
- Employed the **ExecutorService framework** to manage a **pool of worker threads** for event execution, improving scalability.

---

2. Java Reflection API

The **Reflection API** was used to enable **dynamic event creation** and allow **loose coupling** between the event system and the `GreenhouseControls` application.

Challenges Faced:

- Understanding how to **dynamically instantiate event classes** using `Class.forName()` and `getConstructor().newInstance()`.
- Ensuring that new event classes could be **added dynamically without recompiling** `GreenhouseControls`.
- Handling **checked exceptions** such as `ClassNotFoundException`, `InstantiationException`, and `IllegalAccessException`, which occurred when dynamically loading classes.

Lessons Learned & Solutions:

- Used **reflection with factory patterns** to instantiate event objects dynamically, enabling flexible event handling.
- Implemented **better exception handling** to catch and log `ClassNotFoundException`, ensuring that missing event classes do not crash the system.
- Applied **design patterns like dependency injection** to further decouple event instantiation from event execution.

---

3. Event-Driven Programming

The **event-driven architecture** required careful planning to allow multiple events to operate **independently** while responding to **state changes**.

Challenges Faced:

- Designing an event system that allowed **multiple concurrent events** to execute without blocking each other.
- Implementing **event handlers** that could respond to specific **state changes** in `GreenhouseControls`.
- Maintaining an **event queue** to properly schedule and execute events **in parallel**.

Lessons Learned & Solutions:

- Implemented **a priority queue** for managing event execution order efficiently.
- Used **observer patterns** to notify event listeners about state changes, reducing direct coupling between components.
- Ensured that **long-running events did not block other tasks** by using a **thread pool** instead of a single-threaded event loop.

---

4. Synchronization and Shared State Management

To handle **concurrent state modifications**, synchronization techniques were applied to prevent **data corruption**.

Challenges Faced:

- **Race conditions** when multiple threads tried to modify the system state at the same time.
- **Deadlocks** caused by improper locking mechanisms when multiple events tried to acquire locks in an inconsistent order.
- Ensuring **event consistency** while minimizing performance overhead from excessive locking.

Lessons Learned & Solutions:

- Replaced traditional **state variables** with `TwoTuple<K, V>`, a generic key-value pair structure, to simplify state updates.
- Used **synchronized methods** where necessary but avoided overuse to **prevent performance bottlenecks**.
- Ensured **lock ordering consistency** to **eliminate deadlocks** by following best practices in concurrency.

---

5. Object-Oriented Design Principles

Maintaining **clean and modular design** was essential to ensure the **scalability** of the system.

Challenges Faced:

- Maintaining **proper encapsulation** to keep state management separate from event execution logic.
- Implementing **loose coupling** so that new event types could be added easily.
- Ensuring that the code adhered to **SOLID principles** to improve maintainability.

Lessons Learned & Solutions:

- Used **interfaces and abstract classes** to define a standard event structure, ensuring modular and reusable event handling.
- Implemented **dependency injection** to decouple components.

- Applied **Open/Closed Principle (OCP)** by designing the system to allow extension without modifying existing code.

---

6. Java Collections Framework

Using **Java's collection classes** effectively was necessary for **storing and managing event objects** efficiently.

Challenges Faced:

- Choosing the right **data structure** to store active and completed events.
- Efficiently **retrieving and modifying event data** while maintaining thread safety.

Lessons Learned & Solutions:

- Used `LinkedHashMap` to maintain event order while allowing quick lookups.
- Applied `ConcurrentHashMap` for thread-safe event state storage.
- Used `PriorityQueue` for scheduling time-sensitive events efficiently.

---

7. Exception Handling

Handling runtime errors gracefully was crucial to **ensuring system stability**.

Challenges Faced:

- Creating a structured way to handle **event-related exceptions**.
- Ensuring that **exceptions did not cause event execution failures**.

Lessons Learned & Solutions:

- Defined a **custom exception hierarchy**, including `ControllerException`, to handle specific error cases.
- Used **try-catch-finally** blocks effectively to ensure **graceful recovery** from failures.
- Logged critical exceptions to facilitate debugging.

TME 4: Test Plan and Design Document (Part 1: Concurrency)

Course: Computer Science 308 – Java for Programmers

Assignment: TME 4

Author: Boris Bojanov

Student ID: 3608550

Date: Jan 31, 2025

---

1. Overview

This part of the assignment involves redesigning `GreenhouseControls` to use concurrency by implementing `Runnable` for `Event` classes. The key modifications include:

1. Making `Event` implement `Runnable` so each event controls its own timing.
2. Creating event classes dynamically using Java reflection.
3. Managing events as independent threads with the ability to suspend and resume execution.
4. Replacing state variables with a synchronized collection of `TwoTuple` objects.
5. Providing output to a graphical user interface (GUI).

---

2. Test Plan

2.1 Objectives

- Validate concurrent execution of `Event` objects.
- Ensure correct integration of reflection for event instantiation.
- Test suspend and resume functionality for event threads.
- Verify `TwoTuple` correctly maintains state updates.
- Ensure synchronization prevents race conditions when modifying state.

- Test GUI integration for event output.

2.2 Compile & Run Instructions

1. **Compilation:**
    - `javac -d bin src/tme4/*.java`
2. **Execution:**
    - Run GreenhouseControls: `java GreenhouseControls -f examples1.txt`

2.3 Test Cases:

Test Case 1: Event Execution as Threads

- **Purpose:** Verify that `Event` objects run as independent threads.
- **Input:** Create multiple `Event` instances and start them.
- **Expected Result:**
    - Each event runs independently without blocking other events.
    - Console logs indicate simultaneous event execution.

Test Case 2: Dynamic Event Instantiation via Reflection

- **Purpose:** Validate event creation using Java reflection.
- **Input:** Provide an event name as a string and use reflection to instantiate it.
- **Expected Result:**
    - The event is successfully instantiated and runs.
    - No compilation dependency on specific event classes.

Test Case 3: Suspend and Resume Events

- **Purpose:** Verify that events can be suspended and resumed.
- **Input:** Start multiple events and issue a suspend command followed by a resume command.
- **Expected Result:**

- Events pause execution upon suspension.
- Events resume execution when resumed.

---

3. Design Document

3.1 Classes and Methods

3.1.1 GreenhouseControls

- **Fields:**
  - `List<Thread> activeThreads`
  - `Map<String, TwoTuple<String, Object>> state`
- **Methods:**
  - `void startEvent(String eventClass)`
  - `void suspendAll()`
  - `void resumeAll()`
  - `void setVariable(String key, Object value)`

3.1.2 Event (Implements Runnable)

- **Fields:**
  - `Thread eventThread`
- **Methods:**
  - `void run()`
  - `void start()`
  - `void suspend()`
  - `void resume()`

3.1.3 TwoTuple

- **Purpose:** Stores key-value pairs for state updates.

- **Methods:**
    - `String getKey()`
    - `Object getValue()`

3.1.4 Reflection-Based Event Instantiation

- **Purpose:** Enables runtime event creation.
- **Implementation:** Uses `Class.forName(eventName).getConstructor().newInstance()`.

3.1.5 Synchronization Strategy

- **Purpose:** Prevents concurrent state modification issues.
- **Implementation:** Uses `synchronized` methods for `setVariable`.

End of Document

NOTE:

The following link is not working properly!!!

“Please see testplan.html for a sample test plan.” (https://scis.lms.athabascau.ca/file.php/422/tme_files/guidelines.htm)

Results in a black page. DegreeWorks does not present me with a way to find the template of a test plan. I have sent Emails requesting an example test plan.

```
Click here to access DegreeWorks using myAU login
```