TME\_4

Documentation

&

Test Plan

Odinakachukwu Nzekwe

Athabasca University

**Computer Science 308:**

Java for Programmers

COMP 308

Problem\_1 Documentation:

**Greenhouse Controls: A System for Managing Automated Events in a Greenhouse**

The **Greenhouse Controls** application is a sophisticated system designed to simulate and manage various automated events in a greenhouse. Its purpose is to model how different devices and conditions within the greenhouse, such as lights, could be controlled using timed events. The system is designed with flexibility and extensibility in mind, offering the capability to add, suspend, and resume events dynamically. This essay provides an overview of the key components of the system, the role of threads and synchronization, and how the user interacts with the system through the graphical user interface (GUI).

**Components of the Greenhouse Controls System**

The system is built around a central concept of events that trigger specific actions after a set delay. These events, such as turning lights on or off, are represented by the abstract Event class and are extended by specific event classes like LightOnEvent and LightOffEvent. Each event is associated with a delay time, which determines how long the system will wait before performing the associated action. For example, a LightOnEvent will turn the lights on after a given delay, while a LightOffEvent will do the opposite.

The event system is designed to be highly flexible. It uses the concept of dynamic event creation, where the event classes are instantiated at runtime using Java's reflection mechanism. This allows new event types to be added without altering the core system code. The core logic for adding an event is encapsulated in the GreenhouseControls class, which manages the state variables of the greenhouse and maintains a list of event threads.

One of the key features of this system is its ability to handle multiple events concurrently. Each event runs in its own thread, ensuring that they execute independently and can trigger their actions after their respective delays without blocking the execution of other events. The threads are managed using the ThreadWrapper class, which provides functionality for suspending and resuming threads.

**Synchronization and Thread Management:**

The Greenhouse Controls system relies heavily on multithreading to simulate the concurrent operation of multiple devices within the greenhouse. However, managing multiple threads can lead to complications, particularly when ensuring that events can be safely suspended and resumed. To address this, the system uses the ThreadWrapper class, which encapsulates a Thread and adds the ability to pause or resume it.

Synchronization is a critical aspect of this system. The GreenhouseControls class uses a Lock object to synchronize the suspension and resumption of event threads. This ensures that actions like suspending or resuming all events are performed atomically, preventing race conditions and maintaining the integrity of the system. The lock ensures that, while one event is being suspended or resumed, no other operations are allowed to interfere.

In addition to the Lock object, the threads themselves are synchronized using the suspend() and resume() methods within the ThreadWrapper class. When a thread is suspended, it enters a waiting state, where it will remain until it is resumed. This mechanism allows the system to halt the execution of events at any time, offering flexibility in controlling the greenhouse environment.

**The Graphical User Interface (GUI):**

The system’s GUI plays a vital role in making the Greenhouse Controls accessible to the user. It allows users to add new events, suspend ongoing events, and resume them as needed. The GUI is built using Java Swing components, such as JButton for the action buttons and JTextArea for displaying logs of event actions.

The user can interact with the system through three main buttons:

1. **Add Event**: This button allows users to enter the name of an event class (e.g., LightOnEvent) and a delay time in milliseconds. The event is then added to the system, and its execution begins after the specified delay.
2. **Suspend All**: This button suspends the execution of all events. When clicked, all active threads are paused, preventing any actions from being performed.
3. **Resume All**: This button resumes the execution of all suspended events. When clicked, the threads are resumed, and the system continues executing events based on their delays.

The log area within the GUI displays real-time feedback to the user. For instance, when a new event is added, the log will show the event’s name and the delay time. Similarly, when events are suspended or resumed, the log will reflect these actions, keeping the user informed of the system's state.

**Event Handling and Reflection:**

One of the most interesting aspects of the Greenhouse Controls system is its use of reflection to dynamically create events. The addEvent() method in the GreenhouseControls class takes the name of an event class and its associated delay time as input. Using Java's reflection API, the method dynamically loads the specified event class, constructs an instance of it, and starts its execution in a separate thread.

This approach adds a layer of flexibility to the system, as it allows new events to be added without modifying the core system. For example, if a new type of event were required—say, a TemperatureChangeEvent—it could simply be created by extending the Event class and implementing the performAction() method. Once the new event class is created, it can be added to the system without needing any changes to the existing codebase, making the system easily extensible.

Problem\_2 Documentation:

**Documentation and Test Plan for Greenhouse Controls Application**

The GreenhouseControlsApp is a Java-based graphical user interface (GUI) application designed to manage and simulate greenhouse operations. This application provides an interactive platform for users to control the lifecycle of greenhouse events, such as starting, stopping, suspending, and resuming processes. The system employs the Swing framework to deliver a user-friendly interface and integrates various components like buttons, menus, and popup options to enhance usability.

**Overview of the Application**

The primary goal of the GreenhouseControlsApp is to allow users to manage greenhouse operations effectively through an intuitive interface. The application features a log area where users can track actions and system messages in real time. Key functionalities include the ability to start and stop events, restart processes, suspend and resume operations, and terminate events after a user-specified delay. Furthermore, the system allows users to open event files and restore greenhouse states from saved files, ensuring operational continuity.

The application is built around two main classes: GreenhouseControlsApp and GreenhouseControls. The former serves as the GUI layer, providing interactive elements and linking them to backend operations. The latter acts as the core logic layer, managing the state and execution of greenhouse events using a thread pool for multitasking. Together, these classes create a cohesive system that combines user-friendly controls with robust backend functionality.

**Key Features**

The GreenhouseControlsApp includes several features to enhance usability and functionality. First, the application provides a set of buttons for basic operations: Start, Restart, Terminate, Suspend, and Resume. Each button is enabled or disabled based on the current state of the greenhouse operations, ensuring users can only perform valid actions. Second, the application includes a menu bar with options for opening event files, restoring states, creating new windows, and exiting the application. These options come with associated keyboard shortcuts for quick access.

Additionally, a popup menu replicates the functionality of the control buttons, offering users an alternative interaction method. The application also features file management capabilities, allowing users to open event files and restore the greenhouse state from a saved file. A confirmation mechanism is in place to prevent accidental closures or exits while operations are running, further enhancing reliability.

**Test Plan**

To ensure the correctness and robustness of the GreenhouseControlsApp, a comprehensive test plan has been developed. The test plan focuses on verifying the functionality of each component, both in isolation and as part of the overall system. The following test cases are defined:

1. **Start Button Test**: Clicking the Start button should initiate greenhouse operations and update the log with appropriate messages.
2. **Restart Button Test**: Clicking the Restart button should add a restart event to the system and display a confirmation in the log.
3. **Terminate Button Test**: Clicking the Terminate button should prompt the user for a delay and schedule a termination event accordingly.
4. **Suspend and Resume Test**: These buttons should toggle the state of the greenhouse operations, with the log reflecting the changes.
5. **File Management Tests**: Selecting "Open Events" or "Restore" from the menu should allow users to select a file and provide feedback on the outcome, including error messages for invalid files.
6. **Window Management Tests**: Creating new windows and closing existing ones should function as expected, with confirmation prompts where necessary.
7. **Exit Application Test**: Attempting to exit the application while events are running should trigger a confirmation dialog.

Each test case includes specific input actions and expected outcomes, ensuring that all functionalities are covered. The application is tested on various platforms, including macOS, Windows, and Linux, with Java 17 or higher as the runtime environment.

**Known Limitations and Future Improvements**

While the GreenhouseControlsApp provides a strong foundation for managing greenhouse operations, there are areas for improvement. For instance, the file management features currently lack the ability to parse and validate event files. Adding this functionality would enhance the application's utility. Additionally, user input validation for fields like termination delay could prevent errors and improve user experience. Error handling for unexpected scenarios, such as file access issues or thread interruptions, is another area that could be enhanced.