Names: Madison Betz and Kaiden Pollesch

Date: 4/28/2025

Assignment Title: Lab 12 A State-Driven Security Light

Repository Link: https://github.com/msoe-SWE2721/2025-swe2721-lab-12-state-machines-121_betz_pollesch_lab12_2025.git

Introduction

What are you trying to accomplish with this lab?

In this lab, we are analyzing and testing a security light control system that is implemented as a finite-state machine. Specifically, we are working to develop comprehensive state transition test cases to verify the correctness of the implementation against its formal specification. We will be identifying and fixing defects in the existing code while also creating a new, simplified implementation that meets the same specifications. Additionally, we will be using mock objects to verify correct system operation, comparing complexity metrics between implementations, and exploring the advantages of state-based testing versus control flow analysis. Throughout this process, we will gain practical experience with formal specification-based testing approaches.

Diagrams

Control Flow Graph for Initial signalAction method

[START] → [Check currentState]
  ↓
[LAMP_OFF_DAYLIGHT] → [Check request] → [LIGHT_SENSOR_DARKENED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [MANUAL_SWITCH_ON] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [Other requests] → [END]
[LAMP_ON_FULL_BRIGHTNESS] → [Check request] → [MANUAL_SWITCH_OFF] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [Other requests] → [END]
[LAMP_OFF_NIGHTIME] → [Check request] → [LIGHT_SENSOR_LIGHTENED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [MANUAL_SWITCH_ON] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [MOTION_DETECTED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [SECURITY_ALARM_TRIPPED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [Other requests] → [END]
[LAMP_ON_NIGHTIME_BRIGHTNESS] → [Check request] → [MANUAL_SWITCH_OFF] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [LIGHT_SENSOR_LIGHTENED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [SECURITY_ALARM_TRIPPED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [Other requests] → [END]
[MOTION_DETECTED] → [Check request] → [MOTION_DETECTION_TIMER_EXPIRED] → [handleExitConditions] → [handleEntryConditions] → [END]
  ↓ → [LIGHT_SENSOR_LIGHTENED] → [handleExitConditions] → [handleEntryConditions] → [END]

↓ → [SECURITY_ALARM_TRIPPED] → [handleExitConditions] → [handleEntryConditions] → [END]
↓ → [MANUAL_SWITCH_OFF] → [handleExitConditions] → [handleEntryConditions] → [END]
↓ → [Other requests] → [END]
[INTRUSION_DETECTED] → [manageIntrusionDetectedState] → [END]

Calculation
V(G) = E – N + 2 = 43 – 30 + 2 = 15
E: Number of edges = 43
N: number of nodes = 30
V(G): cyclomatic complexity = 15

State Transition Table Discussion

State Table

Current State	Trigger Event	Next State
LAMP_OFF_DAYLIGHT	LIGHT_SENSOR_DARKENED	LAMP_OFF_NIG
LAMP_OFF_DAYLIGHT	MANUAL_SWITCH_ON	LAMP_ON_FULI
LAMP_ON_FULL_BRIGHTNESS	MANUAL_SWITCH_OFF	LAMP_OFF_DAY
LAMP_OFF_NIGHTIME	LIGHT_SENSOR_LIGHTENED	LAMP_OFF_DAY
LAMP_OFF_NIGHTIME	MANUAL_SWITCH_ON	LAMP_ON_NIGH
LAMP_OFF_NIGHTIME	MOTION_DETECTED	MOTION_DETEC
LAMP_OFF_NIGHTIME	SECURITY_ALARM_TRIPPED	INTRUSION_DE
LAMP_ON_NIGHTIME_BRIGHTNESS	MANUAL_SWITCH_OFF	LAMP_OFF_NIG
LAMP_ON_NIGHTIME_BRIGHTNESS	LIGHT_SENSOR_LIGHTENED	LAMP_ON_FULI
LAMP_ON_NIGHTIME_BRIGHTNESS	SECURITY_ALARM_TRIPPED	INTRUSION_DE
MOTION_DETECTED	MOTION_DETECTION_TIMER_EXPIRED	LAMP_OFF_NIG
MOTION_DETECTED	LIGHT_SENSOR_LIGHTENED	LAMP_ON_FULI
MOTION_DETECTED	SECURITY_ALARM_TRIPPED	INTRUSION_DE
MOTION_DETECTED	MANUAL_SWITCH_OFF	LAMP_OFF_NIG
INTRUSION_DETECTED.LAMP_ON	LAMP_TIMER_EXPIRED	INTRUSION_DE
INTRUSION_DETECTED.LAMP_OFF	LAMP_TIMER_EXPIRED	INTRUSION_DE
INTRUSION_DETECTED	ALARM_CLEARED	LAMP_OFF_NIG
INTRUSION_DETECTED	LIGHT_SENSOR_LIGHTENED	LAMP_OFF_DAY

Explanation:
To test the subscribe and unsubscribe methods, which are not part of the state machine diagram but are crucial to the Observer pattern implementation, we created specific test cases that verify:

- When a new observer subscribes, it should receive state updates
- When an observer unsubscribes, it should no longer receive state updates
- Multiple observers can subscribe simultaneously and all receive updates
- The system handles invalid subscription attempts properly
- The system handles invalid unsubscription attempts properly

Fault Locations

Defect ID	Description	Location	Fix
1	unsubscribe method incorrectly adds observers instead of removing them.	LightControllerStateMachine.java	this.observers.add(obs); with this.observers.remove(obs);.
2	updateObservers method calls updateLightState multiple times unnecessarily.	LightControllerStateMachine.java	Remove duplicate calls to obs.updateLightState(state);.
3	State change condition in signalAction always evaluates to true.	LightControllerStateMachine.java	Replace if (stateChange=true) with if (stateChange).
4	NewLightControllerStateMachine methods are unimplemented.	NewLightControllerStateMachine.java	Implement all methods to match the behavior of LightControllerStateMachine.

New Implementation

Control Flow Graph

[START] → [Initialize State Machine]
  ↓
[Get Current State and Request]
  ↓
[Look up transition in transition table]
  ↓
[Transition Found?] → [NO] → [END]
  ↓ [YES]
[Execute Exit Actions]
  ↓
[Update Current State]
  ↓
[Execute Entry Actions]
  ↓
[Update Observers]
  ↓
[END]

Calculation
V(G) = E – N + 2 = 9 – 8 + 2 = 3
E: number of edges = 9
N: number of nodes = 8
V(G): cyclomatic complexity = 3

Discussion

Why might using the control flow graph be inadequate to verify the operation of the class for the initial code?
Using the control flow graph for verifying the initial code is inadequate because it only captures the structural aspects of the code, but not the semantic meaning or purpose of the state transitions. The security light system's behavior is fundamentally state based, where each state has specific entry/exit actions and allowed transitions. The control flow graph doesn't convey the intended relationship between states or ensure that all valid state transitions are properly implemented. It also does not guarantee that proper exit and entry actions are performed during transitions, which are critical for this system's correct operation.

What was different about defining tests from the control flow graph versus defining tests based upon the state machine definition?
When defining tests from the control flow graph, we focused on ensuring each branch in the code was exercised, which resulted in tests that traced execution paths rather than verifying behavior. This approach might miss logical errors where a path exists but leads to an incorrect state or perform incorrect actions. In contrast, when defining tests based on the state machine definition, we focused on verifying each possible state transition and the associated entry/exit actions. This approach ensured that the implementation matched the specification's behavioral requirements regardless of the specific control flow implementation. State based tests are more meaningful for verifying the system's behavior from the user's perspective.

Which method has a lower cyclomatic complexity associated with it, the initial provided code or the code you developed?
The code we developed has a significantly lower cyclomatic complexity of 3 compared to the initial provided code of 15. This reduction occurs because the new implementation uses a state transition table approach rather than nested conditional statements. By storing the valid transitions and their associated actions in a data structure, the control flow becomes much simpler, with fewer decision points and branches.

Which method is easier to test, the initial method or the new method you have developed?
The new method we developed is easier to test because its structure is simpler and more directly aligned with the state machine specification. With fewer branches and a clear separation between the transition logic and the actions, it's easier to inject mock objects and verify the correct sequence of calls. The table-driven approach means that adding test cases for new transitions doesn't require changes to the core logic, making the tests more maintainable. Additionally, the

new implementation's lower cyclomatic complexity means fewer test cases are needed to achieve full branch coverage.

Which method is stylistically more like the code you would write if you were developing this program from scratch either earlier this year or last year as a freshman?
Earlier in our academics, we would have likely written code more like the initial implementation with nested if/switch statements for each state and condition. This approach is more direct and requires less abstraction, making it seem simpler at first. As freshmen, we did not have experience with design patterns like state patterns or table-driven designs that could simplify the implementation. The initial implementation follows a more procedural style that beginners often find more intuitive before they learn about software design principles and patterns.

How effective would your tests be if you used input domain analysis to test this program?
Input domain analysis would be less effective for testing this state-driven system compared to state transition testing. The system's behavior depends not just on the current input but on the current state, making it difficult to partition inputs independently of the system state. While you could identify equivalence classes for inputs, these inputs have different effects depending on the current state. Boundary value analysis also has limited applicability since the inputs are discrete events rather than values within ranges. Input domain testing might help identify missing input validations but would not verify the state transitions that are central to the system's behavior.

How difficult was it to verify the observers in this system?
Verifying the observers in this system presented moderate challenges. We needed to create mock observers to verify they received notifications when states changed and that they stopped receiving notifications after unsubscribed. The main challenge was ensuring that the timing of notifications was correct relative to the state changes, especially during complex transitions like those involving timers. Additionally, the testing required careful setup to isolate the observer mechanism from the rest of the state machine logic.

Things Gone Right / Things Gone Wrong

Things that went right in this lab were successfully identifying all state transitions from the UML state diagram and creating a comprehensive state transition table to guide testing. We also effectively used mock objects to isolate components for testing, allowing us to test other functionalities that are unreachable for the most par.t
Things that went wrong in this lab were initially missing some subtle state transitions in the diagram. We had difficultly correctly mocking the timer mechanism in some test cases and found it challenging to verify the exact sequence of method calls during complex transitions. The initial tests were too dependent on implementation details rather than behavior.

Conclusion

What have you learned from this experience?
Through this lab, we have gained valuable insights into state-based testing and the importance of formal specifications for complex behaviors. We learned that while control flow testing is useful for verifying code structure, it is often insufficient for systems with complex state behavior. The

exercise reinforced the value of design patterns for creating maintainable code. We also learned practical strategies for using mock objects to isolate components during testing, which was essential for verifying the interactions between the state machine and its observers/timers. We also learned how a table-driven approach can dramatically reduce complexity in state machine implementations while making the code more maintainable and testable.

What improvements can be made in this experience in the future?

- More detailed introduction to state machine testing techniques before starting the lab
- Include examples of well-implemented state machines using different design patterns for comparison
- Include studying of automated state machine generation tools or frameworks