SWE2721 Lab 12: A State Driven Security Light

1 Introduction

Thus far in lab, you have done ad hoc testing in which you were given a series of use case scenarios and asked to test a software package based on those scenarios. You have also used boundary value analysis and equivalence partitioning, as well as mock objects. Now it is time to move into a different realm of test design, namely test design from a formal specification.

A finite-state machine is a commonly used technique to design digital logic and computer software. It is a behavior model composed of a finite number of states, transitions between those states, and actions which occur upon entry and exit from states.

In this lab, you will be analyzing a formal state machine and developing test cases to ensure that the implementation of that state machine is correct.

2 Lab Objectives

- Analyze and construct test cases from a state diagram.
- Perform state testing on an existing class, noting any found defects.
- Use Mock objects and Mock Object tools to verify correct operation of an existing software package.
- Implement source code to a specification and verify the operation using existing tests.

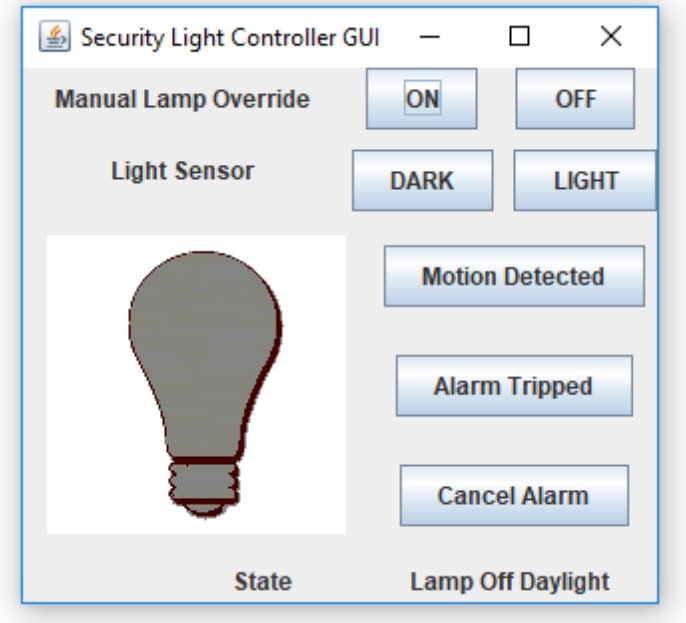

Figure 1 The UI for the completed software.

3 Deliverables / Submission

When you have completed your lab assignment, tag your final version (which should be on the trunk) with the label LabSubmission1.0. Your code test cases should be thoroughly commented and build cleanly without warnings, and all code should pass the submitted tests.

In addition, submit one report in **pdf format**. The report should contain the following contents:

1. Title Page
   a. Names of all partners
   b. Date
   c. Assignment Title
   d. Repository Link
2. Introduction
   a. What are you trying to accomplish with this lab?
      i. (This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.)
3. Diagrams
   a. Include the sketch of the control flow graph for the initial provided signalAction method.
   b. Show the calculation you used for the cyclomatic complexity of the method.
4. State Transition Table Discussion
   a. Include a state table showing the given states and conditions, as well as what methods are expected to be invoked when a state change occurs.
   b. Explain how you tested the other methods, such as the subscribe and unsubscribe, given that they are not directly part of the state machine.
5. Fault Locations
   a. Did you find any locations in your testing, and if so, how did you fix them? These do not need to be implemented in the defect tracking system but should be clearly commented in the source code as to what was wrong and why it was fixed.[1]
6. New Implementation
   a. Include a control flow graph for the new implementation of the signalAction method.
   b. Calculate the cyclomatic complexity of the new implementation.
7. Discussion
   a. Why might using the control flow graph be inadequate to verify the operation of the class for the initial code?
   b. What was different about defining tests from the control flow graph versus defining tests based upon the state machine definition?
   c. Which method has a lower cyclomatic complexity associated with it, the initial provided code or the code you developed?
   d. Which method is easier to test, the initial method or the new method you have developed?
   e. 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?
   f. How effective would your tests be if you used input domain analysis to test this program?
   g. How difficult was it to verify the observers in this system?
8. Things gone right / Things gone wrong.
   a. This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.
9. Conclusions
   a. What have you learned with this experience?
   b. What improvements can be made in this experience in the future?

If you have any questions, consult your instructor.

[1] Note: There are not as many in this codebase as in previous labs. They are much more buried to better reflect the real world.

4 Domain background

The system you are to verify represents a security light. The security light has a light sensor which detects if it is night or day. During the day, the motion sensor is not active, and the light will not turn on. However, at night, the system will look for motion, and if motion is found, the lamp will be turned on for 30 seconds. If at night the security alarm is tripped, the light will flash on and off until it is cleared by the user. The system also has a manual override switch which will turn the lamp on at any time.

A UI for this program has been developed for prototyping purposes, as is shown in Figure 1. It allows the user to simulate inputs and monitor the behavior. Note that for the purposes of this lab, the flash rate and 30 second timeout are sped up a bit. Formally, the behavior for this system is specified using a UML state chart. This is shown in Figure 2. Note specifically the entry and exit actions for each of the states.

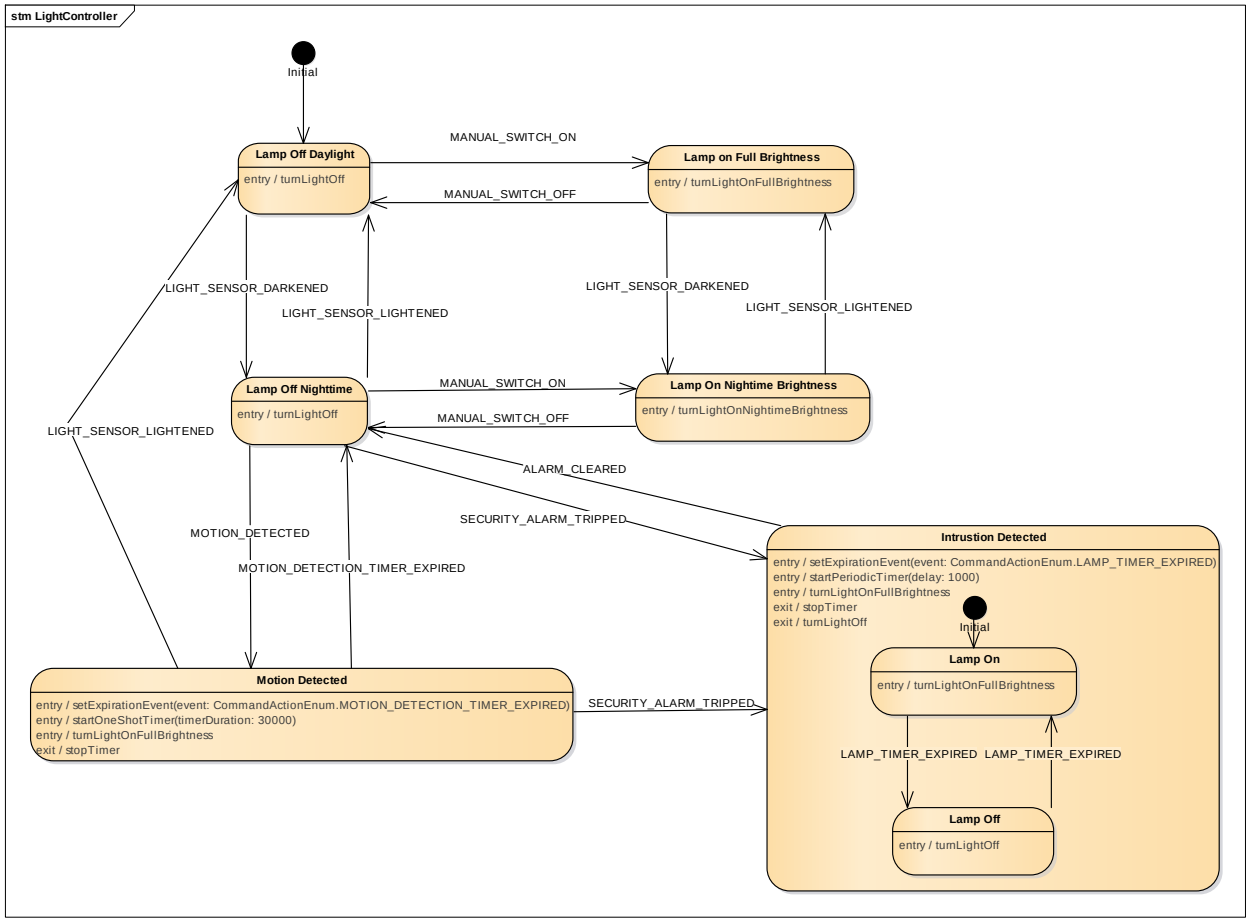

Figure 2 UML state diagram for program behavior. (Note: For testing purposes, the execution of the timers has been sped up by a factor of 6, meaning that a timeout of 30 seconds will result in a timeout of only 5 seconds. This should not impact state testing or interaction testing but will change the current execution of the full program.)

The system itself has been implemented as is shown in the class diagram of Figure 3. Methods are fully described in the Javadoc.

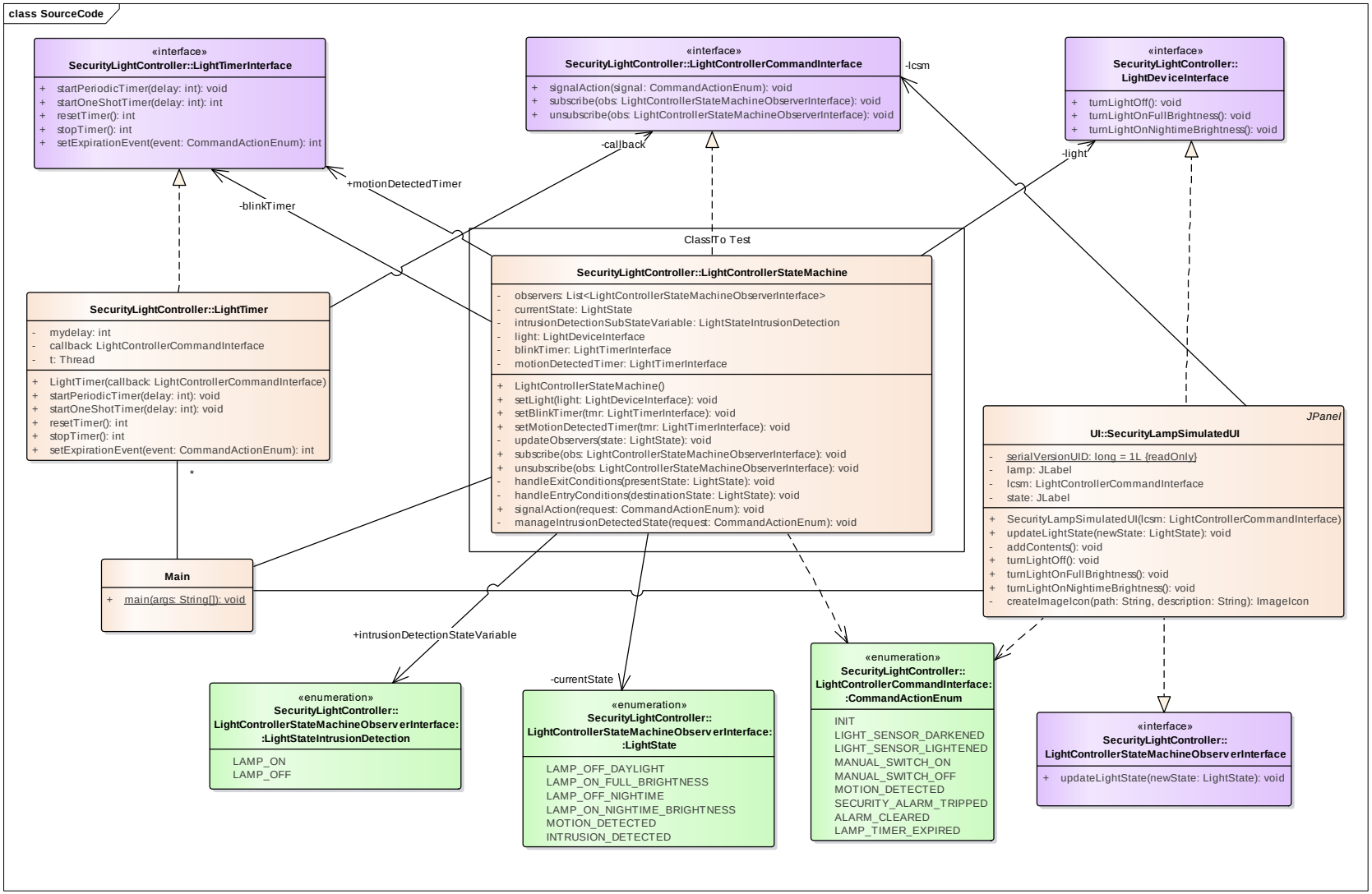

Figure 3 UML class diagram for the system.

5 Assignment Specifics

For this assignment, you will work in a team. At the highest level, you will be responsible for debugging the implementation you are provided, as well as developing a new implementation to the same specification that you are provided with but using different implementation techniques. The new implementation should match the behavior of the initial class but should be implemented in a different fashion. This implementation should ideally be simpler and easier for you to work with. However, the main variables should basically be the same in order to facilitate appropriate testing.

To begin, you should also develop a set of tests that are designed based upon the state machine, ensuring that all state transitions are exercised, and that the behavior of the class is correct. These tests should correctly verify the operation of the state machine as well as all other methods in the class. (Hint: The class uses an Observer pattern as part of its implementation. The observer pattern is something you should have seen by now in your Cloud and Design Patterns course.) In doing this verification, you may need to fix a few bugs that are present in the state machine. You only need to test the LightControllerStateMachine class. All others are assumed to be fine. You will need to use TestNG as well as a Mock Object tool. (Hint: You may want to draw sequence diagrams from the state chart to determine the correct interactions and message sequences between objects.) You may also need to use reflection as well and / or output redirection.

You will need to design your tests to ensure that you obtain 100% transition coverage within your state machine, correcting any defects in the code found as you perform your testing. You also should specifically test the subscribe and unsubscribe methods, making sure that the Observer pattern is being handled properly. These tests should be in the group “studentsmtests”. (Note that there are about four defects in the code that your tests should find.)

In tandem with this, the developer should be developing a new implementation of the LightControllerStateMachine, named simply NewLightControllerStateMachine, which will have the same behavior but be implemented in a different manner. The developer writing this code should be able to use the test cases developed in the previous step to verify their state machine is working properly. However, this implementation should ideally be simpler.

To aid the developer, a template file has been created and a second main has been created for this code. The developer is responsible for making certain that all the tests pass and that their code works properly. You’ll need to pay careful attention to the interface and any possible variables that are non-private in scope. When copying the tests cases and setting them up to run against the new state machine, these tests should be placed in the group “studentnewsmtests”.

When the developer is finished, sketch two control flow graphs, one for the signalAction method in the initial code and one for the method in the new code. From these control flow graphs, calculate the cyclomatic complexity of the methods.

When finished, there should be two different implementations for the state machine which have identical behaviors associated with them, as well as two test files. The first test file implements transition testing for the base state machine, meaning that the behavior of the state machine is tested by ensuring that all transitions were taken on the state machine. The second file implements transition testing for the new state machine, and overall, should have minimal differences with the second file other than the name of the class under test and the groups for the tests, as the state machine that is being tested is identical.