Software Design and Programming
Software and Programming III

Exercise sheet — S.O.L.I.D.

Session: 2017

Purposes of these exercises

To examine the five principles embodied by S.O.L.I.D. via an appropriate example.

Preamble

The questions on this sheet revolve around the following set of use cases involving a set of sensors:

- The system should support any number of sensors.
- The system should support different types of sensors.
- The system should poll all sensors to see if any are triggered (an alarm is raised).
- The system should also check for the battery percentage of the sensors (different types of sensors drains faster).

You are given the following specifications:

Interfaces

- `Sensor` — This interface defines methods for all sensors to implement.

  `isTriggered():` returns true/false for whether the sensor is triggered or not. Different sensors have different rules applied to them.

  `getLocation():` returns a description of the location of the sensor such as *Lobby 1st floor* or *In the auditorium.*

  `getSensortype():` returns a textual description of the sensor type such as `"Fire sensor"` or `"Smoke sensor"`.

  `getBatteryPercentage():` Returns a number between 0-100 where 0 is empty and 100 is fully charged.

Classes

`FireSensor`: This sensor implements the `Sensor` interface but has no logic yet.

`SmokeSensor`: This sensor implements the `Sensor` interface but has no logic yet.

`ControlUnit`: This is the starting point for the alarm system. It is the main entry point for polling sensors and controlling the system.

Exercises

1. Implement the `FireSensor` methods.
   - 5% of the time it is called, it raises an alarm
   - Drains 10% battery between each poll
2. Implement the `SmokeSensor` methods.
   - 10% of the time it is called, it raises an alarm
   - Drains 20% battery between each poll
3. Examine the `ControlUnit.pollSensors()` method. What are its current responsibilities? (No need to do anything, just make sure you find all the responsibilities before you continue). Ask an instructor if you're not sure.
4. Instead of *newing up* the various sensors when we call `pollSensors()`, we want to pass a collection of sensors in to the `ControlUnit` somehow. However, we don't want to pass the sensors in when we are polling. When we poll sensors, the control unit should be configured with all of the required sensors. (Hint: Dependency Inversion Principle).
5. Investigate the `pollSensors` method again, as you did in the previous exercise. What are the responsibilities now?
6. A new use case! This is no longer a alarm system for only detecting hazards, it should now also include security such as motion and heat sensors. However, these sensors don't run on a battery so one of the `Sensor` interface methods is suddenly redundant for a whole set of sensors. Which method is this and what SOLID principle does this break?
7. Following the principle you figured out in the previous exercise, solve the problem by following the presentation slide hints on this principle.
8. Create a new `MotionSensor` sensor, which inherits from the `Sensor` interface. These new security sensors should be polled separately from the hazard sensors. This requires a way to distinguish between the two sensor categories. Make changes to the `Sensor` interface to accommodate this.
9. Security sensors should only be polled at night between 22:00-06:00. This is the same for all security sensors. Since we don't want to mix security sensor and hazard sensor behaviour in the same polling mechanism, we decide to make use of inheritance and create a new `SecurityControlUnit` which enhances the existing `ControlUnit`. Our

intention is to pass in the security sensors through the parent constructor and then implement a rule that checks the current time and if it's between 22:00-06:00, we run the `super.pollSensors()` method which already does the heavy lifting for us.

10. Which SOLID principle are we maintaining/not breaking by doing this?

11. Create the `SecurityControlUnit` by enhancing `ControlUnit`

12. Implement the *timecheck* rule and poll the sensors.

13. We want to test that an alarm is raised, and also implement different alarm notification methods (For example: automatically call the fire department, call the owner's phone, and trigger the control unit alarm siren). Plan how you can extract the announcement logic and:

    a) Trigger many different announcement strategies

    b) Test that the different strategies was triggered

    Hint: Dependency Inversion Principle and the *Strategy Design Pattern* (`https://en.wikipedia.org/wiki/Strategy_pattern`). We are jumping a bit ahead of ourselves here because we haven't covered *Design Patterns* yet so don't be afraid to ask for help if you need it.

14. Implement the plan you made in the previous exercise.