Spring 2012 Software Engineering Qualifying Exam

This is an open book exam. Basic calculators are allowed, but no computers or other devices that have communication capability are allowed; this means that cell phones cannot be used as calculators. There are a total of 100 points on this exam. Be sure to show your work in case your answer may deserve partial credit, but do not add spurious or frivolous content in hopes that something you say might be right. Content in an answer that is irrelevant to the question may cause point deductions.

Problem Description (all questions use this description and code)

You and your team are setting out to build a "smart home" system. A smart home has a computer system that uses devices throughout the house to sense and control the home. The two basic smart home device types are *sensors* and *controls*. These are installed throughout the house and each has a unique name and ID, location, and description. The house has a layout (floorplan) image, but is also managed as a collection of rooms. Device locations are rooms, and per-room views and functions must be supported.

Sensors are of two types: queriable and event announcer. For example, a thermostat is a queriable sensor: the computer application sends out a query and the thermostat replies with the currently measured temperature. An example of an event announcer is a motion sensor: it must immediately announce the event that motion was sensed, without waiting for a query. Controls actually control something, like the position of a window blind, the state of a ceiling fan, or whether a light is on or off. However, all controls are also queriable sensors; querying a control results in receiving the current settings of the control.

Device data (received from a sensor or sent to a control) depends on the type of device, and could as simple as one boolean flag (e.g., is door open or closed, turn light on or off), or could be a tuple of data fields (e.g., the current temperature and the thermostat setting, or fan on/off and speed). Each field (even just one) has a name.

The system will provide a "programming" environment using something like a scripting language for the user to customize their smart home environment. It should also allow graphical browsing of the current state of the house, and direct manipulation of controls (overriding any scripting control). The system must also provide some remote web-based access for use when the homeowner is traveling.

Below is some partial code from a potential system implementation; some of the methods are incomplete and have comments that act as placeholders for code that would be in that place.

```
public interface SensorObserver
{
   public void sensorChanged(Sensor changedSensor);
}

public interface Sensor
{
   public String getID();
   public boolean setID(String id);
   public boolean addObserver(SensorObserver so);
   public boolean deleteObserver(SensorObserver so);
   public SensorValue getCurrentValue(String valueName);
}

public class Thermometer implements Sensor
{
   private vector<SensorObserver> observers;
   private String id;
   private SensorValue lastReading;

   public Thermometer() { observers = new vector<SensorObserver>(); }

   public boolean setID(String id) { this.id = id; }
   public String getID() { return id; }

   public boolean addObserver(SensorObserver so) { observers.add(so); }
   public boolean deleteObserver(SensorObserver so) { observers.delete(so); }

   private void notifyChange()
   {
      for (SensorObserver so: observers) {
         so.sensorChanged(this);
      }
   }

   public SensorValue getCurrentValue(String valueName)
   {
      if (!valueName.equals("temperature"))
         return null;
      SensorValue val = /* code here to actually query the physical sensor */;
      lastReading = val;
      return val;
   }

   public void checkSensor()
   {
      SensorValue oldval = lastReading;
      SensorValue val = getCurrentValue("temperature");
      if (!val.equals(oldval))
         notifyChange();
   }
}
```

```
public class Rule implements SensorObserver
{
   private SmartVector mySensors;  // Java vector with method "find(int sensorID)"
   private Control control;        // a rule activates only one control
   private ParseTree parsedRule;

   public ErrorCode setRule(String RuleScript)
   {
      ErrorCode error = ERROR.AllOK;  // default is no error
      //
      // parse script string and verify syntax; identify
      // sensors, make sure they are valid, and attach to them
      // as an observer. For each sensor token in the script, we do:
      //
         sensor = System.findSensor(id);
         if (sensor != null) {
            mySensors.add(sensor);
            sensor.addObserver(this);
         } else {
            error = ERROR.UnknownSensorID;
            break;
         }
      // finally, after all is done, we return our status
      return error;
   }

   public void sensorChanged(Sensor changedSensor)
   {
      evaluateRule(changedSensor);
   }

   private void evaluateRule(Sensor changedSensor)
   {
      // walk through the rule as stored in parsedRule,
      // each time we come to a sensor value being referred to
      // (as id.valueName) we do:
         if (id == changedSensor.getID())
            tmpSensor = changedSensor;
         else
            tmpSensor = mySensors.find(id);
         value = tmpSensor.getCurrentValue(valueName);
         // value is then used as appropriate in the rule
      // after rule is fully evaluated we do:
      if (ruleResult == true) {
         // fire necessary actions on control
      }
   }
}
```

Questions

[20pts] 1. Create a domain model for this problem, modeling the domain information structure in a UML class diagram. Include classes and their relevant attributes, and relationships between classes, including name, cardinality, and direction where appropriate. Remember that some of what is described above might not be relevant to a *problem domain* model.

> *Answers will vary, but need to capture the composition of rooms in a house, and the inheritance of queriable and event sensors, and controls.*

[10pts] 2. Pick one software development process style (e.g., waterfall, spiral, or others) that you would prefer your team to use, and **explain** why. What benefits would this process give you? What assumptions are you making about your team? What would this process style be good at, and what would it be not so good at? (Note that it is the explanation that is the important part of the answer, and is where the problem points will be earned.)

> *Answers will vary, and points will be given on explanation.*

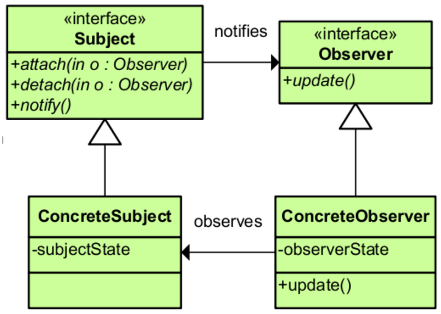
[10pts] 3. What are two potential risks that could jeopardize the success of your project? **Explain** why you chose them.

> *Answers will vary, maybe in relation to the answers in 2. Interfacing with the hardware could be one, unknown deployment environments, dynamic configurations, inexperience with a reactive control system, all could be potential risks.*

[10pts] 4. Pick one software architectural style that you think would be useful for at least part of this system and **explain** how it would help. You can choose any known architectural style from any reference, **except** client-server.

> *Probably the best answer would be some sort of loosely coupled style, such as publish-subscribe and similar event or message based styles. Points will be earned by the explanation given.*

Problems 5 and 6 make use of the Observer design pattern:



(from Design Pattern Card, (C) 2007 Jason McDonald, www.mcdonaldland.info)

[10pts] 5. In the code above the Observer pattern is used, yet some of the suggested implementation details in the pattern figure are changed. **Describe** how the classes in the code above map to the classes in the figure, **explain** where the code differs from the suggested pattern, and **explain** why the differences might be a good idea.

> *Minor differences are that the method names are tailored to the specific domain. A significant difference is that the notifyChange() method is private rather than public, and even does not appear in the Sensor interface class (which is the Subject class). In the code this method is invoked internally, and there is no reason for external code to be able to invoke it, so it makes sense to make it private. The notify operation can be something the Subject is entirely responsible for, and so it does not necessarily need to be public.*

[10pts] 6. In looking at the code above it would seem that many sensors might have very similar code in parts of their class when implementing the Sensor interface. Thus we might consider making the Sensor interface an abstract class. **Explain** the benefits and potential drawbacks. As part of your explanation, describe which code in the Thermometer class would be moved into the Sensor class.

> *All of the observer handling is completely generic, as are the get/set ID methods, and so it makes no sense to require each implemented sensor to duplicate this code. It would be better moved into an abstract Sensor class, and then every sensor could inherit it and not rewrite it. Drawbacks include a tighter coupling between the sensor classes, especially with the system written in Java where there can be only one "extends" inheritance relation. The means that our sensors will not be able to extend any other class, and so we better make sure we get the correct base class implemented.*

[10pts] 7. Consider sensor values and the variables that hold sensor values (some for very short times, some for longer times). **Explain** in terms of dataflow coverage testing why from an individual variable view, the code shown here is fairly easy to test for dataflow coverage.

> *Most of the variables that hold sensor values are local variables and thus their data flow would be fairly simple to cover. The only one that is somewhat persistent is "lastReading", and in the given code it is defined (written) in only one place (in getCurrentValue()) and used in only one other place (in checkSensor()), so its dataflow is pretty simple. It would be straightforward to achieve full coverage on all of the def-use pairs in dataflow testing.*

[10pts] 8. If we take a more abstract view by thinking of "sensor values", Then the code might actually be considered very hard to test. **Explain** why. (Hint: view the code from an object-oriented viewpoint rather than just a static "lines of source code" view, and consider what would be happening in a fully deployed system.)

> *In thinking more abstractly about sensor values, this is where the code gets quite a bit more complex. The loose coupling between sensors and rules with the Observer pattern means that there are no hard-coded data flow paths from the individual sensor objects and their values to the rule objects; it is all implicit through the observer attachment and event announcement. Furthermore, it is likely that there are **lots** of sensor and rule objects, and so the interaction will likely be complex and unpredictable. Designing and executing tests that might actually cover the expected deployment configurations will be very hard!*

[10pts] 9. For an overall view of this part of the system, pick one of the statements "this code is easy to test" or "this code is hard to test", and justify why your selection is the proper view of this code.

> *EASY: Since the code in each class is quite straightforward, testing the correct operation of each class should be pretty easy, and thus we can*

have good confidence that the objects will do what they are supposed to, individually. If our design is sound, and using design patterns such as Observer should help, then even if the composition of the objects is complex, we should be able to scale up and trust our implementations. If we test some basic interactions, such as a sensor informing multiple rules, and a rule using multiple sensors, then we should be able to trust our system to scale up. The hard part might be in rule evaluation, but again here we can use formal tools such as parser generators to help ensure that we create a proper rule language and that we parse and evaluate it correctly.

HARD: The system execution will in fact be very dynamic, with unknown object relationships and interactions, and thus even extensive testing will fail to reasonably cover or explore the potential object interactions that this system will have. On top of this is the fact that the system behavior is not pre-determined but is under programmatic control of the user means that it is likely impossible to be confident whether all scriptable behaviors will be implemented correctly. A great variety of tests will be needed, especially with complex rules that interact with the same group of sensors. What if rules conflict? Or if they feed back on each other in the way they affect sensors? Since controls are also sensors, what if they are used as sensors for intertwining rules? There are many such issues that make this system very hard to test and ensure that it will function soundly.