Object-Oriented Analysis and Design

Objectives

The objectives of this chapter are to identify the following:

- Define a weather monitoring station.
- Analyze the system to determine its key classes.
- Design those classes.
- Create basic UML diagrams used to represent class relationships.
Problem Statement

Grady Booch provides this example of a software system in need of analysis and design\(^1\):

“The system shall provide automatic monitoring of various weather conditions. Specifically, it must measure:

- Wind speed and direction
- Temperature
- Barometric pressure
- Humidity

The system shall also provide the following derived measurements:

- Wind chill
- Dew point temperature
- Temperature trend
- Barometric pressure trend

The system shall have a means of determining the current time and date, so that it can report the highest and lowest values of any of the four primary measurements during the previous 24 hour period.

The system shall have a display that continuously indicates all eight primary and derived measurements, as well as the current time and date. Through the use of a keypad, the user may direct the system to display the 24-hour high or low value of any one primary measurement, together with the time of the reported value.

The system shall allow the user to calibrate its sensors against known values, and to set the current time and date.”

Hardware Assumptions

In addition, Booch also makes some basic hardware assumptions:

- We will use a single-board computer (SBC) with a 486-class processor.
- Time and date are supplied by an on-board clock, accessible via memory-mapped I/O.
- Temperature, barometric pressure, and humidity are measure by on-board circuits (with remote sensors), also accessible via memory-mapped I/O.

Wind direction and speed are measured from a boom encompassing a wind vane (capable of sensing wind from any of 16 directions) and cups (which advance a counter for each revolution).

User input is provided through an off-the-shelf telephone keypad, managed by an on-board circuit supplying audible feedback for each key press. Last user input is accessible via memory mapped I/O.

The display is an off-the-shelf LCD graphic device, managed by an on-board circuit capable of processing a simple set of graphics primitives, including messages for drawing lines and arcs, filling regions, and displaying text.

An on-board timer interrupts the computer every 1/60 second.

The kind of hardware is chosen to simplify the software development process and to alleviate much of the low-level coding that would otherwise be required.

The Goals of Object-Oriented Analysis

The analysis phase of an object-oriented system has several key objectives:

- Identify the key actors within the system. An actor is an element within the system that performs some task, either actively or passively. These actors will usually end up becoming the classes of the system.
- Segregate the roles and responsibilities of each actor. This helps us begin to see how the actors will inter-relate during system execution and to make sure all of our required functionality is accounted for. These roles and responsibilities generally end up becoming methods available on the classes representing the various actors.

Determining the System Boundary

Before embarking on the analysis and design phase, we must first analyze the system boundary to determine what we need to build versus what can be bought or what is already available to us in pre-written class libraries.

Once the system boundary has been established, we can begin a class analysis. During a class analysis we seek to identify the important classes within the system. This need not be an exhaustive list (we will be identifying classes all the way through deployment), but it should serve to capture the essential elements of the system.

We can find these classes in numerous locations:

- Tangible things
- People or roles
- Places or locations
- Interactions
Useful Techniques for Class Analysis

There are several useful techniques that can be employed to facilitate class analysis. Some of these include:

- Data dictionary
- Informal requirements
- CRC cards
- Use-Case Analysis

Data Dictionary

A data dictionary is generally maintained by a database administrator. It outlines, at a fairly detailed level, what the entities are that are being stored in the database, the attributes of those entities, and the relationships that the entities maintain to one another.

Entities from this data dictionary may prove useful as classes. Another advantage to this technique is that many of the attributes and relationships have already been defined.

Informal Requirements

We can sometimes glean classes from informal specifications. For instance we might identify nouns as either classes or attributes and verbs as operations. This method is sometimes useful for a quick analysis, but for any in-depth analysis, I would recommend either CRC cards, or use-case analysis.

CRC Cards

Another way of keeping track of the actors, their responsibilities, and their interrelationships, is by using CRC cards.

A CRC card is nothing more than an index card. It identifies a Class, that class’ Responsibilities, and the other classes with whom it Collaborates. While usually not suitable as a final deliverable to a client, CRC cards can be a good first step in identifying the main components of the system.

Use-Case

One good method of doing systems analysis and design is called use-case analysis. Essentially this is nothing more than a “story board” of various scenarios that
the systems is meant to handle and the method in which it will handle those scenarios. The following is an example from the Booch text:

Setting the time and date.

1. User presses the select key.
2. System displays selecting.
3. User presses any one of the keys time or date; any other key (except run, wind speed, temperature, pressure, or humidity) will be ignored.
4. System flashes the corresponding label; display also flashes the first field of the selected item (hours for time, and month for date).
5. User presses left or right keys to select another field. User presses the up or down keys to raise or lower the value of the selected field.
6. Control passes back to step 3 or 5.

Time-Date Class

The first class we’ll look at is the TimeDate class. This class is responsible for acquiring, calibrating, and displaying the current time and date.

While it is possible to capture this class using text, it will be much more efficient to document it using a simple UML class diagram. This diagram describes the class using some of its key properties and methods. It also allows the analyst to indicate class-level versus object-level properties and methods as well as the access (public, private, or protected) of each member. For the analysis phase we can dispense with much of this detail and make some preliminary assumptions.

A class diagram consists of a box divided into three horizontal layers. In the first layer we write the class name. In the second we place the class attributes, and in the third layer we place the class methods. For example, the class diagram for the TimeDate class might appear as the one to the left.

This diagram is an example of an analysis diagram; it doesn’t contain too many details. However, for a design diagram we embellish this diagram a bit. Specifically we begin to add types and access specifiers.
For instance, the time and date might be represented as long integers. Each of the class methods will have some set of parameters and return types. More importantly, we begin to identify the visibility of each class member by providing an indication of its access specifiers. An example of a design class diagram is seen to the right. Notice that to the right of each member a type is listed. This represents the type of data used to implement each of the attributes and the type of data returned by the methods.

Also notice that next to each member is a symbol, either ‘+’ or ‘-‘. These are access specifiers. A minus sign (-) indicates that the member is private to the class. A plus sign (+) indicates that the member is public. A pound sign (#) would indicate that the member is protected.

Note that these specifiers map to the corresponding specifiers in both C++ and Java.

**Temperature Sensor**

The next class describes the temperature sensor. This sensor should be able to report on:

- the current temperature
- the high and low temperatures for the prior 24 hour period
- any temperature trends

The class for this sensor is given as:

<table>
<thead>
<tr>
<th>Class:</th>
<th>Temperature Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes:</td>
<td>temperature trend</td>
</tr>
<tr>
<td>Operations:</td>
<td>currentTemperature</td>
</tr>
<tr>
<td></td>
<td>setLowTemperature</td>
</tr>
<tr>
<td></td>
<td>setHighTemperature</td>
</tr>
<tr>
<td></td>
<td>getTrend</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TimeDate</th>
</tr>
</thead>
</table>
| +currentTime:
| +currentDate:
| +setFormat:
| +setHour:
| +setMinute:
| +setSecond:
| +setMonth:
| +setDay:
| +setYear:

Note that these specifiers map to the corresponding specifiers in both C++ and Java.
Pressure Sensor

The next class describes the pressure sensor. This sensor should report on:

- the current pressure
- the high and low pressures for the prior 24 hour period
- any pressure trends

The class for this sensor is given as:

Class: Pressure Sensor
Attributes: pressure
trend
Operations: currentPressure
            setLowPressure
            setHighPressure
            getTrend

Trend Sensor

Notice that both of these classes must report on the trends of their various measurements. In this case we choose to pull out this common functionality into a separate class and use inheritance to make this functionality common to the two classes. Thus we end up with a new class, Trend Sensor, which is able to:

- report on trends

The class for this sensor is given as:

Class: Trend Sensor
Attributes: trend
Operations: getTrend

Since we have now pulled the common functionality of trend sensing from the other two classes, they become somewhat simplified:

Class: Temperature Sensor
Attributes: temperature
Operations: currentTemperature
            setLowTemperature
            setHighTemperature

Class: Pressure Sensor
Attributes: pressure
Operations: currentPressure
            setLowPressure
            setHighPressure
Humidity Sensor

The humidity sensor is similar to the other sensors. This sensor should be able to report on:

- the current humidity
- the high and low humidities for the prior 24 hour period

The class for this sensor is given as:

**Class:** Humidity Sensor  
**Attributes:** humidity  
**Operations:**  
  - currentHumidity  
  - setLowHumidity  
  - setHighHumidity

History Sensor

As before, we can see that the humidity sensor, like the temperature and pressure sensors exhibits the ability to report on the highest and lowest values of their respective measurements within the prior 24-hour period. This similarity prompts us to abstract the sensor classes still further. To do this we create the History Sensor. This sensor is able to report on:

- the low and high values encountered within a 24-hour period  
- the date and time of those measurements

The class for this sensor is given as:

**Class:** History Sensor  
**Operations:**  
  - lowValue  
  - timeOfLowValue  
  - highValue  
  - timeOfHighValue

Notice that unlike the other classes, the History Sensor has no attributes: its only members are methods. This is a common phenomenon and allows us to observe the following “best practice:”

“Program to an interface, not an implementation.”

We will discuss this more later.
Wind Speed Sensor

The wind speed sensor is similar to the other sensors. This sensor should be able to report on:

- the current wind speed
- the high and low wind speeds for the prior 24 hour period

The class for this sensor is given as:

Class: Wind Speed Sensor
Attributes: speed
Operations: currentSpeed
           setLowSpeed
           setHighSpeed

Calibrating Sensor

Reviewing the requirements, it appears that we have missed a key point, that is, that most of the sensors must be able to be calibrated to some known value. While this functionality is straightforward, we don’t want to duplicate it if we can avoid it. The obvious solution is to place this functionality into another super-class called Calibrating Sensor.

The calibrating sensor takes most of the common data from the other classes. Each calibrating sensor will be able to report on:

- the current value
- the high and low values for the prior 24 hour period

The class for this sensor is given as:

Class: Calibrating Sensor
Operations: currentValue
           setLowValue
           setHighValue

The Calibrating Sensor is an immediate superclass of the History Sensor. This makes sense since any History Sensor must be able to report on values that were tracked via the functionality provided by the Calibrating Sensor.
Wind Direction Sensor  

The wind direction sensor is the last sensor. This sensor is different from the others in that it requires no history and needs to observe no trends. The Wind Direction Sensor will report on:

- the current wind direction

The class for this sensor is given as:

**Class:** Wind Direction Sensor  
**Attributes:** direction  
**Operations:** currentDirection

Sensor Hierarchy  

As a final unification for all of our sensors, we provide a single abstract class called Sensor. The final sensor hierarchy appears as follows:

```
  Sensor
    / \ 
   /  \  
  Calibrating Wind Direction
       /    
      /     
     Historical
        /   
       /    
      Trend Humidity Wind Speed
        /   
       /    
      Temperature Pressure
```

The display manager, keypad, and timer classes are not as interesting to our discussion, so we will not spend much time on them. Instead we’ll take a look at some ways of performing analysis and design.
Starting the Course Project

You now have enough information to begin analyzing and designing your own course project. For the next phase of your project you need to turn in several UML diagrams outlining some of the core functionality of your system. We will continue our discussion of UML next week.

For this week you should plan on completing:

- Use-case analysis for one of the major functions of your system. The use case should outline, in detail, the steps that will be executed and the results that the system will return.
- Preliminary class analysis and diagrams. These should include the class names, attributes, and operations. It should also show any inheritance that is present. Next week we will see how to show relationships between the classes.