Chapter 11

Data Acquisition: Weather Monitoring Station

For many scientific systems, the automatic collection of data is usually acquired through the use of sensors or devices. This data acquisition typically involves the processing of signals and waveforms to obtain the desired information. The components of a data acquisition system include the appropriate sensors that convert any measured parameter to an electrical signal, which is acquired by data acquisition hardware. Control software is developed that interprets the signals for analysis and display.

Using object-oriented techniques to design a data acquisition system allows us to isolate the hardware that measures and collects the data from the application that then analyzes the information. A robust architecture can be defined to allow for sensors and devices to be added or replaced without disturbing the architecture of the control application. Applying interfaces that act as a skin overlaying the hardware allows for isolation of the measuring devices from the application that processes the information. In this chapter, we provide an example of a data acquisition system, in our case, a Weather Monitoring System. The Weather Monitoring System uses sensors and devices that measure the weather conditions that are analyzed and displayed. This example illustrates an object-oriented solution to a real-time control processing application that provides a reusable component architecture that can isolate the hardware from the application.

11.1 Inception

Our Weather Monitoring System is a simple application, encompassing only a handful of classes. Indeed, at first glance, the object-oriented novice might be tempted to tackle this problem in an inherently non-object-oriented manner by considering the flow of data and the various input/output mappings involved. However, as we shall see, even a system as small as this one lends itself well to an object-oriented architecture, and in so doing exposes some of the basic principles of the object-oriented development process.

Requirements for the Weather Monitoring Station

This system shall provide automatic monitoring of various weather conditions. Specifically, it must measure the following:

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

The system shall also provide these 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.

Defining the Boundaries of the Problem

We begin our analysis by considering the hardware on which our software must execute. This is inherently a problem of systems analysis, involving manufacturability and cost issues that are far beyond the scope of this text. To bound our problem and thus allow us to expose the issues of its software analysis and design, we will make the following strategic assumptions.

- The processor (i.e., CPU) may take the form of a PC or a handheld device.
- Time and date are supplied by a clock.
- Temperature, barometric pressure, and humidity are measured via remote sensors.
- 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 a keypad.
- The display is an off-the-shelf LCD graphic device.
- A timer interrupts the computer every 1/60 second.

Figure 11–1 provides a deployment diagram that illustrates this hardware platform.

We have chosen to throw some hardware at this problem so that we might better focus on the system's software. Obviously, we could require more software by doing less in hardware (e.g., by eliminating some of the hardware for the user input and graphics device), but in this particular application, changing the hardware/software boundary is largely immaterial to our object-oriented architecture. Indeed, one of the characteristics of an object-oriented system is that it tends to speak in the vocabulary of its problem space and so represents a virtual machine that parallels our abstraction of the problem's key entities. Changing the details of the system's hardware impacts only our abstraction of the lower layers of the system.

The details of hardware interfaces can be easily insulated from our software abstractions by wrapping a class around each such interface. For example, we might devise a simple class for accessing the current time and date. We begin by doing a little isolated class analysis, in which we consider what roles and responsibilities this abstraction should encompass.[1] Thus, we might decide that

1. Actually, instead of first setting out to design a new class from scratch, we should start by looking for an existing class that already satisfies our needs. A time and data class is certainly a good candidate for reuse. However, for the purposes of this chapter, we will assume that no such class could be found.

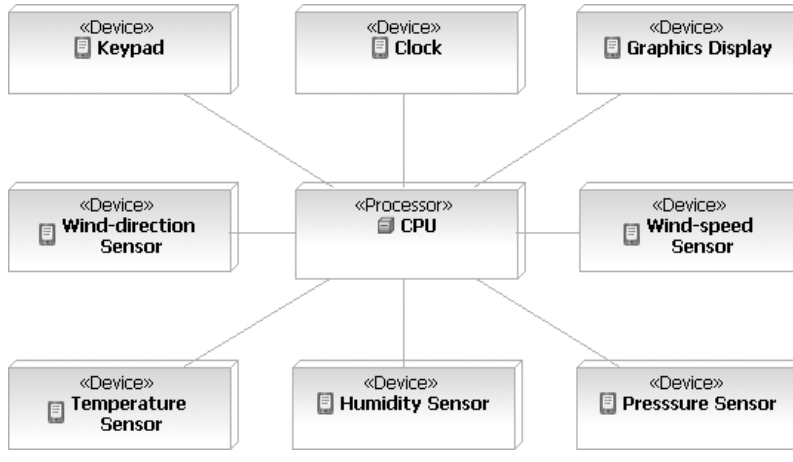

Figure 11–1 The Deployment Diagram for the Weather Monitoring System

this class is responsible for keeping track of the current time in hours, minutes, and seconds, as well as the current month, day, and year. Our analysis might decide to turn these responsibilities into two services, denoted by the operations `currentTime` and `currentDate`, respectively. The operation `currentTime` returns a string in the following format:

```
13:56:42
```

showing the current hour, minute, and second. The operation `currentDate` returns a string in the following format:

```
6-10-93
```

showing the current month, day, and year.

Further analysis suggests that a more complete abstraction would allow a client to chose either a 12- or 24-hour format for the time, which we may provide in the form of an additional modifier named `setFormat`.

By specifying the behavior of this abstraction from the perspective of its public clients, we have devised a clear separation between its interface and implementation. The basic idea here is to build the outside view of each class as if we had complete control over its underlying platform, then implement the class as a bridge to its real inside view. Thus, the implementation of a class at the system's hardware/software boundary serves to bolt the outside view of the abstraction to its underlying platform, which is often constrained by system decisions that are out of the hands of the software engineer. Of course, the gap between an abstrac-

tion's outside and inside views must not be so wide as to require a thick and inefficient implementation to glue the two views together.

One responsibility of our time and date class must therefore include setting the date and time. Carrying out this responsibility requires a new set of services to set the time and date, which we provide via the operations `setHour`, `setMinute`, `setSecond`, `setDay`, `setMonth`, and `setYear`.

We may summarize our abstraction of a time/date class as follows.

Class name:
`TimeDate`
Responsibility:
Keep track of the current time and date.
Operations:
`currentTime`
`currentDate`
`setFormat`
`setHour`
`setMinute`
`setSecond`
`setMonth`
`setDay`
`setYear`
Attributes:
`time`
`date`

Instances of this class have a dynamic lifecycle, which we can express in the state transition diagram shown in Figure 11–2. Here we see that upon initialization, an instance of this class resets its `time` and `date` attributes and then unconditionally enters the `Running` state, where it begins in 24-hour mode. Once in the `Running` state, receipt of the operation `setFormat` toggles the object between 12- and 24-hour mode. No matter what its nested state, however, setting the time or date causes the object to renormalize its attributes. Similarly, requesting its time or date causes the object to calculate a new string value.

We have specified the behavior of this abstraction in enough detail that we can offer it for use in scenarios with other clients we might discover during analysis. Before we consider these scenarios, let's specify the behavior of the other tangible objects in our system.

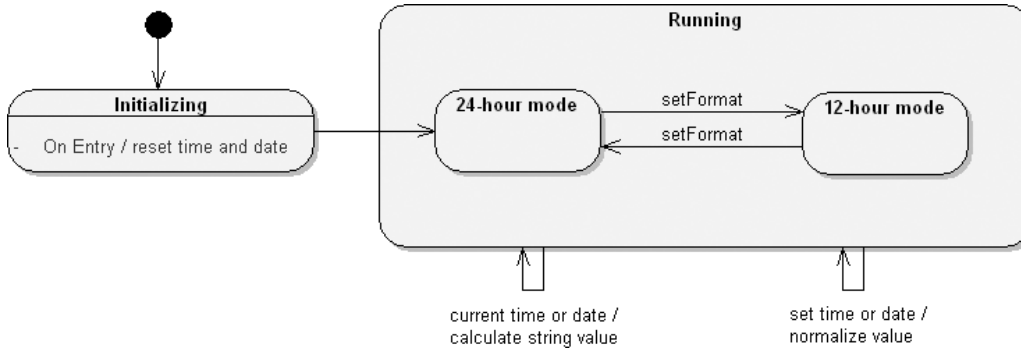

Figure 11–2 The `TimeDate` Lifecycle

The class `Temperature Sensor` serves as an analog to the hardware temperature sensors in our system. Isolated class analysis yields the following first cut at this abstraction's outside view.

Class name:
`Temperature Sensor`
Responsibility:
Keep track of the current temperature.
Operations:
`currentTemperature`
`setLowTemperature`
`setHighTemperature`
Attribute:
`temperature`

The operation `currentTemperature` is self-explanatory. The other two operations derive directly from our requirements, which obligate us to provide a mechanism for calibrating each sensor. For the moment, we will assume that each temperature sensor value is represented by a fixed-point number, whose low and high points can be calibrated to fit known actual values. We translate intermediate numbers to their actual temperatures by simple linear interpolation between these two points, as illustrated in Figure 11–3.

The careful reader may wonder why we have proposed a class for this abstraction, when our requirements imply that there is exactly one temperature sensor in the system. That is indeed true, but in anticipation of reusing this abstraction, we choose to capture it as a class, thereby decoupling it from the particulars of this one system. In fact, the number of temperature sensors monitored by a particular

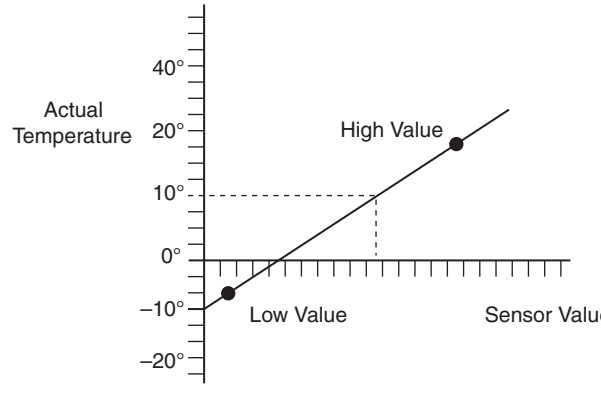

Figure 11–3 `Temperature Sensor` Calibration

system is largely immaterial to our architecture, and by devising a class, we make it simple for other programs in this family of systems to manipulate any number of sensors.

We can express our abstraction of the barometric pressure sensor in the following specification.

Class name:
`Pressure Sensor`
Responsibility:
Keep track of the current barometric pressure.
Operations:
`currentPressure`
`setLowPressure`
`setHighPressure`
Attribute:
`pressure`

A review of the system's requirements reveals that we may have missed one important behavior for this and the previous class, `Temperature Sensor`. Specifically, our requirements compel us to provide a means for reporting the temperature and pressure trends. For the moment (because we are doing analysis, not design), we will be content to focus on the nature of this behavior and, most important, on deciding which abstraction we should make responsible for this behavior.

For both the `Temperature Sensor` and the `Pressure Sensor`, we can express the trends as floating-point numbers between –1 and 1, representing the

slope of a line fitting a number of values over some interval of time.[2] Thus, we may add the following responsibility and its corresponding operation to both of these classes.

Responsibility:

Report the temperature or pressure trend as the slope of a line fitting the past values over the given interval.

Operation:

`trend`

Because this behavior is common to both the `Temperature Sensor` and `Pressure Sensor` classes, our analysis suggests the invention of a common superclass, which we will call `Trend Sensor`, responsible for providing this common behavior.

For completeness, we should point out that there is an alternative view of the world that we might have chosen in our analysis. Our decision was to make this common behavior a responsibility of the sensor class itself. We could have decided to make this behavior a part of some external agent that periodically queried the particular sensor and calculated its trend, but we rejected this approach because it was unnecessarily complex. Our original specification of the `Temperature Sensor` and `Pressure Sensor` classes suggested that each abstraction had sufficient knowledge to carry out this trend-reporting behavior, and by combining responsibilities (albeit in the form of a superclass), we end up with a simple and conceptually cohesive abstraction.

Our abstraction of the humidity sensor can be expressed in the following specification.

Class name:

`Humidity Sensor`

Responsibility:

Keep track of the current humidity, expressed as a percentage of saturation from 0% to 100%.

Operations:

`currentHumidity`
`setLowHumidity`
`setHighHumidity`

2. A value of 0 means that the temperature or pressure is stable. A value of 0.1 denotes a modest rise; a value of –0.3 denotes rapidly declining values. A value approaching –1 or 1 suggests an environmental cataclysm, which is beyond the scope of the scenarios our system is expected to handle properly.

Attribute:
`humidity`

The `Humidity Sensor` has no responsibility for calculating its trend and is therefore not a subclass of `Trend Sensor`.

A review of the system's requirements suggests some behavior common to the classes `Temperature Sensor`, `Pressure Sensor`, and `Humidity Sensor`. In particular, our requirements compel us to provide a means of reporting the highest and lowest values of each of these sensors during a 24-hour period. We defer deciding how to carry out this responsibility because that is an issue of design, not analysis. However, because this behavior is common to all three sensor classes, our analysis suggests the invention of a common superclass, which we call `Historical Sensor`, responsible for providing this common behavior.

Class name:
`Historical Sensor`
Responsibility:
Report the highest and lowest values over a 24-hour period.
Operations:
`highValue`
`lowValue`
`timeOfHighValue`
`timeOfLowValue`

`Humidity Sensor` is a direct subclass of `Historical Sensor`, as is `Trend Sensor`, which serves as an intermediate abstract class, bridging our abstractions of `Historical Sensor` and the concrete classes `Temperature Sensor` and `Pressure Sensor`.

Our abstraction of the wind-speed sensor can be expressed in the following specification.

Class name:
`WindSpeed Sensor`
Responsibility:
Keep track of the current wind speed.
Operations:
`currentSpeed`
`setLowSpeed`
`setHighSpeed`
Attribute:
`speed`

Our requirements suggest that we cannot detect the current wind speed directly; rather, we must calculate its value by taking the number of revolutions of the cups on the boom, dividing by the interval over which those revolutions were counted, and then applying a scaling value appropriate to the particular boom assembly. Needless to say, this calculation is one of the secrets of this class; clients could care less how `currentSpeed` is calculated, as long as this operation satisfies its contract and delivers meaningful values.

A quick domain analysis of the last four concrete classes (`Temperature Sensor`, `Pressure Sensor`, `Humidity Sensor`, and `WindSpeed Sensor`) reveals yet another behavior in common: Each of these classes knows how to calibrate itself by providing a linear interpolation against two known data points. Rather than replicating this behavior in all four classes, we instead choose to make this behavior the responsibility of an even higher superclass, which we call `Calibrating Sensor`, whose specification includes the following.

Class name:
`Calibrating Sensor`
Responsibility:
Provide a linear interpolation of values, given two known data points.
Operations:
`currentValue`
`setHighValue`
`setLowValue`

`Calibrating Sensor` is an immediate superclass of `Historical Sensor`.[3]

Our final concrete sensor for wind direction is a bit different because it requires neither calibration nor history. We may express our abstraction of this entity in the following specification.

Class name:
`WindDirection Sensor`
Responsibility:
Keep track of the current wind direction, in terms of points along a compass rose.
Operation:
`currentDirection`

3. This hierarchy passes our litmus test for inheritance: a `Temperature Sensor` is a kind of `Trend Sensor`, which is also a kind of `Historical Sensor`, which in turn is a kind of `Calibrating Sensor`.

Attribute:
`direction`

To unify our sensor abstractions, we generate the abstract base class `Sensor`, which serves as the immediate superclass to both the classes `WindDirection Sensor` and `Calibrating Sensor`. Figure 11–4 illustrates this complete hierarchy.

Although not part of the sensor hierarchy, our abstraction of the keypad for user input has a simple specification.

Class name:
`Keypad`
Responsibility:
Keep track of the last user input.
Operation:
`lastKeyPress`
Attribute:
`key`

Notice that this class has no knowledge of the meaning of any particular key: Instances of this class know only that one of several keys was pressed. We delegate responsibility for interpreting the meaning of these keys to a different class, which we will identify when we apply these concrete boundary classes to our scenarios.

Our abstraction of an `LCD Device` class serves to insulate our software from the particular hardware we might use. To decouple our software from the particular graphics hardware we might use, our analysis leads us to prototype some common displays for the Weather Monitoring System, and then determine our interface needs.

Figure 11–5 provides such a prototype. Here, we have omitted the required display of wind chill and dew point, as well as such details as how to display the 24-hour high or low value of primary measurements. Nonetheless, some patterns emerge: We need to display only text (in two different sizes and two different styles), circles, and lines (of varying thickness). Additionally, we note that some elements of our display are static (such as the label TEMP), while others are dynamic (such as the wind direction). We choose to display both static and dynamic elements via software. In this manner, we lessen the burden on our hardware by eliminating the need for special labels on the LCD itself, but we require slightly more of our software.

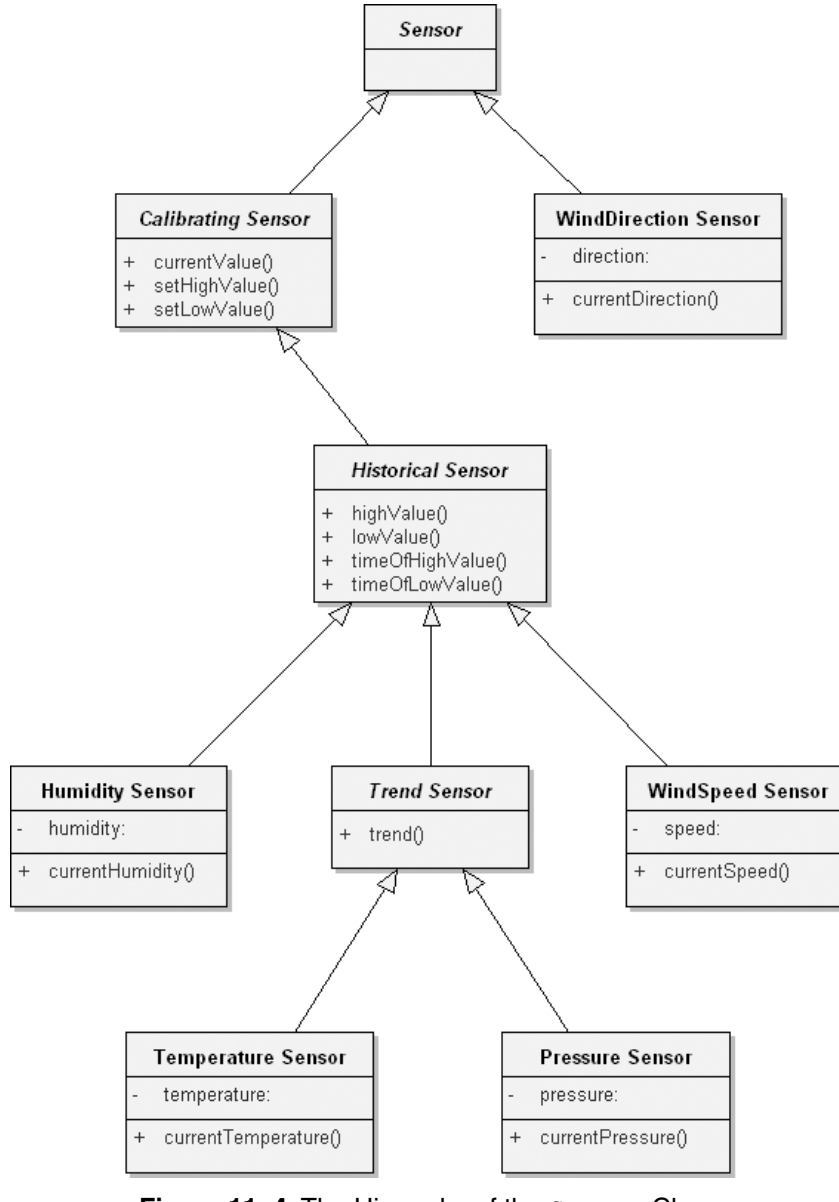

Figure 11–4 The Hierarchy of the `Sensor` Class

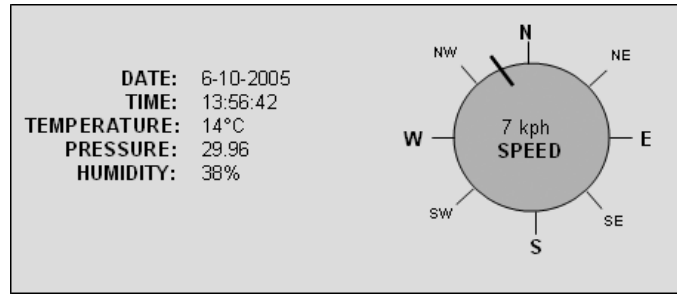

Figure 11–5 The Display for the Weather Monitoring System

We can translate these requirements into the following class specification.

Class name:

`LCD Device`

Responsibility:

Manage the LCD device and provide services for displaying certain graphics elements.

Operations:

```
drawText
drawLine
drawCircle
setTextSize
setTextStyle
setPenSize
```

As with the class `Keypad`, the class `LCD Device` has no knowledge of the meaning of the elements it manipulates. Instances of this class know only how to display text and lines; they do not know what these figures represent. This separation of concerns leaves us with loosely coupled abstractions (which is what we desire), but it does require that we find some agent responsible for mediating between the raw sensors and the display. We defer the invention of this new abstraction until we study some scenarios applicable to this system.

The final boundary class we need to consider is that of the timer. We will make the simplifying assumption that there is exactly one timer per system, whose behavior is to interrupt the computer every 1/60 of a second and, in so doing, to invoke an interrupt service routine. This is a particularly grungy detail, and it would be best if we could hide this implementation detail from the rest of our software abstractions. We can do so by devising a class that uses a callback function and exports only static members (so that we constrain our system to have exactly one timer).

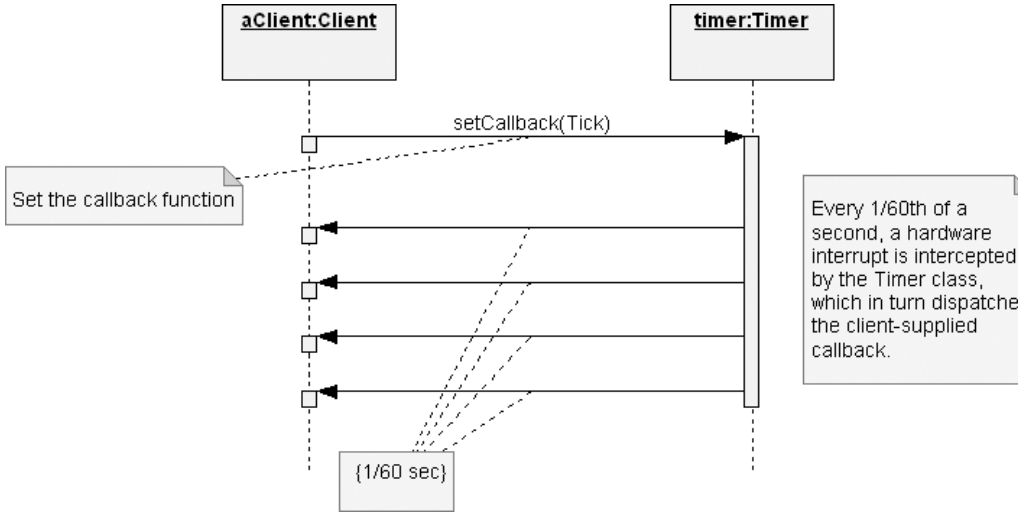

Figure 11–6 The `Timer` Interaction Diagram

Figure 11–6 provides a sequence diagram that illustrates a use case for this abstraction. Here we see how the timer and its client collaborate: The client begins by supplying a callback function, and every 1/60 of a second, the timer calls that function. In this manner, we decouple the client from knowing about how to intercept timed events, and we decouple the timer from knowing what to do when such an event occurs. The primary responsibility that this protocol places on the client is simply that the execution of its callback function must always take less than 1/60 of a second; otherwise, the timer will miss an event.

By intercepting time events, the `Timer` class serves as an active abstraction, meaning that it is at the root of a thread of control. We may express our abstraction of this class in the following specification.

Class name:
`Timer`
Responsibility:
Intercept all timed events and dispatch a callback function accordingly.
Operation:
`setCallback()`

Scenarios

Now that we have established the abstractions at the boundaries of our system, we continue our analysis by studying several scenarios of its use. We begin by enu-

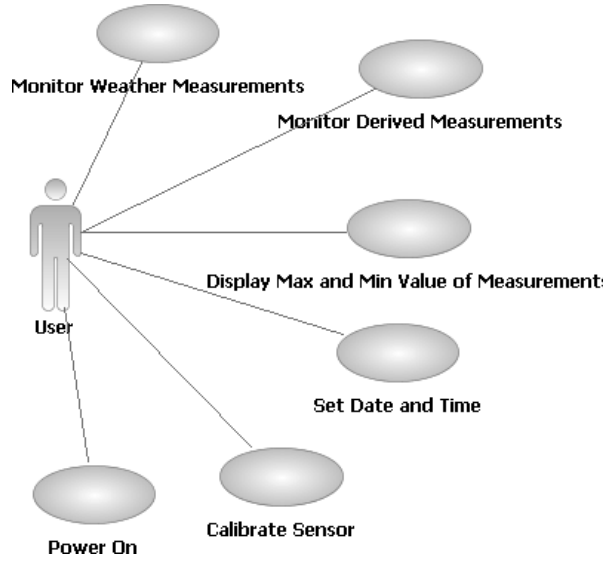

Figure 11–7 Primary Use Cases for the Weather Monitoring System

merating a number of primary use cases (Figure 11–7), as viewed from the point of view of the clients of this system:

- Monitoring basic weather measurements, including wind speed and direction, temperature, barometric pressure, and humidity
- Monitoring derived measurements, including wind chill, dew point, temperature trend, and barometric pressure trend
- Displaying the highest and lowest values of a selected measurement
- Setting the time and date
- Calibrating a selected sensor
- Powering up the system

We add to this list two secondary use cases:

- Power failure
- Sensor failure

11.2 Elaboration

Let's examine a number of these scenarios in order to illuminate the behavior—but not the design—of the system.

Weather Monitoring System Use Cases

Monitoring basic weather measurements is the principal function point of the Weather Monitoring System. One of our system constraints is that we cannot take measurements any faster than 60 times a second. Fortunately, most interesting weather conditions change much more slowly. Our analysis suggests that the following sampling rates are sufficient to capture changing conditions:

- Wind direction: every 0.1 second
- Wind speed: every 0.5 seconds
- Temperature, barometric pressure, and humidity: every 5 minutes

Earlier, we decided that the classes representing each primary sensor should have no responsibility for dealing with timed events. Our analysis therefore requires that we devise an external agent that collaborates with these sensors to carry out this scenario. For the moment, we will defer our specification of the behavior of this agent (how it knows when to initiate a sample is an issue of design, not analysis). The interaction diagram shown in Figure 11–8 illustrates this scenario. Here we see that when the agent begins sampling, it polls each sensor in turn but intentionally skips certain sensors in order to sample them at a slower rate. By polling each sensor rather than letting each sensor act as a thread of control, the execution of our system is more predictable because our agent can control the flow of events. Because this name reflects its place in the behavior of the system, we will make this agent an instance of the class `Sampler`.

We must continue this scenario by asking which of these objects in the interaction diagram is then responsible for displaying the sampled values on the one instance of our `LCD Device` class. Ultimately, we have two choices: We can have each sensor be responsible for displaying itself (the common pattern used in MVC-like architectures), or we can have a separate object be responsible for this behavior. For this particular problem, we choose the latter option because it allows us to

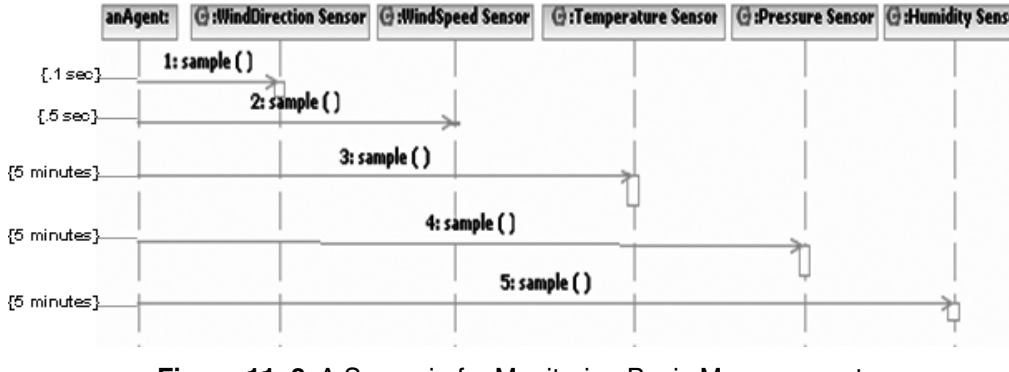

Figure 11–8 A Scenario for Monitoring Basic Measurements

encapsulate all our design decisions about the layout of our display in one class.[4] Thus, we add the following class specification to our products of analysis.

Class name:
`Display Manager`
Responsibility:
Manage the layout of items on the LCD device.
Operations:
`drawStaticItems`
`displayTime`
`displayDate`
`displayTemperature`
`displayHumidity`
`displayPressure`
`displayWindChill`
`displayDewPoint`
`displayWindSpeed`
`displayWindDirection`
`displayHighLow`

The operation `drawStaticItems` exists to draw the unchangeable parts of the display, such as the compass rose used for indicating the wind direction. We will also assume that the operations `displayTemperature` and `displayPressure` are responsible for displaying their corresponding trends (therefore, as we move into implementation, we must provide a suitable signature for these operations).

Figure 11–9 provides a class diagram illustrating the abstractions that must collaborate to carry out this scenario. Note that we also indicate the role that each abstraction plays in its association with other classes.

There is one important side effect from our decision to include the class `Display Manager`.[5] Specifically, internationalizing our software, that is, adapting it to

4. The dominant problem here is where we display each item, not how each item looks. Because this is a decision that is likely to change, it is best for us to encapsulate in one class all the knowledge about where to display each item on the LCD device. Changing our assumptions about front panel layout therefore requires that we touch only one class instead of many.

5. Is this an analysis decision or a design decision? The question can be argued in either direction, although such arguments are largely academic in the face of having to deliver production software. If a decision advances our understanding of the system's desired behavior and in addition leads us to an elegant architecture, we don't really care what it is called.

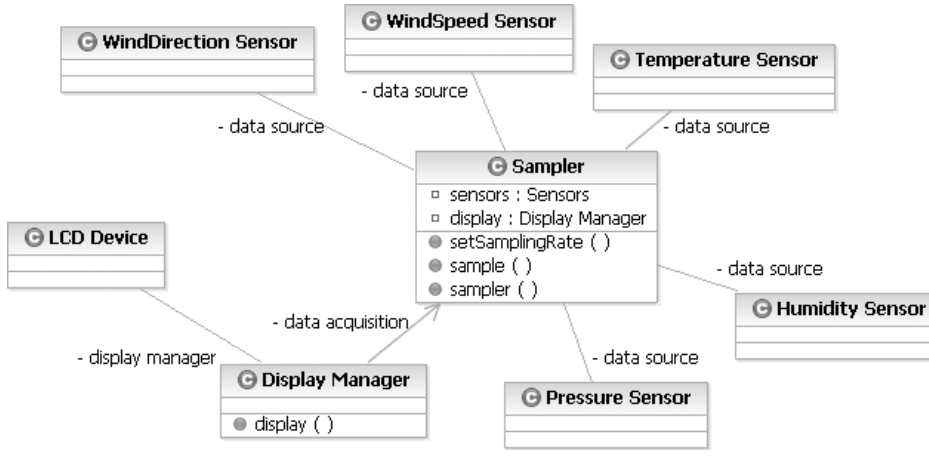

Figure 11–9 The `Sampler` and `Display Manager` Classes

different countries and languages, becomes much easier given this design decision because the knowledge about how elements are named and thus labeled on the display (such as `TEMP` and `WIND`) is part of the secrets of this one class.

Internationalization leads us to consider an issue about which the requirements are silent: Should the system display temperature in Centigrade or Fahrenheit? Similarly, should the system display wind speed in kilometers per hour (kph) or miles per hour (mph)? Ultimately, our software should not constrain us. Because we seek end-user flexibility, we must add an operation `setMode` to both the `Temperature Sensor` and `WindSpeed Sensor` classes. We must also add a new responsibility to each of these classes, which makes their instances construct themselves in a known stable state. Finally, we must modify the signature of the operation `Display Manager::drawStaticItems` accordingly, so that when we change units of measurement, the display manager can update the front panel display if needed.

This discovery leads us to add one more scenario for consideration in our analysis, namely:

- Setting the unit of measurement for temperature and wind speed

We will defer considering this scenario until we study the other use cases that deal with user interaction.

Monitoring the derived measurements for temperature and pressure trends can be achieved through the protocol we have already established for the `Temperature Sensor` and `Pressure Sensor` classes. However, to complete this scenario for all derived measurements, we are now led to discover two new classes, which

we call `Wind Chill` and `Dew Point`, responsible for calculating their respective values. Neither of these abstractions represents sensors because they do not denote any tangible device in the system. Rather, each one acts as an agent that collaborates with two other classes to carry out its responsibilities. Specifically, the `Wind Chill` conspires with the `Temperature Sensor` and `WindSpeed Sensor`, and the `Dew Point` conspires with the `Temperature Sensor` and `Humidity Sensor`. In turn, `Wind Chill` and `Dew Point` collaborate with `Sampler`, using the same mechanism as `Sampler` uses to monitor all the primary weather measurements. Figure 11–10 illustrates the classes involved in this scenario; basically, this class diagram is just a slightly different view of the system than the one shown in Figure 11–9.

Why do we define `Wind Chill` and `Dew Point` as classes, instead of just carrying out their calculation through a simple nonmember function? The answer is that this situation passes our litmus test for object-oriented abstractions: Instances of both `Wind Chill` and `Dew Point` provide some behavior (namely, the calculation of their respective values) and encapsulate some state (each must maintain an association with a particular instance of two different concrete sensors), and each has a unique identity (each particular wind-speed sensor/temperature sensor association must have its own `Wind Chill` object). By "objectifying" these seemingly algorithmic abstractions, we also end up with a more reusable architecture: Both `Wind Chill` and `Dew Point` can be lifted from this particular application because each presents a clear contract to its clients, and each offers a clear separation of concerns relative to all the other abstractions.

Moving on, we next consider the various scenarios that relate to user interaction with the Weather Monitoring System. Deciding on the proper user gestures for

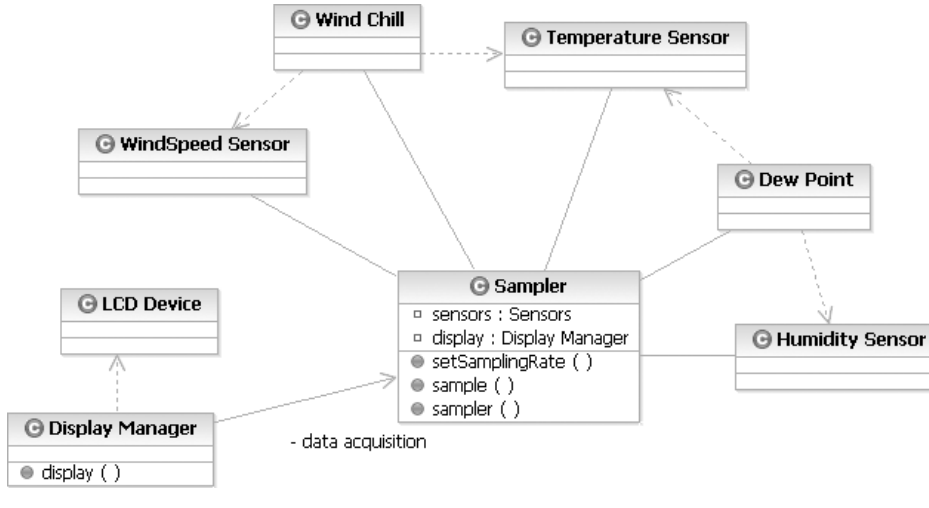

Figure 11–10 Classes for Derived Measurements

interacting with an embedded controller such as this one is still as much of an art as is designing a graphical user interface. A full treatment of how to devise such user interfaces is beyond the scope of this text, but the basic message for the software analyst is that prototyping works and indeed is fundamental in helping to mitigate the risks involved in user interface design. Furthermore, by implementing our decisions in terms of an object-oriented architecture, we make it relatively easy to change these user interface decisions without rending the fabric of our design.

Consider some possible use case scenarios of user interaction.

Use case name:

`Display Max and Min Value of Measurements`

Description:

This use case displays the maximum and minimum values of a selected measurement.

Basic flow:

1. The use case begins when the user presses the SELECT key.
2. The system displays SELECTING.
3. The user presses any one of the keys WIND, TEMP, PRESSURE, or HUMIDITY; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN key to select display of the highest or lowest 24-hour value, respectively; any other key press (except RUN) is ignored.
6. The system displays the selected value, together with its time of occurrence.
7. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display, the selected value, and the SELECTING message are removed.

This scenario leads us to enhance the `Display Manager` class by adding both the operations `flashLabel` (which causes the identified label to flash or stop flashing, according to an appropriate operation argument) and `displayMode` (which displays a text message on the LCD device).

Setting the time and date follows a similar scenario.

Use case name:

`Set Date and Time`

Description:

This use case sets the date and time.

Basic flow:

1. The use case begins when the user presses the SELECT key.
2. The system displays SELECTING.
3. The user presses either of the keys TIME or DATE; any other key press (except RUN and the keys listed in step 3 of the previous scenario) is ignored.
4. The system flashes the corresponding label; the display also flashes the first field of the selected item (namely, the hours field for the time and the month field for the date).
5. The user presses the LEFT or RIGHT keys to select another field (selection wraps around); the 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 step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the SELECTING message are removed, and the time or date are reset.

Calibrating a particular sensor follows a related pattern of user gestures.

Use case name:

`Calibrate Sensor`

Description:

This use case is used to calibrate the sensors.

Basic flow:

1. The use case begins when the user presses the CALIBRATE key.
2. The system displays CALIBRATING.
3. The user presses any one of the keys WIND, TEMP, PRESSURE, or HUMIDITY; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN keys to select the high or low calibration point.
6. The display flashes the corresponding value.
7. The user presses the UP or DOWN keys to adjust the selected value.
8. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the CALIBRATING message are removed, and the calibration function is reset.

While calibrating, instances of the `Sampler` class must be told to not sample the selected item; otherwise, erroneous information would be displayed to the user. This scenario therefore requires that we introduce two new operations for the

`Sampler` class, namely, `inhibitSample` and `resumeSample`, both of which have a signature that specifies a particular measurement.

Our last primary scenario involving the user interface concerns setting units of measurement.

Use case name:

`Set Unit of Measurement`

Description:

This use case sets the unit of measurement for temperature and wind speed.

Basic flow:

1. The use case begins when the user presses the MODE key.
2. The system displays MODE.
3. The user presses either of the keys WIND or TEMP; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN keys to toggle the current unit of measurement.
6. The system updates the unit of measurement for the selected item.
7. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the MODE message are removed, and the current unit of measurement for the item is set.

A study of these scenarios leads us to decide on an arrangement for buttons on the keypad (a system decision), which we illustrate in Figure 11–11.

Each of these user interface scenarios involves some form of modality or event-ordered behavior and so is well suited to expression through the use of state transition diagrams. Because these scenarios are so tightly coupled, we choose to devise a new class, `InputManager`, which is responsible for carrying out the following contractual specification.

Figure 11–11 The User Keypad for the Weather Monitoring System

Class name:
`InputManager`
Responsibility:
Manage and dispatch user input.
Operation:
`processKeyPress`

The sole operation, `processKeyPress`, animates the state machine that lives behind instances of this class.

As shown in Figure 11–12, the outermost state machine diagram for this class encompasses four states: `Running`, `Calibrating`, `Selecting`, and `Mode`. These states correspond directly to the earlier scenarios. We transition to the respective states based on the first key press intercepted while `Running`, and we return to the `Running` state when the last key press is again `Run`. Each time we enter `Running`, we clear the message on the display.

We have expanded the `Mode` state to show how we might more formally express the dynamic semantics of our scenario. As we first enter this state, our entry action is to display an appropriate message on the display. We begin in the `Waiting` state and transition out of this state if we intercept a user key press of the `TEMP` or `WIND` keys, which causes us to enter a nested state of `Processing`,

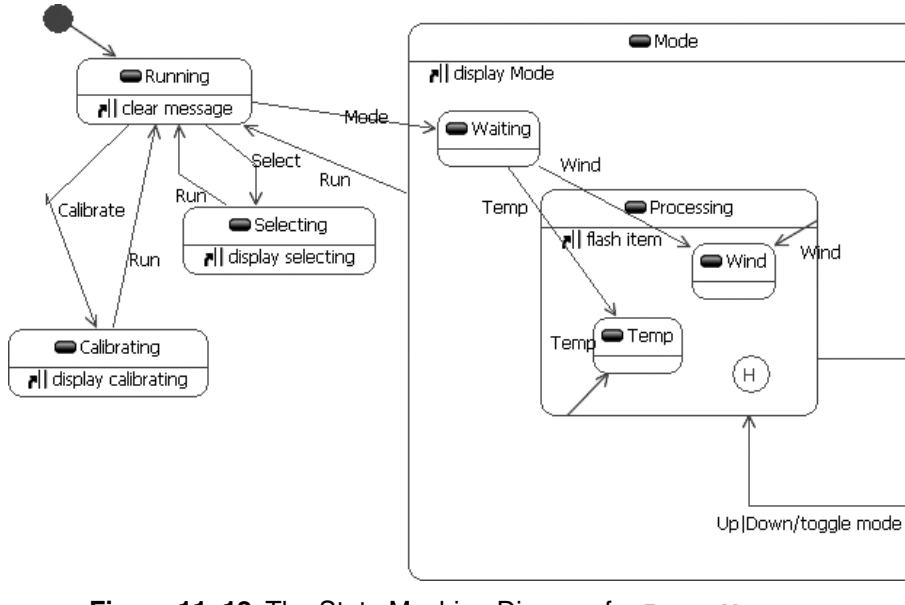

Figure 11–12 The State Machine Diagram for `InputManager`

or a user key press of RUN, which transitions us back to the outermost Running state. Each time we enter Processing, we flash the appropriate item; in subsequent entries to this state, we enter the previously entered nested state, Temp or Wind.

While in the Temp or Wind state, we may intercept one of five key presses: UP or DOWN (which toggles the corresponding mode), TEMP or WIND (which reenters the appropriate nested state), or RUN (which ejects us from the outer Mode state).

The Selecting and Calibrating states similarly expand out to reveal more nested states. We will not show their expanded state machine diagrams here because their presentation does not reveal anything particularly interesting about the problem at hand.[6]

Our final primary scenario involves powering up the system, which requires that we bring all of its objects to life in an orderly fashion, ensuring that each one starts in a stable initial state. We may write a script for our analysis of this scenario as follows.

Use case name:

Power On

Description:

Power up the system.

Basic flow:

1. This use case begins when power is applied.
2. Each sensor is constructed; historical sensors clear their history, and trend sensors prime their slope-calculating algorithms.
3. The user input buffer is initialized, causing garbage key presses (due to noise on power up) to be discarded.
4. The static elements of the display are drawn.
5. The sampling process is initiated.

Postconditions:

The past high/low values of each primary measurement is set to the value and time of their first sample.

The temperature and pressure trends are flat.

The InputManager is in the Running state.

6. Of course, for a production product, a comprehensive analysis would complete the exposition of this state transition diagram. We can defer this task here because it is more tedious than not and in fact does not reveal anything we do not already know about the system under construction.

Notice the use of postconditions in our script to specify the expected state of the system after this scenario completes. As we shall see, there is no one agent in the system that carries out this scenario; rather, this behavior results from the collaboration of a number of objects, each of which is given the responsibility to bring itself to a stable initial state.

This completes our study of the Weather Monitoring System's primary scenarios. To be utterly complete, we might want to walk through the various secondary scenarios. At this point, however, we have exposed a sufficient number of the system's function points, and we want to proceed with architectural design, so that we might begin to validate our strategic decisions.

Every software system needs to have a simple yet powerful organizational philosophy (think of it as the software equivalent of a sound bite that describes the system's architecture), and the Weather Monitoring System is no exception. The next step in our development process is to articulate this architectural framework, so that we might have a stable foundation on which to evolve the system's function points.

The Architecture Framework

In data acquisition and process control domains, we might follow many possible architectural patterns, but the two most common alternatives involve either the synchronization of autonomous actors or time-frame-based processing.

In the first pattern, our architecture encompasses a number of relatively independent objects, each of which serves as a thread of control. For example, we might invent several new sensor objects that build on more primitive hardware/software abstractions, with each such object responsible for taking its own sample and reporting back to some central agent that processes these samples. This architecture has its merits; it may be the only meaningful framework if we have a distributed system in which we must collect samples from many remote locations. This architecture also allows for more local optimization of the sampling process (each sampling actor has the knowledge to adjust itself to changing conditions, perhaps by increasing or decreasing its sampling rate as conditions warrant).

However, this architectural pattern is generally not well suited to hard real-time systems, wherein we must have complete predictability over when events take place. Although the Weather Monitoring System is not hard real-time, it does require some modicum of predictable, ordered behavior. For this reason, we turn to an alternative pattern, that of time-frame-based processing.

As we illustrate in Figure 11–13, this model takes time and divides it into several (usually fixed-length) frames, which we further divide into subframes, each of

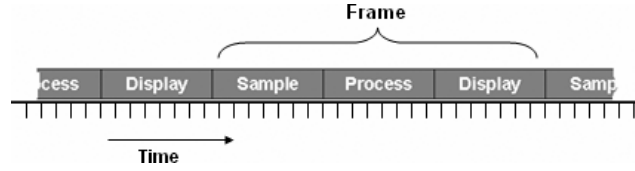

Figure 11–13 Time-Frame Processing

which encompasses some functional behavior. The activity from one frame to another may be different. For example, we might sample the wind direction every 10 frames but sample the wind speed only every 30 frames.[7] The primary merit of this architectural pattern is that we can more rigorously control the order of events.

Figure 11–14 provides a class diagram that expresses this architecture for the Weather Monitoring System. Here we find most of the classes we discovered earlier during analysis, the main difference being that we now show how all the key abstractions collaborate with one another. As is typical in class diagrams for production systems, we do not (and cannot) show every class and every relationship. For example, we have omitted the class hierarchy regarding all of the sensors.

We have invented one new class in this architecture, namely, the class `Sensors`, whose responsibility is to serve as the collection of all the physical sensors in the system. Because at least two other agents in the system (`Sampler` and `InputManager`) must associate with the entire collection of sensors, bundling them in one container class allows us to treat our system's sensors as a logical whole.

11.3 Construction

The central behavior of this architecture is carried out by a collaboration of the `Sampler` and `Timer` classes. We would be wise during architectural design to concretely prototype these classes so that we can validate our assumptions.

The Frame Mechanism

We begin by refining the interface of the class `Timer`, which dispatches a callback function. Figure 11–15 shows the class design.

7. For example, if each frame is allocated to be 1/60 second, 30 frames represents 0.5 second.

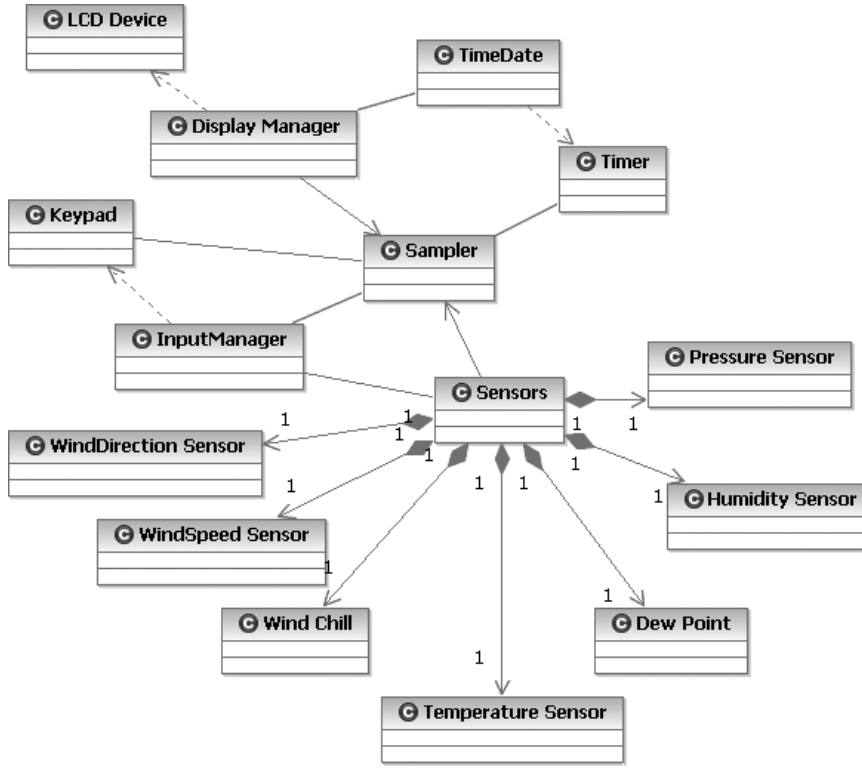

Figure 11–14 The Architecture of the Weather Monitoring System

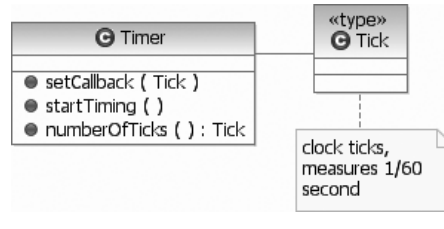

Figure 11–15 The Design of the `Timer` Class

`Timer` is an unusual class, but remember that it holds some unusual secrets. We use the first operation `setCallback` to attach a callback function to the timer. We launch the timer's behavior by invoking `startTiming`, after which time the one `Timer` entity dispatches the callback function every 1/60 of a second. Notice that we introduce an explicit starting operation because we cannot rely on any particular implementation-dependent ordering in the elaboration of declarations.

Before we turn to the `Sampler` class, we introduce a new declaration that names the various sensors in this particular system. The enumeration class

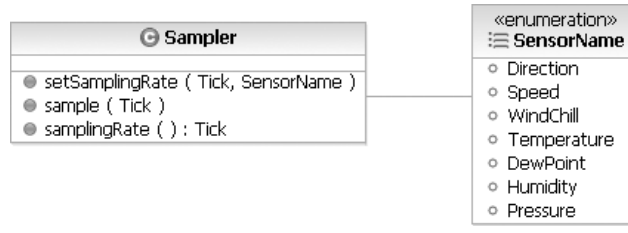

Figure 11–16 The Interface of the `Sampler` Class

`SensorName` contains enumeration literals for all the sensors in our system. Figure 11–16 shows the interface of the `Sampler` class.

We have introduced the modifier `setSamplingRate` and its selector `samplingRate` so that clients can dynamically alter the behavior of the sampling objects.

To tie the `Timer` and `Sampler` classes together, we just need a little bit of C++ glue code. First we declare an instance of `Sampler` and a nonmember function.

```
Sampler sampler;

void acquire(Tick t)
{
  sampler.sample(t);
}
```

Now we can write a fragment of our main function, which simply attaches the callback function to the timer and starts the sampling process.

```
main() {

  Timer::setCallback(acquire);
  Timer::startTiming();

  while(1) {
    ;
  }

  return 0;

}
```

This is a fairly typical main program for object-oriented systems: It is short (because the real work is delegated to key objects in the system), and it involves a

dispatch loop (which in this case does nothing because we have no background processing to complete).[8]

To continue this thread of the system's architecture, we next provide an interface for the `Sensors` class (Figure 11–17). For the moment, we assume the existence of the various concrete sensor classes.

This is basically a collection type class, and for this reason we make `Sensors` a subclass of the foundation class `Collection`.[9] We make `Collection` a protected superclass because we don't want to expose most of its operations to clients of the `Sensors` class. Our declaration of `Sensors` provides only a sparse set of operations because our problem is sufficiently constrained that we know sensors are only added and never removed from the collection.

We have invented a generalized sensor collection class that can hold multiple instances of the same kind of sensor, with each instance within its class distinguished by a unique ID, numbered starting at zero.

We specify in the `Sampler` class the associations with the `Sensors` and `Display Manager` classes and revise our declaration of the one instance of the `Sampler` class (Figure 11–18).

The construction of the `Sampler` object connects this agent with the specific collection of sensors and the particular display manager used in the system.

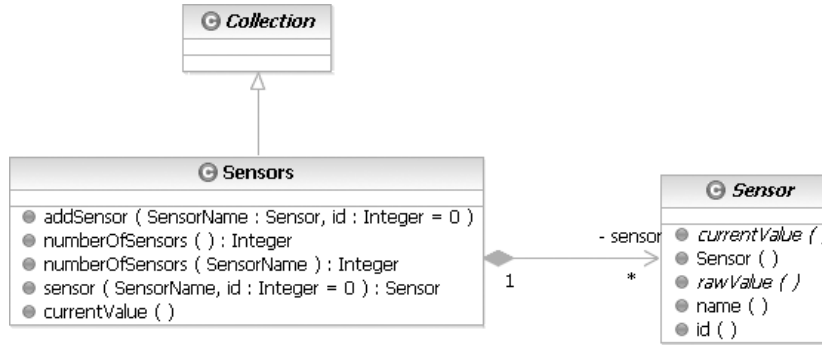

Figure 11–17 The Interface of the `Sensors` class

8. This is yet another common architectural pattern: Dispatch loops serve to intercept external or internal events and then dispatch them to the appropriate agents.

9. The `Collection` class is an abstract superclass that provides common operations for a collection of items supplied by language libraries.

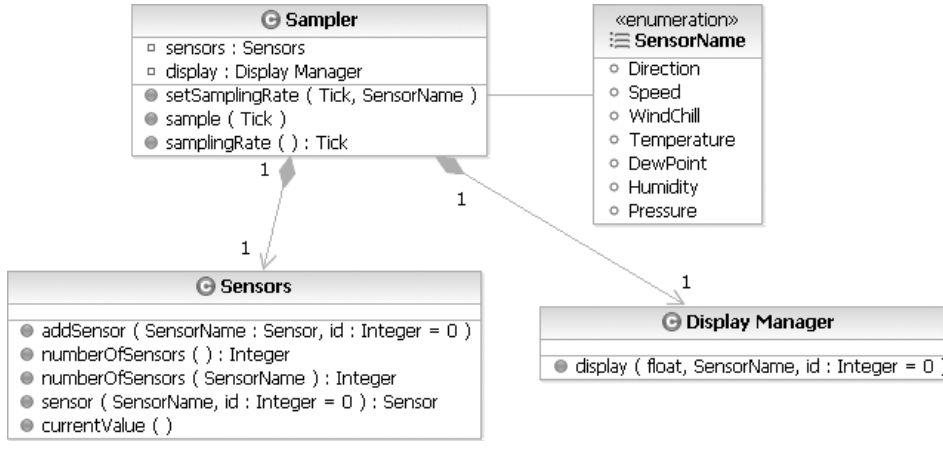

Figure 11–18 The Design of the `Sampler` Class

Now we can implement the `Sampler` class's key operation, `sample`.

```
void Sampler::sample(Tick t)
{
  for (SensorName name = Direction; name <= Pressure; name++)
    for (unsigned int id = 0; id  <
            repSensors.numberOfSensors(name); id++)

      if (!(t % samplingRate(name)))

        repDisplayManager.display(repSensors.sensor(name,
            id).currentValue(), name, id);
}
```

The action of this member function is to iterate through each kind of sensor and, in turn, each unique sensor of that kind in the collection. For each sensor it encounters, `sample` checks to see whether it is time to sample its value and, if so, references the sensor from the collection, takes its current value, and delivers this value to the display manager associated with the `Sampler` instance.[10]

The semantics of this operation relies on the polymorphic behavior of the operation `currentValue` defined for the base class `Sensor`. This operation also relies on the operation `display` defined for the class `Display Manager`.

10. An alternate approach would be to have each sensor provide a member function that returns its sampling rate and another member function that draws the sensor on the LCD. This design would make the implementation of the `Sampler` class simpler and more extensible, although it would shift more responsibilities to the sensor classes.

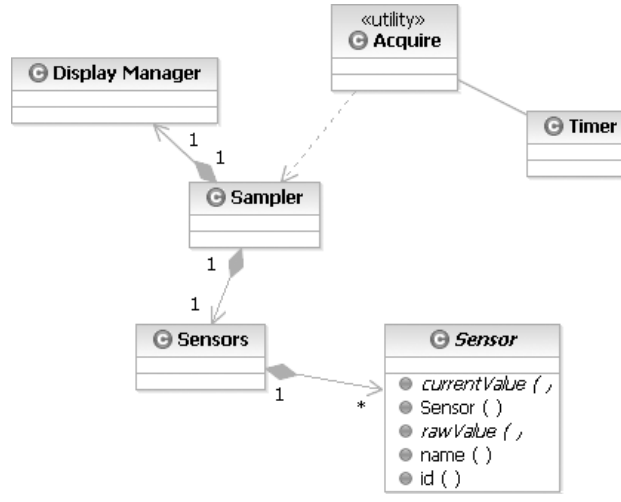

Figure 11–19 The Frame Mechanism

Now that we have refined this element of our architecture, we present a new class diagram in Figure 11–19 that highlights this frame mechanism.

Now that we have validated our architecture by walking through several scenarios, we can continue with the incremental development of the system's function points.

Release Planning

We start this process by proposing a sequence of releases, each of which builds on the previous release.

- Develop a minimal functionality release, which monitors just one sensor.
- Complete the sensor hierarchy.
- Complete the classes responsible for managing the display.
- Complete the classes responsible for managing the user interface.

We could order these releases in just about any manner, but we choose this one, which progresses from highest to lowest risk, thereby forcing our development process to directly attack the hard problems first.

Developing the minimal functionality release forces us to take a vertical slice through our architecture and implement small parts of just about every key abstraction. This activity addresses the highest risk in the project, namely,

whether we have the right abstractions with the right roles and responsibilities. This activity also gives us early feedback because we can now play with an executable system. Forcing early closure like this has a number of technical and social benefits. On the technical side, it forces us to begin to bolt the hardware and software parts of our system together, thereby identifying any impedance mismatches early. On the social side, it allows us to get early feedback about the look and feel of the system, from the perspectives of real users.

Because completing this release is largely a manner of tactical implementation, we will not bother with exposing any more of its structure. We will now turn to elements of later releases because they reveal some interesting insights about the development process.

The Sensor Mechanism

In inventing the architecture for this system, we have already seen how we had to iteratively and incrementally evolve our abstraction of the sensor classes, which we began during analysis. In this evolutionary release, we expect to build on the earlier completion of a minimal functional system and finish the details of this class hierarchy.

At this point in our development cycle, the class hierarchy we first presented in Figure 11–4 remains stable, although, not surprisingly, we had to adjust the location of certain polymorphic operations in order to extract greater commonality. Specifically, in an earlier section we noted the requirement for the `currentValue` operation, declared in the abstract base class `Sensor`. We may complete our design of the `Sensor` class (Figure 11–20).

Notice that through the class constructor, we gave the instances of this class knowledge of their name and ID. This is essentially a kind of runtime type identification, but providing this information is unavoidable here because, per the requirements, each sensor instance must have a mapping to a particular interface. We can hide the secrets of this mapping by making this interface a function of a sensor name and ID.

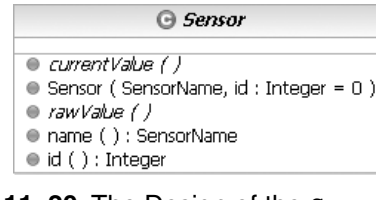

Figure 11–20 The Design of the `Sensor` Class

Now that we have added this new responsibility, we can go back and simplify the signature of `DisplayManager::display` to take only a single argument, namely, a reference to a `Sensor` object. We can eliminate the other arguments to this function because the `Display Manager` can now ask the `Sensor` object its name and ID.

Making this change is advisable because it simplifies certain cross-class interfaces. Indeed, if we fail to keep up with small, rippling changes such as this one, our architecture will eventually suffer as the protocols among collaborating classes become inconsistently applied.

The declaration of the immediate subclass `Calibrating Sensor` builds on the base class `Sensor` (Figure 11–21).

`Calibrating Sensor` introduces two new operations (`setHighValue` and `setLowValue`) and implements the previously defined function `currentValue`.

Next, consider the declaration of the subclass `Historical Sensor`, which builds on the class `Calibrating Sensor` (Figure 11–22).

`Historical Sensor` has four operations whose implementation requires collaboration with the `TimeDate` class for the time of the high or low values. Note that `Historical Sensor` is still an abstract class because we have not yet completed the definition of the abstract function `rawValue`, which we defer to be a concrete subclass responsibility.

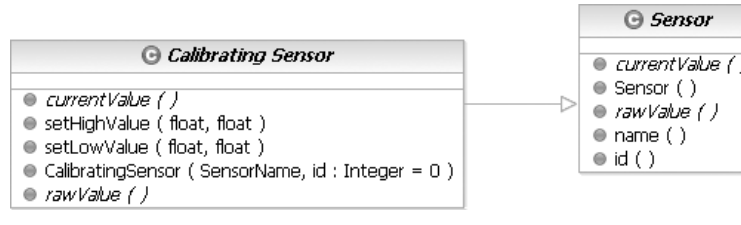

Figure 11–21 The Design of the `Calibrating Sensor` Class

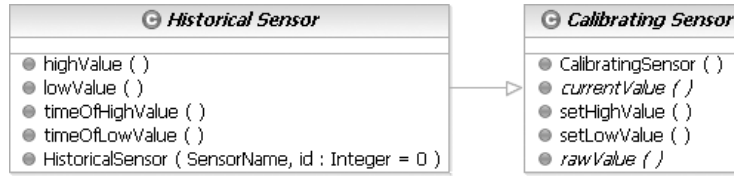

Figure 11–22 The Design of the `Historical Sensor` Class

The class `Trend Sensor` inherits from `Historical Sensor` and adds one new responsibility (Figure 11–23).

`Trend Sensor` introduces one new function. As with some of the other operations that some other intermediate classes have added, we declare `trend` as concrete because we do not desire that subclasses change their behavior.

Ultimately, we reach concrete subclasses such as `Temperature Sensor` (Figure 11–24).

Notice that the signature of this class's constructor is slightly different than its superclass's, simply because at this level of abstraction, we know the specific name of the class. Also, notice that we have introduced the operation `currentTemperature`, which follows from our earlier analysis. This operation is semantically the same as the polymorphic function `currentValue`, but we choose to include both of them because the operation `currentTemperature` is slightly more type-safe.

Once we have successfully completed the implementation of all classes in this hierarchy and integrated them with the previous release, we may proceed to the next level of the system's functionality.

The Display Mechanism

Implementing the next release, which completes the functionality of the classes `DisplayManager` and `LCD Device`, requires virtually no new design work, just some tactical decisions about the signature and semantics of certain func-

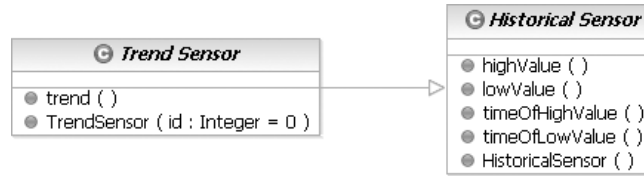

Figure 11–23 The Design of the `Trend Sensor` Class

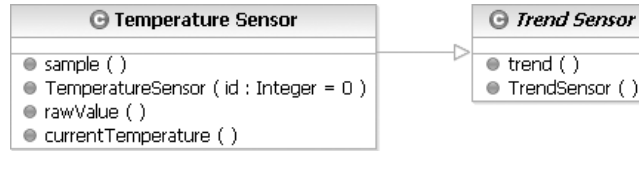

Figure 11–24 The Design of the `Temperature Sensor` Class

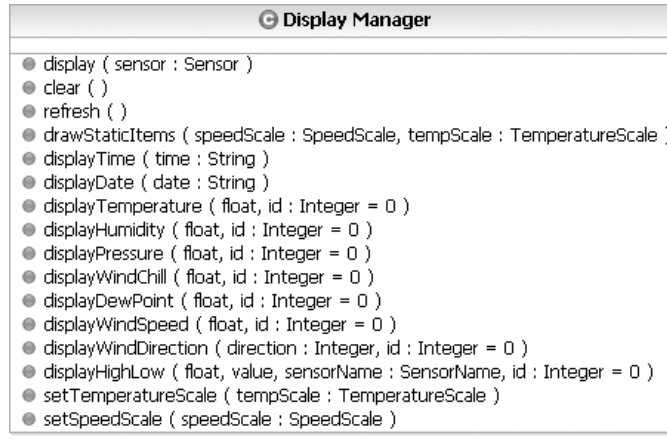

Figure 11–25 The Design of the `Display Manager` Interface

tions. Combining the decisions we made during analysis with our first architectural prototype, wherein we made some important decisions about the protocol for displaying sensor values, we can derive the concrete interface shown in Figure 11–25.

None of these operations are abstract because we neither expect nor desire any subclasses.

Notice that this class exports several primitive operations (such as `displayTime` and `refresh`) but also exposes the composite operation `display`, whose presence greatly simplifies the action of clients that must interact with instances of `Display Manager`.

`Display Manager` ultimately uses the resources of the `LCD Device` class, which, as we described earlier, serves as a skin over the underlying hardware. In this manner, `Display Manager` raises our level of abstraction by providing a protocol that speaks more directly to the nature of the problem space.

The User Interface Mechanism

The focus of our last major release is the tactical design and implementation of the classes `Keypad` and `InputManager`. Similar to the `LCD Device` class, the `Keypad` class serves as a skin over the underlying hardware, which thereby relieves the `InputManager` of the nasty details of talking directly to the hardware. Decoupling these two abstractions also makes it far easier to replace the physical input device without destabilizing our architecture.

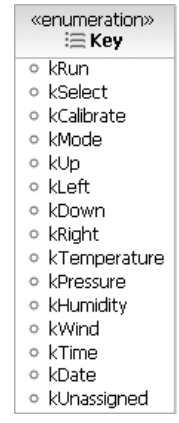

Figure 11–26 The Design of the `Key` Enumeration Class

We start with a declaration that names the physical keys in the vocabulary of our problem space. An enumeration class, `Key`, is defined as shown in Figure 11–26.

We use the `k` prefix to avoid name clashes with literals defined in `SensorName`.

Continuing, we may capture our abstraction of the `Keypad` class as shown in Figure 11–27.

The protocol of this class derives from our earlier analysis. We have added the operation `inputPending` so that clients can query if user input exists that has not yet been processed.

The class `InputManager` has a similarly sparse interface (Figure 11–28).

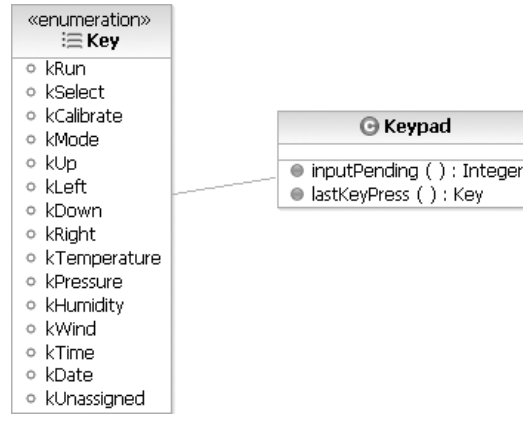

Figure 11–27 The Design of the `Keypad` Class

Figure 11–28 The Design of the `InputManager` Interface

As we will see, most of the interesting work of this class is carried out in the implementation of its finite state machine.

As we illustrated earlier in Figure 11–14, instances of the `Sampler`, `InputManager`, and `Keypad` classes collaborate to respond to user input. To integrate these three abstractions, we must subtly modify the interface of the `Sampler` class to include a new object, `repInputManager` (Figure 11–29).

Through this design decision, we establish an association among instances of the `Sensors`, `Display Manager`, and `InputManager` classes at the time we construct an instance of `Sampler`. This design asserts that instances of `Sampler` must always have a collection of sensors, a display manager, and an input manager.

We must also incrementally modify the implementation of the function `Sampler::sample`.

```
void Sampler::sample(Tick t)
{
  repInputManager.processKeyPress();
  for (SensorName name = Direction; name <= Pressure; name++)
    for (unsigned int id = 0; id <
       repSensors.numberOfSensors(name); id++)
      if (!(t % samplingRate(name)))
        repDisplayManager.display(repSensors.sensor(name,
        id));
}
```

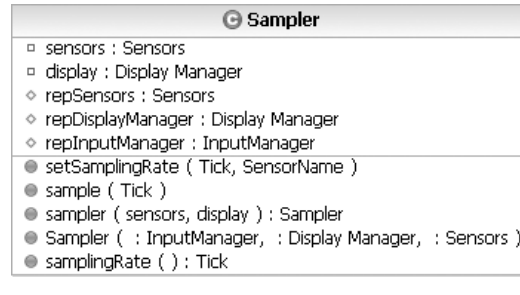

Figure 11–29 The Revised Design of the `Sampler` Interface

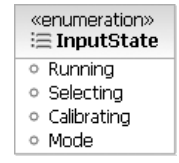

Figure 11–30 The Design of the `InputState` Enumeration Class

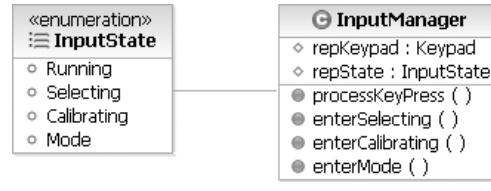

Figure 11–31 `InputManager` with the `InputState` Class Design

Here we have added an invocation to `processKeyPress` at the beginning of every time frame.

The `processKeyPress` operation is the entry point to the finite state machine that drives the instances of this class. Ultimately, there are two approaches we can take to implement this or any other finite state machine: We can explicitly represent states as objects (and thereby depend on their polymorphic behavior), or we can use enumeration literals to denote each distinct state.

For modest-sized finite state machines such as the one embodied by the `InputManager` class, it is sufficient for us to use the latter approach. Thus, we might first introduce the names of the class's outermost states (Figure 11–30).

Next, we introduce some protected helper functions (Figure 11–31).

Finally, we can begin to implement the state transitions we first introduced in Figure 11–12.

```
void InputManager::processKeyPress()
{
  if (repKeypad.inputPending()) {
    Key key = repKeypad.lastKeyPress();
    switch (repState) {
      case Running:
        if (key == kSelect)
          enterSelecting();
        else if (key == kCalibrate)
          enterCalibrating();
        else if (key == kMode)
```

```
            enterMode();
          break;
        case Selecting:
          ...
          break;
        case Calibrating:
           ...
          break;
        case Mode:
           ...
           break;
        }
    }
}
```

The implementation of this function and its associated helper functions thus parallels the state transition diagram shown in Figure 11–12.

11.4 Post-Transition

The complete implementation of this basic Weather Monitoring System is of modest size, encompassing only about 20 classes. However, for any truly useful piece of software, change is inevitable. Let's consider the impact of two enhancements to the architecture of this system.

Our system thus far provides for the monitoring of many interesting weather conditions, but we may soon discover that users want to measure rainfall as well. What is the impact of adding a rain gauge?

Happily, we do not have to radically alter our architecture; we must merely augment it. Using the architectural view of the system from Figure 11–14 as a baseline, to implement this new feature, we must do the following.

- Create a new class, `RainFall Sensor`, and insert it in the proper place in the sensor class hierarchy (a `RainFall Sensor` is a kind of `Historical Sensor`).
- Update the enumeration `SensorName`.
- Update the `Display Manager` so that it knows how to display values of this sensor.
- Update the `InputManager` so that it knows how to evaluate the newly-defined key `RainFall`.
- Properly add instances of this class to the system's `Sensors` collection.

We must deal with a few other small tactical issues needed to graft in this new abstraction, but ultimately, we need not disrupt the system's architecture or its key mechanisms.

Let's consider a totally different kind of functionality. Suppose we desire the ability to download a day's record of weather conditions to a remote computer. To implement this feature, we must make the following changes.

- Create a new class, `SerialPort`, responsible for managing a port used for serial communication.
- Invent a new class, `Report Manager`, responsible for collecting the information required for the download. Basically, this class must use the resources of the collection class `Sensors` together with its associated concrete sensors.
- Modify the implementation of `Sampler::sample` to periodically service the serial port.

It is the mark of a well-engineered object-oriented system that making this change does not rend our existing architecture but, rather, reuses and then augments its existing mechanisms.