Data Acquisition: Weather Monitoring Station

In a weather monitoring system, data is collected from sensors that measure various weather parameters like temperature, humidity, wind speed, etc. This data acquisition process involves converting the measured parameters into electrical signals, which are then acquired by data acquisition hardware. Control software interprets these signals for analysis and display.

Using object-oriented design for such systems allows for a clear separation between the hardware responsible for data collection and the software that analyzes the data. This separation enables flexibility, as sensors and devices can be added or replaced without disrupting the application's architecture.

Interfaces play a crucial role in this architecture, acting as a layer between the hardware and the application. This isolation ensures that changes in the hardware do not affect the application's functionality.

In the example of a Weather Monitoring System, sensors and devices measure weather conditions, and the data is then analyzed and displayed. This object-oriented approach provides a reusable architecture that effectively separates the hardware from the application, making it easier to maintain and upgrade.

1 INCEPTION

Even small systems, like the Weather Monitoring System, benefit from an object-oriented approach. Instead of focusing on data flow and mappings, we organize functionality into classes representing real-world elements, like sensors and weather conditions. This makes the system easier to understand, maintain, and expand. We'll explore this approach in the next sections, showing how it aligns with key principles of object-oriented development.

1.1 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.

1.2 Defining the Boundaries of the Problem

To narrow down our problem and focus on software aspects, let's make some assumptions about the hardware:

- The processor can be either a PC or a handheld device.
- Time and date information comes from a clock.
- Temperature, barometric pressure, and humidity are measured by remote sensors.
- Wind direction and speed are measured using a wind vane and cups on a boom.
- Users interact with the system through a keypad.
- The display is a standard LCD graphic device.
- A timer interrupts the system every 1/60 second.

We've outlined a hardware setup to provide a foundation for our software design. By allocating some tasks to hardware components like sensors and displays, we can streamline our focus on software development. Although we could shift some responsibilities to software, like handling user input directly instead of using hardware keypads, in this case, the choice doesn't significantly affect our overall software architecture. Object-oriented systems, like the one we're building, abstract away from hardware specifics, focusing instead on representing problem entities effectively. Changing hardware details mainly affects the lower layers of our system's abstraction.

We can create a class to handle interactions with hardware interfaces, like accessing the current time and date. This class should be responsible for maintaining the current time and date, providing methods to retrieve this information. We can break down these responsibilities into two main services: `currentTime` and `currentDate`. The `currentTime` operation would return a string showing the current hour, minute, and second, formatted like "13:56:42". Similarly, the `currentDate` operation would return a string showing the current month, day, and year, formatted like "6-10-93".

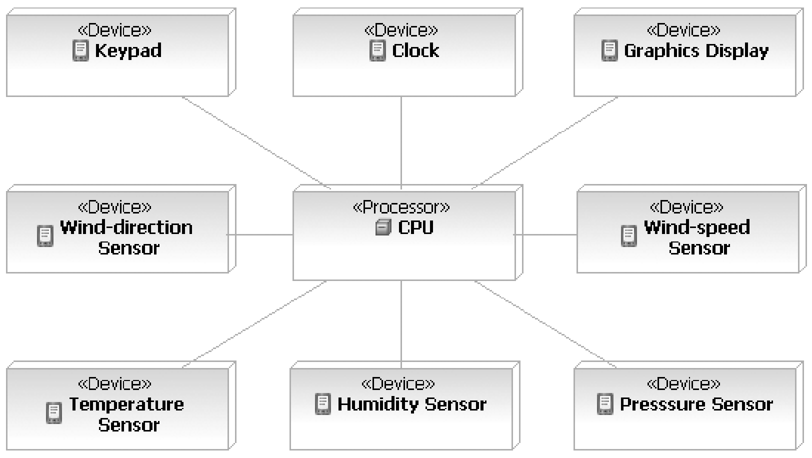

Figure 11–1 The Deployment Diagram for the Weather Monitoring System

We can enhance our time and date abstraction by allowing clients to choose between a 12- or 24-hour format using a modifier called **setFormat**. This separation between interface and implementation ensures clarity and flexibility for clients. By focusing on the public interface, we maintain a clear boundary between the abstraction and its underlying platform. Additionally, our class should be able to set the date and time, requiring new services like **setHour**, **setMinute**, **setSecond**, **setDay**, **setMonth**, and **setYear**.

Here's a summary of our time and date class:

- **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

The lifecycle of instances of this class can be represented with a state transition diagram, as shown in Figure 11–2. Upon initialization, an instance resets its time and date attributes and enters the Running state in 24-hour mode. While in the Running state, it can toggle between 12- and 24-hour mode with the `setFormat` operation. Regardless of its mode, setting the time or date renormalizes its attributes, and requesting its time or date calculates a new string value.

With this detailed behavior specification, we can offer this abstraction for use in various scenarios with other clients we might discover during analysis. Before exploring these scenarios, let's define the behavior of the other tangible objects in our system.

The **TemperatureSensor** class mimics the behavior of the hardware temperature sensors in our system. Initial analysis suggests the following abstraction's outside view:

Class Name: Temperature Sensor
Responsibility: Track the current temperature.
Operations:

- currentTemperature
- setLowTemperature
- setHighTemperature

Attribute: temperature

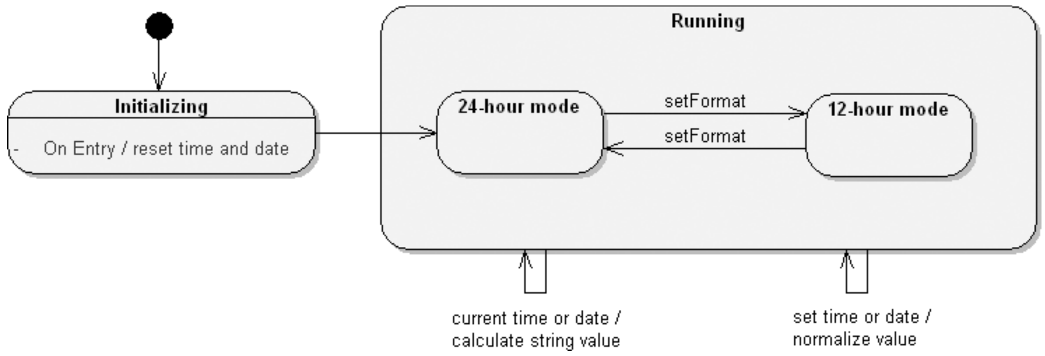
Running
Initializing
- On Entry / reset time and date
24-hour mode
setFormat
setFormat
12-hour mode
current time or date / calculate string value
set time or date / normalize value

Figure 11–2 The `TimeDate` Lifecycle

1.3 Sensors

After defining the system's boundaries, we explore various scenarios to understand its usage better. We identify primary and secondary use cases from the perspective of system clients:

Primary Use Cases:

1. Monitoring basic weather measurements (e.g., wind speed, temperature).
2. Monitoring derived measurements (e.g., wind chill, dew point).
3. Displaying highest and lowest measurement values.
4. Setting time and date.
5. Calibrating sensors.
6. Powering up the system.

Secondary Use Cases:

1. Handling power failure.
2. Managing sensor failure.

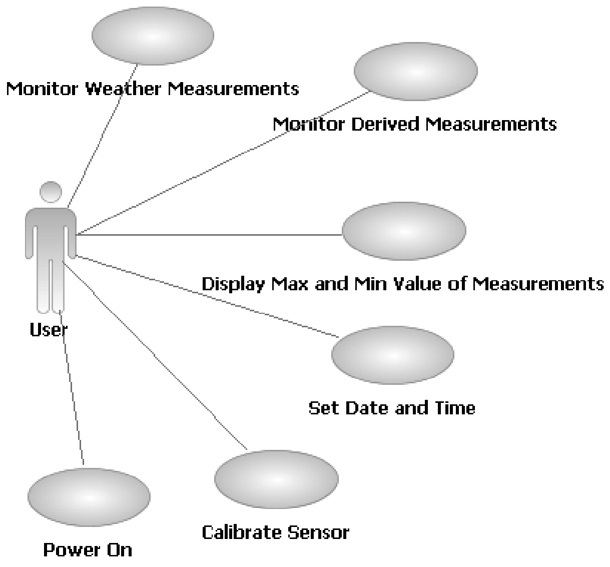
Monitor Weather Measurements
Monitor Derived Measurements
Display Max and Min Value of Measurements
User
Set Date and Time
Calibrate Sensor
Power On

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

2 ELABORATION

Let's delve into some of these scenarios to understand how the system behaves in different situations. We'll focus on elucidating the system's behavior without delving into its design details.

2.1 Weather Monitoring System Use Cases

In the Weather Monitoring System, the primary function is to monitor basic weather measurements. However, due to system constraints, measurements cannot be taken more frequently than 60 times a second. Fortunately, most weather conditions change slowly enough to accommodate this limitation. Our analysis suggests the following sampling rates for capturing changing conditions:

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

We've decided that each primary sensor class should not handle timed events. Therefore, we need an external agent to collaborate with these sensors to carry out the monitoring scenario. For now, we'll defer specifying the behavior of this agent, as it's a design issue rather than an analysis one.

The interaction diagram illustrates this scenario. When the agent begins sampling, it polls each sensor sequentially, intentionally skipping certain sensors to sample them at a slower rate. Polling each sensor instead of letting each sensor control its thread ensures more predictable system execution. We'll name this agent an instance of the Sampler class, reflecting its role in the system's behavior.

Next, we need to determine which object is responsible for displaying the sampled values on the LCD Device class instance. We have two options: either have each sensor display itself (common in MVC-like architectures) or have a separate object handle this behavior. We opt for the latter to encapsulate all display layout decisions in one class. This addition completes our analysis products.

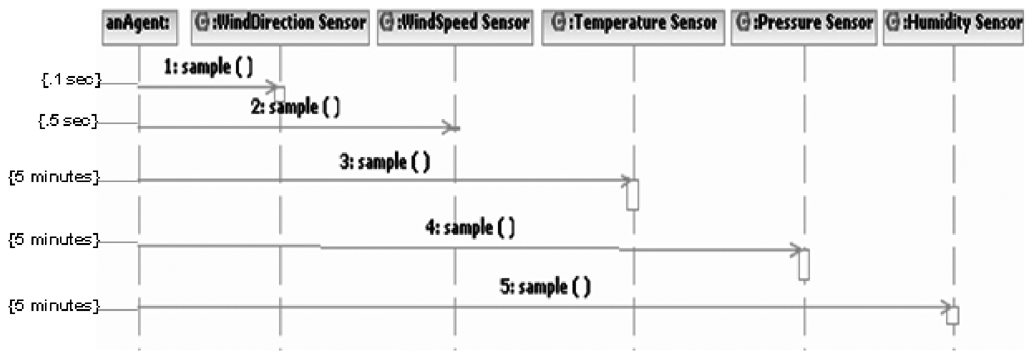

Figure 11–8 A Scenario for Monitoring Basic Measurements

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

We define Wind Chill and Dew Point as classes rather than simple nonmember functions because they encapsulate behavior and state, adhering to the principles of object-oriented design. Each instance of Wind Chill or Dew Point provides behavior (calculating respective values) and maintains state (association with specific sensor instances). This approach enhances reusability and maintains a clear separation of concerns.

Considering user interaction scenarios, defining proper gestures for an embedded controller like the Weather Monitoring System involves artistry, and prototyping is crucial for mitigating risks in interface design. Implementing decisions with an object-oriented architecture allows for flexible adjustments to user interface choices without disrupting the overall design fabric.

These use case scenarios outline various interactions a user might have with the Weather Monitoring System:

1. **Display Max and Min Value of Measurements:**
    - User selects a measurement type.
    - User chooses to display either the highest or lowest value for the past 24 hours.
    - System displays the selected value and its time of occurrence.
    - User can repeat the process or exit.
2. **Set Date and Time:**
    - User selects whether to set the time or date.
    - User adjusts the selected field (e.g., hours, months) using directional keys.
    - System allows navigation and modification of fields.
    - User can confirm or cancel the operation.
3. **Calibrate Sensor:**
    - User selects a sensor to calibrate.
    - User chooses between high or low calibration.
    - System displays and allows adjustment of the selected value.
    - User can confirm or cancel the calibration.
4. **Set Unit of Measurement:**
    - User selects whether to set the unit for wind speed or temperature.
    - User toggles between available units.
    - System updates the unit for the selected measurement.
    - User can confirm or cancel the change.

Each scenario involves user input, system response, and options for confirmation or cancellation, ensuring a user-friendly interaction experience.

2.2 The Architecture Framework

In the architecture framework for the Weather Monitoring System, we consider two main patterns: synchronization of autonomous actors and time-frame-based processing.

1. **Synchronization of Autonomous Actors:**
   - In this pattern, each object acts relatively independently, serving as a thread of control.
   - New sensor objects could be created, responsible for taking samples and reporting back to a central agent.
   - Suitable for distributed systems with samples from remote locations, allowing local optimization of sampling.
   - However, it's not ideal for hard real-time systems where complete predictability is required.
2. **Time-Frame-Based Processing:**
   - Time is divided into fixed-length frames, each further divided into subframes for specific functional behavior.
   - Allows for more rigorous control over the order of events.
   - Provides predictability, suitable for systems like the Weather Monitoring System.

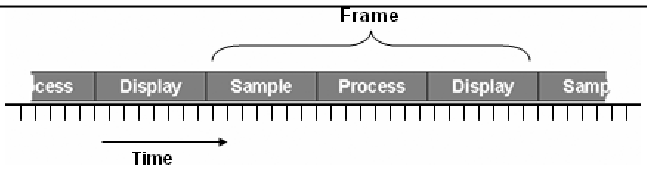

Figure 11–13 Time-Frame Processing

In the architecture diagram, we see how all key abstractions collaborate with each other within this time-frame-based processing model. While not every class and relationship is shown, it illustrates the interaction between various components. A new class called Sensors is introduced to represent the collection of physical sensors in the system, allowing other agents like Sampler and InputManager to associate with the entire sensor collection conveniently.

3 CONSTRUCTION

“1”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.

“1”During the construction phase of our architecture, it's essential to focus on prototyping the central behavior carried out by the Sampler and Timer classes. By concretely prototyping these classes, we can validate our assumptions and ensure that our design aligns with the system requirements and architectural principles. This process helps us identify any potential issues early on and allows for necessary adjustments before moving forward with full implementation.

3.1 The Frame Mechanism

In the construction phase, we refine the interface of the Timer class to dispatch a callback function, allowing for precise timing in our architecture. We attach a callback function to the timer using the setCallback operation, and then initiate the timer's behavior with startTiming, ensuring that the callback function is dispatched every 1/60 of a second.

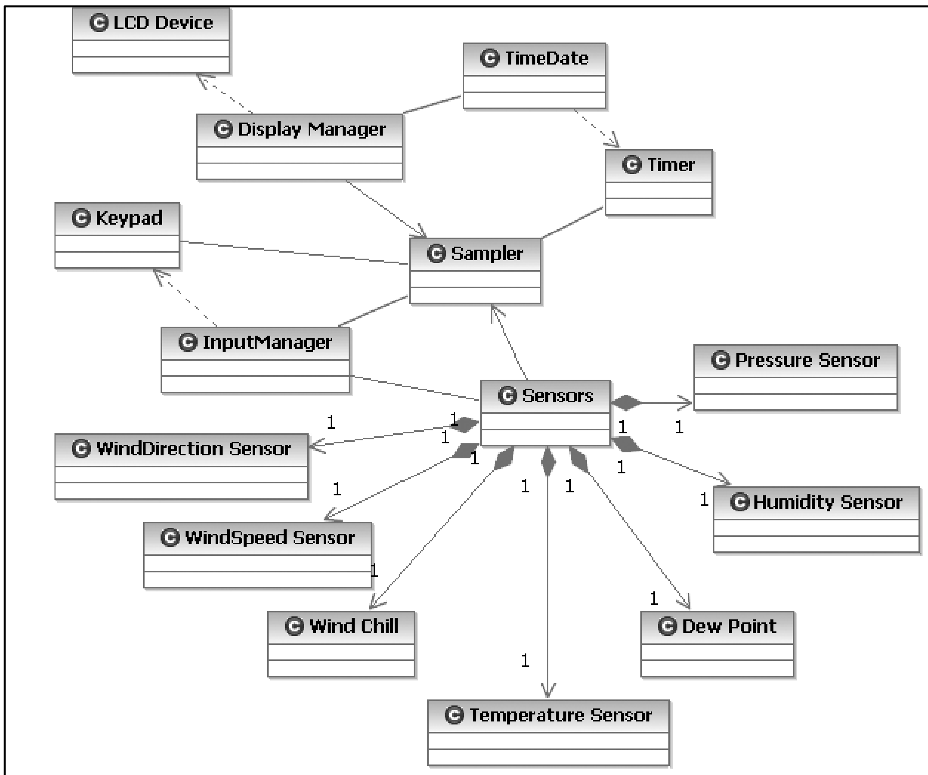

Figure 11–14 The Architecture of the Weather Monitoring System

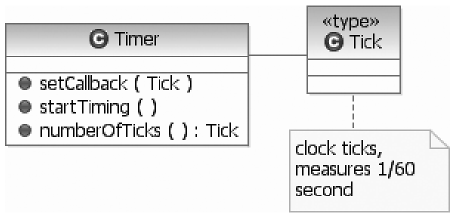

Figure 11–15 The Design of the `Timer` Class

To further tie our architecture together, we introduce a new declaration naming the various sensors in the system using the SensorName enumeration class. Then, we define the interface of the Sampler class, including the modifier setSamplingRate and its selector samplingRate, allowing clients to dynamically adjust sampling behavior.

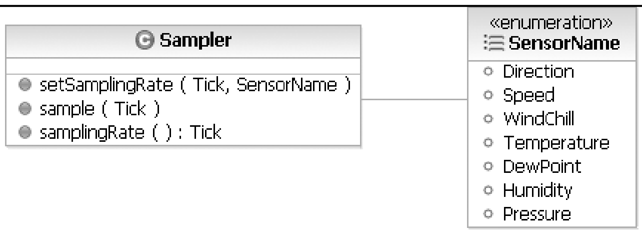

Figure 11–16 The Interface of the `Sampler` Class

To connect the Timer and Sampler classes, we write some C++ glue code, declaring an instance of Sampler and a nonmember function to handle sampling. In our main function, we attach the callback function to the timer and start the sampling process.

Next, we provide an interface for the Sensors class, assuming the existence of various concrete sensor classes. The Sensors class acts as a collection type, subclassing the Collection class. We limit the operations exposed to clients of the Sensors class, as sensors are only added and never removed from the collection.

In the Sampler class, we specify associations with the Sensors and Display Manager classes and revise the declaration of the Sampler object to connect it with the specific collection of sensors and the display manager used in the system. This ensures that the Sampler agent is properly linked with the necessary components for sampling and displaying data.

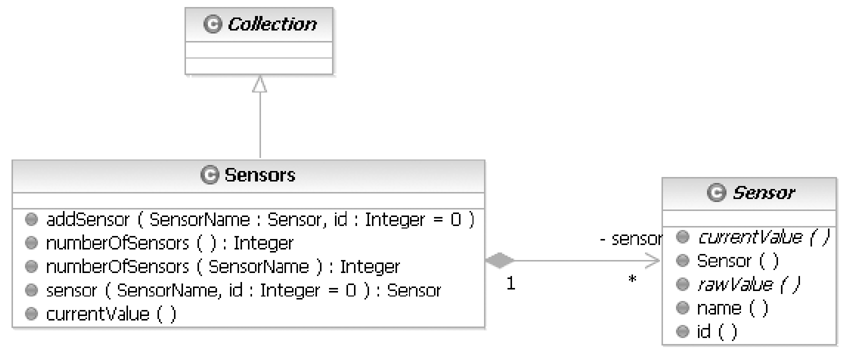
Collection
Sensors
addSensor (SensorName : Sensor, id : Integer = 0)
numberOfSensors ( ) : Integer
numberOfSensors (SensorName) : Integer
sensor (SensorName, id : Integer = 0) : Sensor
currentValue ( )
- sensor
1
*
Sensor
currentValue ( )
Sensor ( )
rawValue ( )
name ( )
id ( )

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

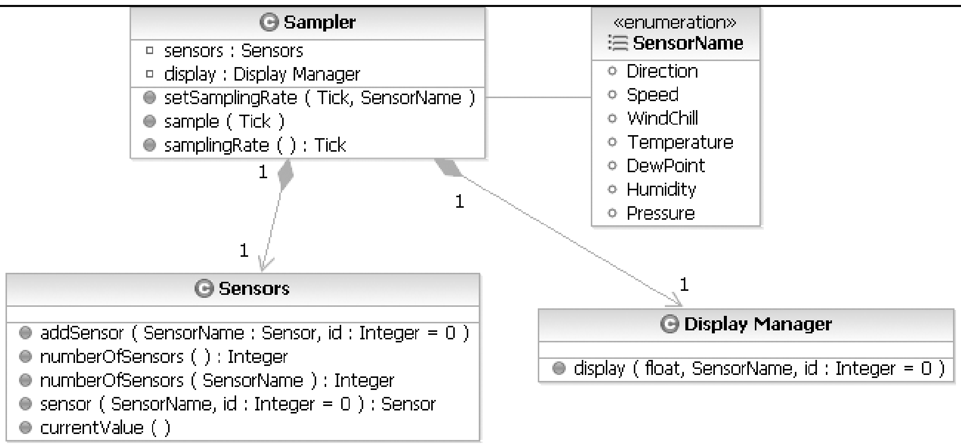
Sampler
sensors : Sensors
display : Display Manager
setSamplingRate (Tick, SensorName)
sample (Tick)
samplingRate ( ) : Tick
«enumeration»
SensorName
Direction
Speed
WindChill
Temperature
DewPoint
Humidity
Pressure
1
1
1
1
Sensors
addSensor (SensorName : Sensor, id : Integer = 0)
numberOfSensors ( ) : Integer
numberOfSensors (SensorName) : Integer
sensor (SensorName, id : Integer = 0) : Sensor
currentValue ( )
Display Manager
display (float, SensorName, id : Integer = 0)

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

In the implementation of the Sampler class's key operation, sample, we iterate through each type of sensor and each unique sensor within that type in the collection. For each sensor encountered, we check if it's time to sample its value based on the sampling rate. If so, we retrieve the sensor's current value from the collection and pass it to the associated display manager for presentation.

This operation relies on polymorphism, as the currentValue operation is defined for the base class Sensor, allowing us to access the current value of any type of sensor uniformly. Additionally, it relies on the display operation defined for the Display Manager class to present the sampled values on the display.

With this refinement in our architecture, we update the class diagram to highlight the frame mechanism, showing how the Sampler class interacts with the sensors and the display manager.

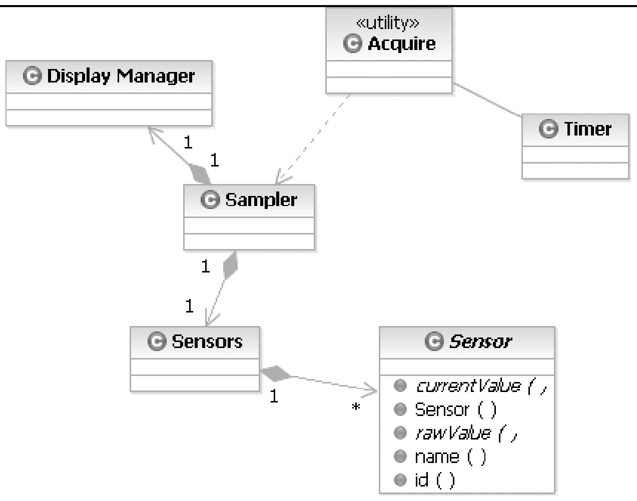

Figure 11–19 The Frame Mechanism

Having validated our architecture through various scenarios, we can now proceed with the incremental development of the system's functionality.

3.2 Release Planning

Starting with the highest-risk components, we propose a sequence of releases to gradually build the system's functionality:

- 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.

By following this sequence, we address the most challenging aspects of the system first and gradually build upon them, minimizing risks and ensuring a solid foundation for further development.

Developing the minimal functionality release is akin to creating a prototype that gives us a glimpse into how our system will operate in reality. By implementing small parts of each key abstraction, we address the highest risks upfront, ensuring that our system's architecture is sound.

This approach provides several benefits:

1. **Early Feedback**: We gain early insights into the effectiveness of our design by having a runnable system. This allows us to validate our assumptions and make necessary adjustments before investing more resources.

2. **Integration Testing**: Implementing a vertical slice forces us to integrate hardware and software components early on, helping us identify and resolve any compatibility issues or impedance mismatches.

3. **User Validation**: Getting early feedback from real users on the look and feel of the system allows us to make improvements based on their perspectives. This early involvement enhances user satisfaction and acceptance of the final product.

Overall, this iterative approach mitigates risks, fosters collaboration, and ensures that our final product meets the needs of both users and stakeholders.

Focusing on completing this release through tactical implementation means we will not bother with exposing any more of its structure.

“1”We will now turn to elements of later releases because they reveal some interesting insights about the development process.

“1”Instead, we'll shift our attention to elements of subsequent releases. These later releases will provide valuable insights into the development process and help us refine our approach based on evolving requirements and feedback.

3.3 The Sensor Mechanism

In this step, we're finalizing the design of the sensor classes. We're making sure each sensor has the right features and abilities. For example, we're giving each sensor a way to identify itself and connect to its interface. This helps the system know which sensor is which.

By completing this part, we're getting closer to having all the sensors fully working in our system. We're following a step-by-step approach to ensure everything fits together smoothly and meets the needs of the project.

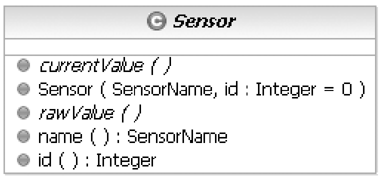

Figure 11–20 The Design of the `Sensor` Class

We're simplifying the Display Manager's function to take just one argument, a reference to a Sensor object. This change makes the system's interactions smoother and more consistent.

The Calibrating Sensor class, which is a subclass of Sensor, adds two new operations: setHighValue and setLowValue. It also implements the currentValue function.

Moving on, the Historical Sensor class, also a subclass of Sensor, includes operations that rely on the TimeDate class for time-related functions. However, Historical Sensor remains abstract because we haven't finished defining the rawValue function, which will be the responsibility of a concrete subclass.

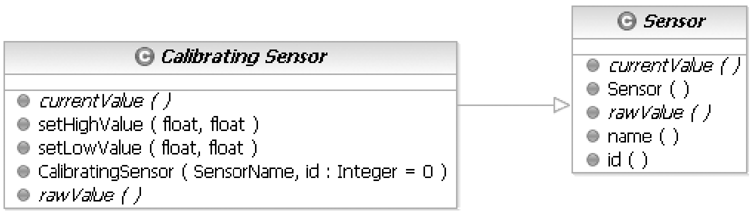
Calibrating Sensor
currentValue ( )
setHighValue (float, float)
setLowValue (float, float)
CalibratingSensor (SensorName, id : Integer = 0)
rawValue ( )
Sensor
currentValue ( )
Sensor ( )
rawValue ( )
name ( )
id ( )

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

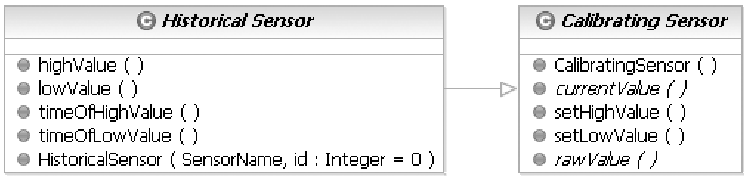

Figure 11–22 The Design of the `Historical Sensor` Class

We're simplifying the Display Manager's function to take just one argument, a reference to a Sensor object. This change makes the system's interactions smoother and more consistent.

The Calibrating Sensor class, which is a subclass of Sensor, adds two new operations: setHighValue and setLowValue. It also implements the currentValue function.

Moving on, the Historical Sensor class, also a subclass of Sensor, includes operations that rely on the TimeDate class for time-related functions. However, Historical Sensor remains abstract because we haven't finished defining the rawValue function, which will be the responsibility of a concrete subclass.

3.4 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 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.

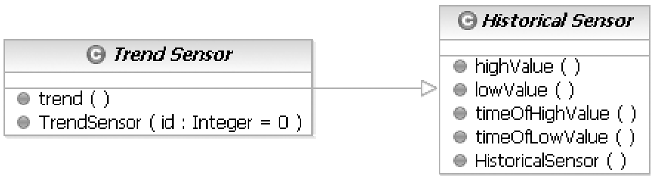

Figure 11–23 The Design of the `Trend Sensor` Class

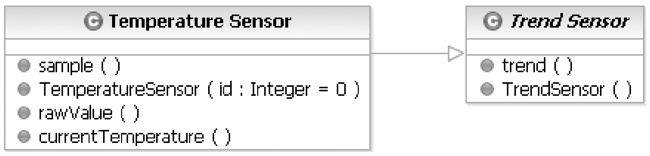

Figure 11–24 The Design of the `Temperature Sensor` Class

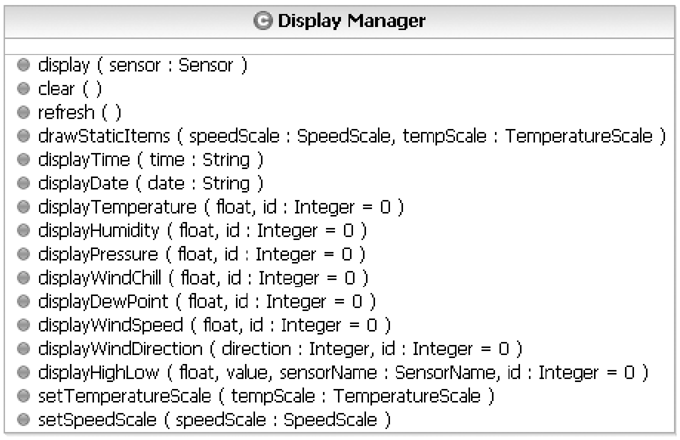
Display Manager
display (sensor : Sensor)
clear ( )
refresh ( )
drawStaticItems (speedScale : SpeedScale, tempScale : TemperatureScale)
displayTime (time : String)
displayDate (date : String)
displayTemperature (float, id : Integer = 0)
displayHumidity (float, id : Integer = 0)
displayPressure (float, id : Integer = 0)
displayWindChill (float, id : Integer = 0)
displayDewPoint (float, id : Integer = 0)
displayWindSpeed (float, id : Integer = 0)
displayWindDirection (direction : Integer, id : Integer = 0)
displayHighLow (float, value, sensorName : SensorName, id : Integer = 0)
setTemperatureScale (tempScale : TemperatureScale)
setSpeedScale (speedScale : SpeedScale)

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

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 inter- act 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.

3.5 The User Interface Mechanism

In our final major release, we concentrate on designing and implementing the Keypad and InputManager classes. The Keypad class acts as an intermediary for the underlying hardware, shielding the InputManager from the intricate hardware operations. This separation simplifies the system and ensures that replacing the physical input device won't disrupt the overall architecture.

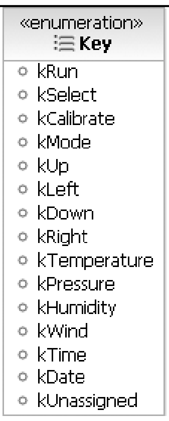

Figure 11–26 The Design of the `Key` Enumeration Class

We begin by defining an enumeration class called Key to represent the physical keys relevant to our system. The class Keypad is then outlined, reflecting our previous analysis. We introduce the operation inputPending to allow clients to check for unprocessed user input. Similarly, the class InputManager also has a minimal interface.

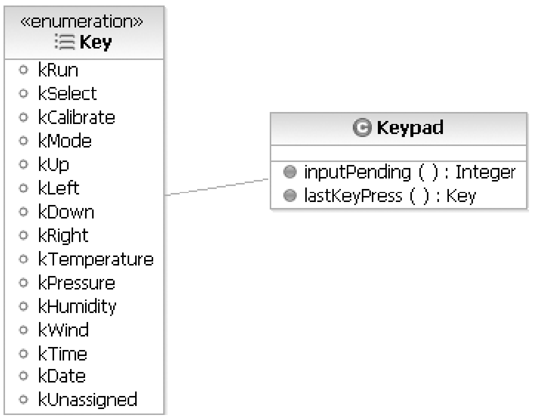

Figure 11–27 The Design of the `Keypad` Class

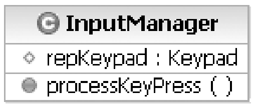

Figure 11–28 The Design of the `InputManager` Interface

The implementation of the InputManager class involves managing its finite state machine, which handles user input. As depicted in our previous diagram (Figure 11–14), instances of Sampler, InputManager, and Keypad collaborate to respond to user actions. To integrate these classes, we adjust the interface of the Sampler class to include a reference to an InputManager object, ensuring that every Sampler instance has access to 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));
}
```

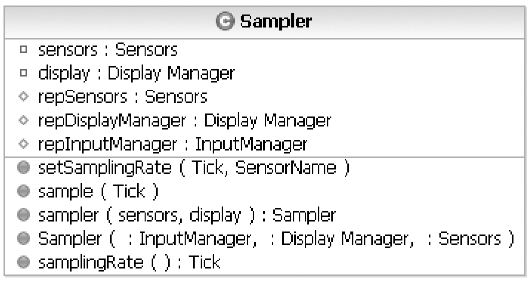
Sampler
sensors : Sensors
display : Display Manager
repSensors : Sensors
repDisplayManager : Display Manager
repInputManager : InputManager
setSamplingRate (Tick, SensorName)
sample (Tick)
sampler (sensors, display) : Sampler
Sampler (: InputManager, : Display Manager, : Sensors)
samplingRate ( ) : Tick

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

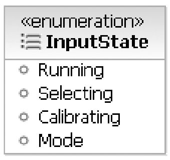

Figure 11–30 The Design of the `InputState` Enumeration Class

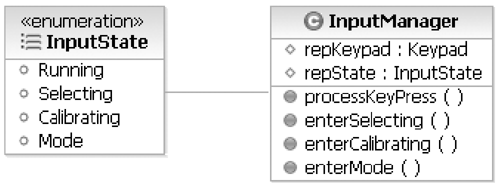

Figure 11–31 `InputManager` with the `InputState` Class Design

In every time frame, we start by invoking the `processKeyPress` operation, which is the entry point to the finite state machine driving this class. There are two ways to implement this: representing states as objects or using enumeration literals. For simplicity, we'll use the latter. We introduce the names of the outermost states first. Then, we define some protected helper functions. Finally, we implement the state transitions outlined earlier.

```
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.

4 TRANSISTION

Adding a rain gauge to the Weather Monitoring System involves several steps without drastically altering the existing architecture. Here's what we need to do based on the current system architecture:

1. **Create a New Class**: Develop a new class called RainFallSensor and position it appropriately within the sensor class hierarchy. Since a RainFallSensor is a type of HistoricalSensor, it should inherit from that class.
2. **Update Enumeration**: Update the SensorName enumeration to include the new RainFall sensor.
3. **Update Display Manager**: Modify the Display Manager to handle the display of RainFall sensor values. This ensures that the system can visually represent rainfall data.

4. **Update Input Manager**: Update the Input Manager to recognize and process user input related to the RainFall sensor. This allows users to interact with the system effectively when dealing with rainfall measurements.
5. **Add Instances to Sensors Collection**: Integrate instances of the RainFallSensor class into the Sensors collection of the system. This ensures that the system can access and utilize data from the rain gauge.

Although there may be some tactical adjustments required to seamlessly integrate the new feature, the overall architecture and key mechanisms of the system remain intact.

Now, let's consider another enhancement: enabling the system to download a day's weather data to a remote computer. Here's what we need to do for this feature:

1. **Create SerialPort Class**: Develop a new class called SerialPort responsible for managing communication over a serial port, facilitating data transfer to the remote computer.
2. **Introduce Report Manager**: Invent a ReportManager class tasked with gathering the necessary information for the download. It should utilize resources from the Sensors collection and its associated sensors to compile the weather data.
3. **Modify Sampler Implementation**: Adjust the Sampler::sample method to periodically service the serial port, enabling the system to transmit data to the remote computer.

By implementing these changes, the existing architecture remains intact, demonstrating the flexibility and adaptability of the object-oriented design.

The complete implementation of this basic Weather Monitoring System is of modest size, encompassing only about 20 classes. However, for any truly useful 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 newlydefined 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.