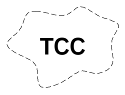
CHAPTER 3

Case Study: Weather Station

This chapter begins an in-depth case study of a simple weather monitoring system. Although this case study is ficticious, it has nevertheless been constructed with a high degree of realism. We will encounter the problems of time-pressure, legacy code, poor and mutating specifications, new untried technologies, etc. Our goal is to demonstrate how object oriented design and UML are used in the real world of software engineering.

The Cloud Company



The Cloud Company has been the leader in industrial weather monitoring systems (WMS) for the past several years. Their flagship product has been a WMS that keeps track of temperature, humidity, barometric pressure, wind speed and direction, etc. The system displays these readings in real time on a display. It also keeps track of historical information on an hourly and daily basis. This historical data can be pulled up on the display at the request of the user.

The primary customers of Cloud Company products have been the aviation, maritime, agricultural, and broadcast industries. For these industries, WMSs are mission critical applications. The Cloud Company has a reputation for building highly reliable products that can be installed in relatively uncontrolled environments. This makes the systems somewhat expensive.

The high cost of these systems has cut the Cloud Company off from customers that do not need, and cannot afford, the high reliability systems that they sell. Cloud Company managers believe that this is a large potential market, and they would like to tap into it.

The Problem. A competitor named Microburst Inc. has recently announced a product line that starts at the low end, and can be incrementally upgraded to higher reliability. This threatens to cut the Cloud Company off from smaller but growing customers. These customers will already be using Microburst products by the time they grow to a size that would allow them to use Cloud Company products.

More frightening still, the Microburst product boasts the ability to be interconnected at the high end. That is, the high end upgrades can be networked together into a wide area weather monitoring system. This threatens to erode the current Cloud Company customer base.

The Strategy. Although Microburst Inc. has successfully demonstrated its low end units at trade shows, they are not offering production quantity shipments for at least six months. This indicates that there may be engineering or production problems that Microburst has not solved. Moreover, the high reliability upgrades promised by Microburst as part of the product line are currently not available. It may be that Microburst has announced a product that it is not ready to market.

If The Cloud Company can announce a low-end upgradable and connectable product, and begin shipping it *within* six months, then they may be able to capture, or at least stall, customers who would otherwise buy Microburst's products. By stalling the market and thereby depriving Microburst of orders, they might be able to compromise Microburst's ability to solve their engineering and manufacturing problems; a very desirable outcome.

The Dilemma. A new low cost and extendable product line requires a significant amount of engineering. The hardware engineers have flatly refused to commit to a six month development deadline. They believe that it will be twelve months before they could see production quanity units.

The marketing managers believe that in twelve months, Microburst will be shipping production quanity, and will be capturing an irretrievable part of Cloud Company's customers.

The Plan. Cloud Company managers have decided to announce their new product line immediately, and to begin accepting orders that will be shipped before six months have elapsed. They have named the new product Nimbus-LC 1.0. Their plan is to repackage the old expensive high-reliability hardware into a new enclosure with a nice LCD touch panel. The high manufacturing cost of these units means that the company will actually lose money on each one that they sell.

Concurrently, the hardware engineers will begin to develop the true low cost hardware which will be available in twelve months. This configuration of the product has been called Nimbus-LC 2.0. When production quantities are available, the Nimbus-LC 1.0 will be phased out.

When a Nimbus-LC 1.0 customer wants to upgrade to a higher level of service, his unit will be replaced with a Nimbus-LC 2.0 at no additional cost. Thus, the company is willing to lose money on this product for six months in order to capture, or at least stall, potential Microburst customers.

The WMS-LC Software

The software project for the Nimbus-LC project is complex. They must create a software product that can use both the existing hardware as well as the low cost 2.0 hardware. Prototype units of the 2.0 hardware will not be available for nine months. Moreover, the processor on the 2.0 board is not likely to be the same as the processor on the 1.0 board. Still, the system must operate identically regardless of which hardware platform it uses.

The hardware engineers will be writing the lowest level hardware drivers, and they need the application software engineers to design the API for these drivers. This API must be available to the hardware engineers within the next four months.

The software must be production ready in six months, and must be working with the 2.0 hardware in twelve months. They want at least six weeks of Q/A for the 1.0 device, so the software engineers really have only twenty weeks to get the software working. Since the hardware platform for the 2.0 version is new, they need eight to ten weeks of Q/A. This eats up most of the three month period between first prototype and final shipment. Thus the software engineers will have very little time to make the new hardware work.

Software Planning Documents. The software engineers have written several documents that describe the Nimbus-LC project. They are:

1. “Nimbus-LC Requirements” on page 101

This document describes the operating requirements of the Nimbus-LC system, as they were understood at the time the project was begun.[1]

2. "Nimbus-LC Elaboration Phase" on page 103

   This document describes the actors and use cases derived from the requirements document.

3. "Nimbus-LC Construction Plan" on page 109

   This document describes the phased construction plan for the software. This plan tries to address the major risks early in the project lifecycle, while assuring that the software will be complete by the necessary deadlines.

Language Selection

The most important constraint upon the language is portability. The short development time, and the even shorter contact that the software engineers will have with the 2.0 hardware demand that both the 1.0 and 2.0 versions use the same software. That is, the source code needs to be identical; or nearly so. If the portability constraint cannot be met by the language, the release of the 2.0 version at the twelve month mark will be in severe jeaopardy.

Fortunately there are few other constraints. The software is not very large, so space is not much of a problem. There are no hard real-time deadlines that are shorter than one second, so speed is not much of an issue. Indeed, the real-time deadlines are so weak that a moderately fast garbage collecting language would not be inappropriate.

The portability constraints, and the lack of any other serious constraints, make the selection of Java quite appropriate. Indeed, there are few languages which can satisfy the portability constraint as well as Java.

However, Java comes with some risks. There must be a JVM[2] for each of the platforms, and both JVMs must work identically. Also, Java is an immature language that is still changing rapidly. These risks are identified in the "Nimbus-LC Construction Plan" on page 109.

1. We all know that the requirements document is the most volatile document in any software project.
2. Java Virtual Machine

Nimbus-LC Software Design

According to the construction plan, one of the major goals of phase I is to create an architecture that will allow the bulk of the software to be independent of the hardware that it controls. Indeed, we want to separate the abstract behavior of the weather station from its concrete implementation.

For example, the software must be able to display the current temperature regardless of the hardware configuration. This implies the design shown in Figure 3-1.

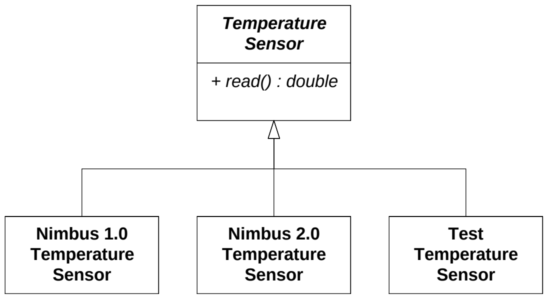


Figure 3-1
Initial Temperature Sensor Design.

An abstract base class named `TemperatureSensor` supplies a polymorphic `read()` function. Derivatives of this base class allow for separate implementations of the `read()` function.

The Test Classes. Notice that there is one derivative for each of the two known hardware platforms. There is also a special derivative named `TestTemperature-Sensor`. This class will be used to test the software in a workstation which is not connected to Nimbus hardware. This allows the software engineers to test their software even when they don't have access to a Nimbus system.

Also, we have very little time to intergrate the Nimbus 2.0 hardware and software together. The Nimbus 2.0 version will be at risk because of this short time frame. By making the Nimbus software work with both the Nimbus 1.0 hardware, and with the test class, we will have made the Nimbus software execute on multiple platforms. This lessens the risk of significant portability issues with the Nimbus 2.0.

Tthe test classes also give us the opportunity to test features or conditions that are hard to capture in the software. For example, we can set up the test classes to produce failures that are difficult to simulate with the hardware.

Making Periodic Measurements. The most common mode of the Nimbus system is when it is displaying current weather monitoring data. Each of the values are updated at their own particular rate. Temperature is updated once per minute, while barometric pressure is updated once every five minutes. Clearly we need some kind of scheduler that will trigger these readings and communicate them to the user. Figure 3-2 shows a possible structure.

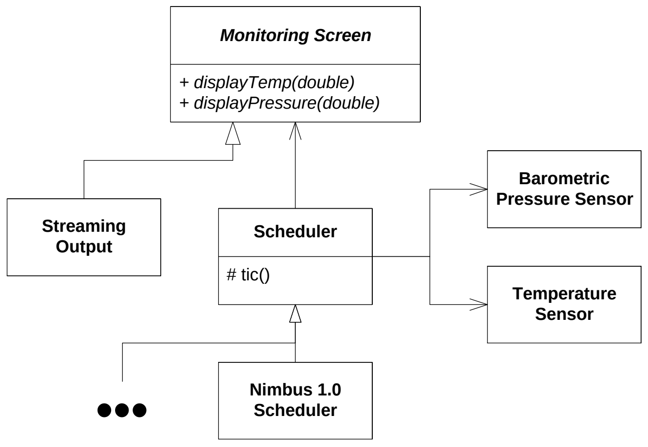


Figure 3-2
Initial Scheduler and Display architecture

We imagine the `Scheduler` to be a base class that has many possible implementation, one for each of the hardware and test platforms. The `Scheduler` has a `tic` function that it expects will be called once every 10ms. It is the responsibility of the derived class to make this call. (See Figure 3-3.) The `Scheduler` counts the `tic()` calls. Once per minute it calls the `read()` function of the `TemperatureSensor`, and passes the returned temperature to the `MonitoringScreen`. For phase I we don't need to show the temperature in a GUI, so the derivative of `MonitoringScreen` simply sends the result to an output stream.

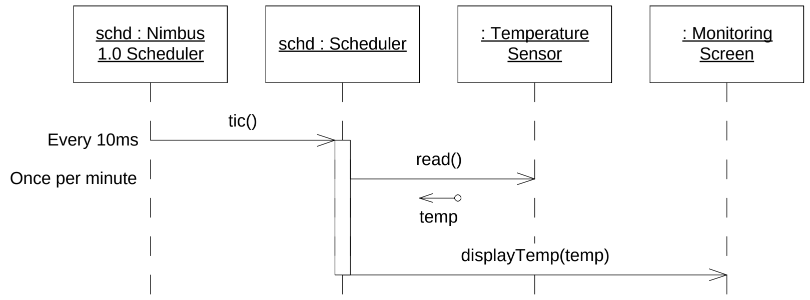


Figure 3-3
Initial Scheduler Sequence Diagram

Barometric Pressure Trend. The requirements document says that we must report the trend of the barometric pressure. This is a value that can have three states: *rising*, *falling*, or *stable*. How do we determine the value of this variable?

According to the Federal Meteorological Handbook[1], barometric pressure trend is calculated as follows:

> If the pressure is rising or falling at a rate of at least 0.06 inch per hour and the pressure change totals 0.02 inch or more at the time of the observation [to be taken once every three hours], a pressure change remark shall be reported.

Where do we put this algorithm? If we put it in the `BarometricPressure-Sensor` class, then that class will need to know the time of each reading, and it will have to keep track of a series of readings going back three hours. Our current design does not allow for this. We could fix this by adding the current time as an argment to the `Read` function of the `BarometricPressureSensor` class, and guaranteeing that that function will be called on a regular basis.

However, this couples the trend calculation to the frequency of user updates. It is not inconceivable that a change to the user interface update scheme could affect the pressure trend algorithm. Also, it is very unfriendly for a sensor to demand that it be read on a regular basis in order to function properly. A better solution needs to be found.

We could have the `Scheduler` keep track of barometric pressure history, and calculate trends at need. However, will we then also put temperature and wind speed his-

1. *Federal Meteorological Handbook No. 1*, Chapter 11, Section 11.4.6 (`http://www.nws.noaa.gov`)

tory in the `Scheduler` class? Every new kind of sensor or history requirement would cause us to change the `Scheduler` class. This is has the makings of a maintenance nighmare™.

Reconsidering the `Scheduler`. Take another look at Figure 3-2. Notice that the `Scheduler` is connected to each of the sensors and to the user interface. As more sensors are added, and as more user interface screens are added, they will have to be added to the `Scheduler` too. Thus, the `Scheduler` is not closed to the addition of new sensors or user interfaces. This is a problem. We would like to design the `Scheduler` so that it is independent of changes and additions to the sensors and user interfaces.

Decoupling the User Interface. User interfaces are volatile. They are subject to the whims of customers, marketting people, and nearly everyone else who comes in contact with the product. It seems very likely that if any part of the system suffers requirements thrashing, it will be the user interface. Therefore we should decouple it first.

There is a standard design pattern for decoupling user interfaces. It is called OBSERVER. The intent of the observer pattern is:

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.[1]

Figure 3-4 and Figure 3-5 show the new design. We have made the UI a dependent of the sensor, so that when the sensor reading changes, the UI will be automatically notified. Notice that the dependency is indirect. The actual observer is an ADAPTER[2] named `TemperatureObserver`. This object is notified by the `TemperatureSensor` when the temperature reading changes. In reponse, the `TemperatureObsever` calls the `DisplayTemp` function of the `MonitoringScreen` object.

This design has nicely decoupled the UI from the `Scheduler`. The `Scheduler` now knows nothing of the UI, and can focues solely upon telling the sensors when to read. The UI binds itself to the sensors, and expects them to report any changes. However, the UI does not know about the sensors themselves. It simply knows about a set of objects that implement the Observable interface. This will allow us to add sensors without making signficant changes to this part of the UI.

1. [GOF95] p. 293
2. [GOF95] p. 139

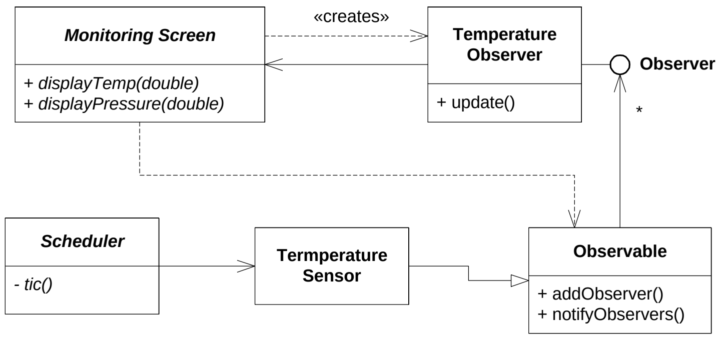


Figure 3-4
Observer decouples UI from Scheduler.

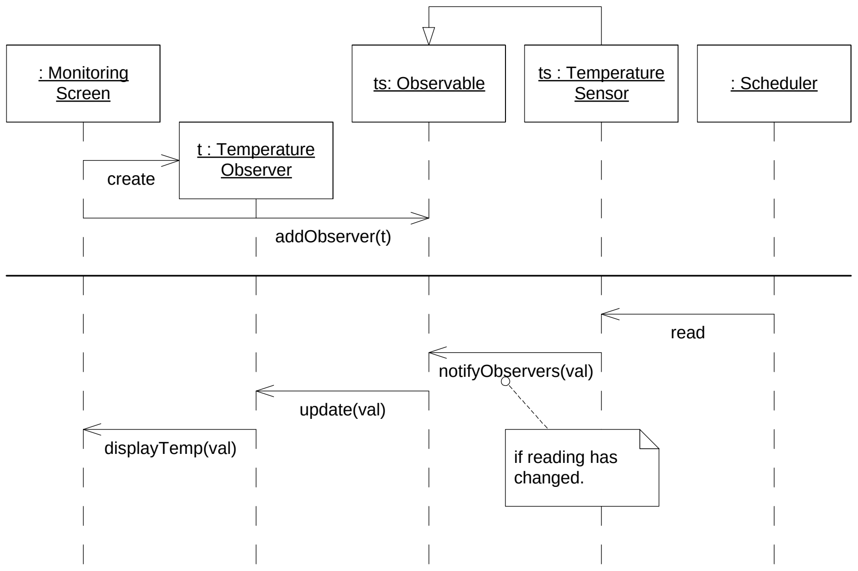
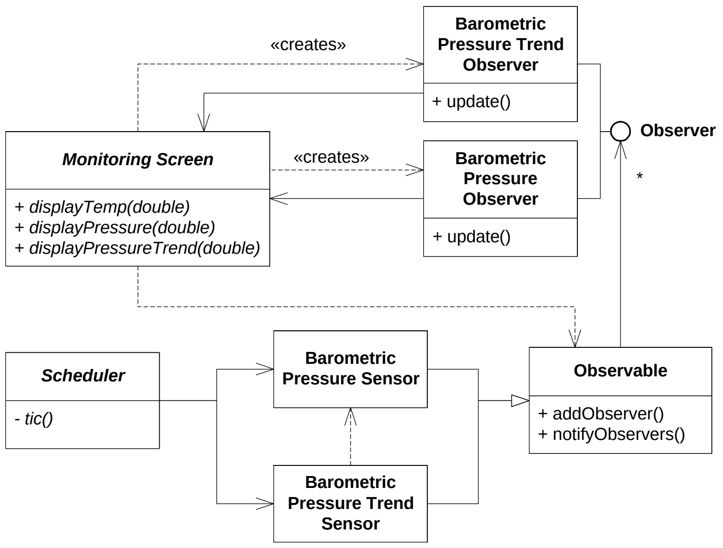


Figure 3-5
Decoupled UI sequence diagram

We have also solved the problem of the barometric pressure trend. This reading can now be calculated by a seperate `BarometricPressureTrendSensor` that observers the `BarometricPressureSensor`. (See Figure 3-6.).



«creates»
Barometric Pressure Trend Observer
+ update()
Observer
*
Monitoring Screen
+ displayTemp(double)
+ displayPressure(double)
+ displayPressureTrend(double)
«creates»
Barometric Pressure Observer
+ update()
Scheduler
- tic()
Barometric Pressure Sensor
Observable
+ addObserver()
+ notifyObservers()
Barometric Pressure Trend Sensor

Figure 3-6
Barometric Pressure Observers

Is this too complex? Have we overengineered the software? If we were certain that the requirements were never going to change, then this solution is probably too complex. However, weather monitoring is the core of our business. We are embarking upon the project because of a signficant change in the industry. We are also entering a new market and using a new technology. There is plenty of reason to suspect that the requirements are going to change over time. Therefore, if our software is going to survive more than two releases, we'd better design it to be changed.

Rethinking the Scheduler -- yet again. One of the major principles of object oriented design is the Open Closed Principle[1] (OCP). This principle says that a class should be extensible without requiring modification. That is, you should be able to change what a class does, without changing the class.

1. [Meyer97] p. 57

The major role of the `Scheduler` is to tell each of the sensors when they should acquire a new value. However, if future requirements force us to add or remove a sensor, the `Scheduler` will need to be changed. Indeed, the `Scheduler` will have to change even if we simply want to change the rate of a sensor. This is an unfortunate violation of the OCP. It seems that the knowledge of a sensor's polling rate belongs to the sensor itself, and not any other part of the system.

We can decouple the Scheduler from the sensors by using the Listener[1] paradigm from the Java class library. This is similar to OBSERVER in that you register to be notified of something; but in this case we want to be notified when a certain event (time) occurs. See Figure 3-7.

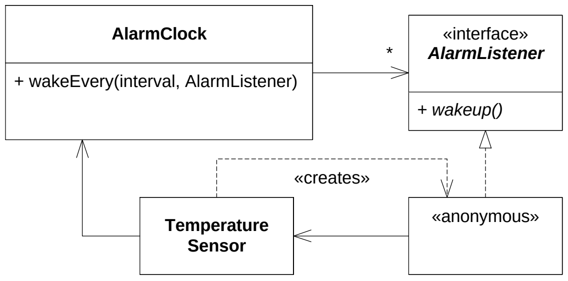


Figure 3-7
Decoupled Alarm clock

Sensors create anonymous[2] ADAPTER classes that implement the `AlarmListener` interface. The sensors then register those adapters with the `AlarmClock` (The class we used to call the `Scheduler`). As part of the registration, they tell the `AlarmClock` how often they would like to be woken up (e.g. every second, or every fifty milliseconds). When that period expires, the `AlarmClock` sends the `wakeup` message to the adapter which then sends the `read` message to the sensor.

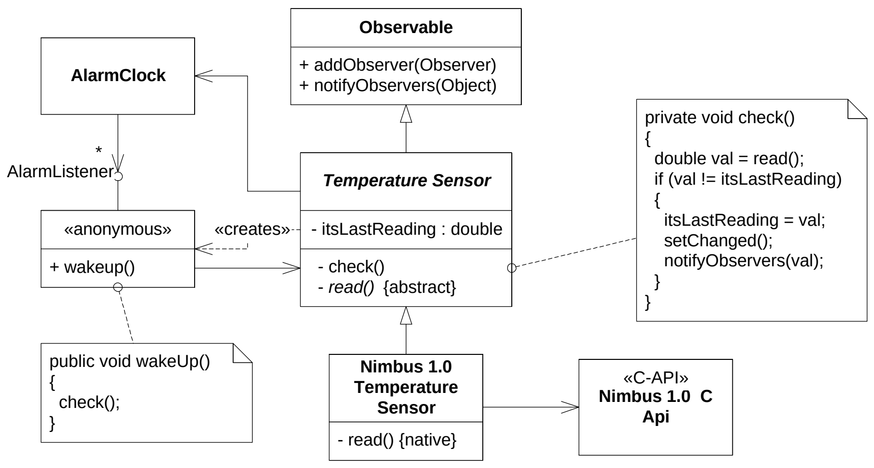
This has completely changed the nature of the `Scheduler` class. In Figure 3-2 it formed the center of our system and knew about most of the other components. But now it simply sits at the side of the system. It knows nothing about the other compo-

1. [JAVA98] p. 360
2. Anonymous classes are a feature of Java. However, the idea can be applied to just about any language. An anonymous class is just a class that implements a well known interface, but which does not have a name of its own. It also has access to the private elements of its creator.

nents. It does one job -- scheduling -- which has nothing whatever to do with weather monitoring. Indeed, it could be reused in many different kinds of applications. In fact the change is so dramatic, that we have changed the name to `AlarmClock`.

The Structure of the Sensors. Having decoupled the sensors from the rest of the system, we should look at their internal structure. Sensors now have three separate functions. First, they have to create and register the anonymous derivative of the `AlarmListener`. Second, they have to determine if their readings have changed, and invoke the `notifyObservers` method of the `Observable` class. Thirdly, they have to interact with the Nimbus hardware in order to read the appropriate values.

Figure 3-1 showed how these concerns might be separated. Figure 3-8 integrates that design with the other changes we have made. The `TemperatureSensor` base class deals with the first two concerns, since they are generic. The derivative of `TemperatureSensor` can then deal with the hardware and perform the actual readings.



```
private void check()
{
  double val = read();
  if (val != itsLastReading)
  {
    itsLastReading = val;
    setChanged();
    notifyObservers(val);
  }
}
```

```
public void wakeUp()
{
  check();
}
```

Figure 3-8
Sensor Structure

Figure 3-8 employs a pattern known as TEMPLATE METHOD[1] in order to achieve the seperation between the between the generic and specific concerns of the `TemperatureSensor`. You can see this pattern in the private `check` and `read` functions of `TemperatureSensor`. When the `AlarmClock` calls `wakeup` on the anonymous class, the anonymous class forward the call to the `check` function[1] of the `TemperatureSensor`. The `check` function then calls the abstract `read` function of `TemperatureSensor`. This function will be implemented by the derivative to properly interact with the hardware and obtain the sensor reading. The `check` function then determines whether the new reading is different from the previous reading[2]. If a difference is detected, then it notifies the waiting observers.

This nicely accomplishes the separation of concerns that we need. For every new hardware or testing platform, we will be able to create a derivative of Temperature-Sensor that will work with it. Morevoer, that derivative must simply override one very simple function: `read()`. The rest of the functionality of the sensor remains in the base class where it belongs.[3]

Where is the API? One of our phase II goals is the creation of a new API for the Nimbus 2.0 board. This API should be written in Java, be extensible, and provide simple and direct access to the Nimbus 2.0 hardware. Furthermore this API must serve the Nimbus 1.0 board as well. Without this API, all the simple debugging and calibration tools that we write for this project will have to be changed when the new board is introduced. Where is this API within our current design?

It turns out that nothing we have created so far can serve as a simple API. What we are looking for is something like this:

```
public interface TemperatureSensor
{
  public double read();
}
```

We are going to want to write tools that have direct access to this API without having to bother with registering observers. We also don't want sensors at this level to be polling themselves automatically, or interacting with the `AlarmClock`. We want something very simple and isolated that acts as the direct interface to the hardware.

1. [GOF95] p. 325
1. Remember, in Java, anonymous classes have direct access to the private variables and functions of the classes that create them.
2. Figure 3-8 uses the != operator to determine this. However, determination of floating point equality is usually more complex than this. For example, the test: a/a == 1.0 will not often succeed because of errors in the least significant bits. This is actually a significant problem for any program that makes heavy use of floating point numbers.
3. C++ programmers might make use of templates in order to further eliminate the duplication of code. Consider that the code inside BarometricPressureSensor and TemperatureSensor is nearly identical. However, the fact that the type of the sensor value is not guaranteed to be the same between the two sensors, forces them to be coded separately in Java.

It may seem that we are reversing all our previous arguments. After all, Figure 3-1 shows exactly what we have just asked for. However, the changes we made subsequent to Figure 3-1 were made for sound reasons. What we need is a hybrid that mixes the best of both schemes.

Figure 3-9 employs the BRIDGE[1] pattern to extract the true API from the `TemperatureSensor`. The intent of this pattern is to separate an implementation from an abstraction; so that both may vary independently. In our case the `TemperatureSensor` is the abstraction, and the `TemperatureSensorImp` is the implementation. Notice that the word “implementation” is being used to describe an abstract interface. And that the “implementation” is itself implemented by the `Nimbus1.0TemperatureSensor` class.

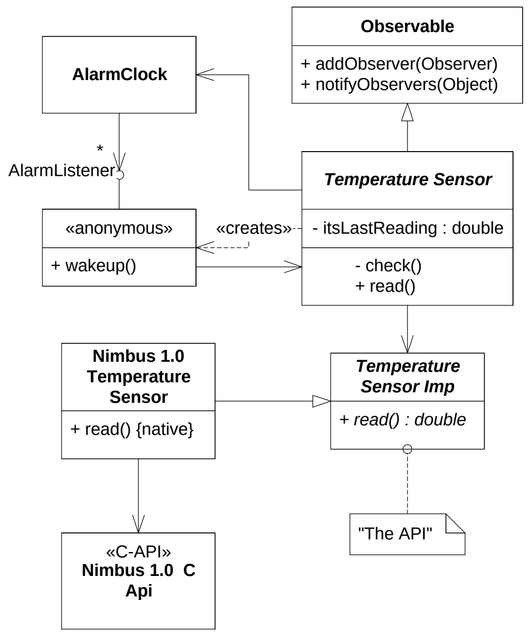


Figure 3-9
Temperature Sensor with API

1. [GOF95] p. 151

So, where are we? Perhaps it's time to sit back and examine what we have accomplished. We have taken a rather long a circuitous route to get to this point. Have we accomplished our goals?

Those goals were:

1. Create a design that would remain relatively unchagned for both versions of the hardware.
2. Create a design that will act as the foundation for many other products of this kind.
3. Lay a foundation for phase I that will migrate well to phase II.

Goals 1 and 2 were known before we began the design. Goal 3 presented itself later on, but is a regular goal of any iterative design. Are there any other goals that we missed?

Certainly we have some more functionality to add. We haven't dealt with history information, nor have we described the phase I streaming output. But as far as software engineering goals are concerned, we seem to be on track.

Again, was all this complexity worth it? Compare Figure 3-1 to Figure 3-9. The difference in complexity is striking. Yet that extra complexity is needed to ensure that the software can evolve and be maintained. It decouples the various sections of the system so that they can vary independently of each other, and can be independently reused.

But we are not done yet. We still have more functionality to add, and some more issues to resolve.

Creational issues. Look again at Figure 3-9. In order for this to work, a `TemperatureSensor` object must be created and bound to a `Nimbus1.0TemperatureSensor` object. Who takes care of this? Certainly, whatever part of the software is responsible for this will not be platform independent, since it must have explicit knowledge of the platform dependent `Nimbus1.0TemperatureSensor`.

We could use the main program to do all this. We could write it as shown in Listing 3-1.

Listing 3-1

```
public class WeatherStation
{
  public static void main(String[] args)
  {
    AlarmClock ac = new AlarmClock(
```

Listing 3-1

```
          new Nimbus1_0AlarmClock;

        TemperatureSensor ts =
          new TemperatureSensor(ac,
           new Nimbus1_0TemperatureSensor);

        BarometricPressureSensor bps =
          new BarometricPressureSensor(ac,
            new Nimbus1_0BarometricPressureSensor);

        BarometricPressureTrend bpt =
          new BarometricPressureTrend(bps)
      }
    }
```

This is a workable solution, but requires an awful lot of clerical overhead. Is there a better way? There is a design pattern that is well known for helping to deal with creatoinal issues like this. It is called ABSTRACTFACTORY[1]. The intent of this pattern is to provide an interface so that clients can create objects without knowing their concrete classes. Factories can also perform some of the clerical overhead involved with that creation. Figure 3-10 shows the structure.

We have named the factory the `StationToolkit`. This is an interface that presents methods that offer to create instances of the API classes. Each platform will have its own derivative of `StationToolkit`, and that derivative will create the appropriate derivatives of the API classes.

Now we can rewrite the main function as shown in Listing 3-2. Notice that in order to alter this main program to work with a different platform, all we have to change is the two lines that create the `Nimbus1.0AlarmClock` and the `Nimbus1.0Toolkit`. This is a dramatic improvement over Listing 3-1 which required a change for every sensor it created..

Notice that the `StationToolkit` is being passed into each sensor. This allows the sensors to create their own implementations. Listing 3-3 shows the constructor for `TemperatureSensor`.

Getting the Station Toolkit to create the AlarmClock. We can improve matters by having the StationToolkit create the appropriate derivative of the Alarm-

1. [GOF95] p. 87

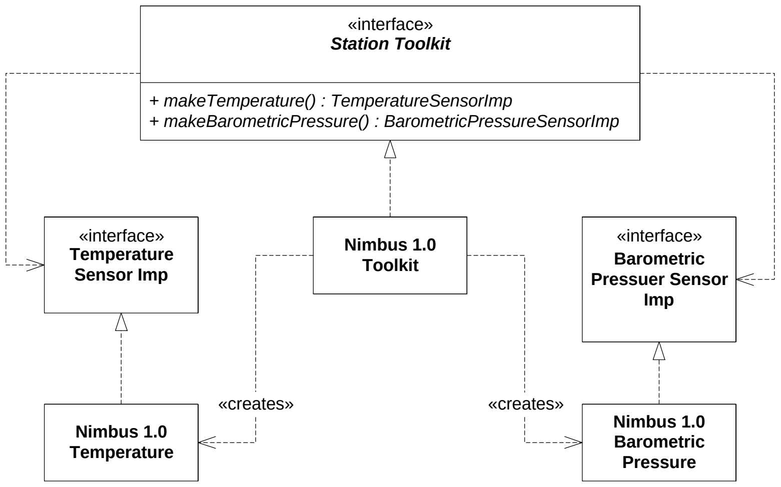

«interface»
Station Toolkit
+ makeTemperature() : TemperatureSensorImp
+ makeBarometricPressure() : BarometricPressureSensorImp
«interface»
Temperature Sensor Imp
Nimbus 1.0 Toolkit
«interface»
Barometric Pressuer Sensor Imp
«creates»
«creates»
Nimbus 1.0 Temperature
Nimbus 1.0 Barometric Pressure

Figure 3-10
Station Toolkit

Listing 3-2

```
public class WeatherStation
{
  public static void main(String[] args)
  {
    AlarmClock ac = new AlarmClock(
      new Nimbus1_0AlarmClock;

    StationToolkit st = new Nimbus1_0Toolkit();

    TemperatureSensor ts =
      new TemperatureSensor(ac,st);

    BarometricPressureSensor bps =
      new BarometricPressureSensor(ac,st);

    BarometricPressureTrend bpt =
      new BarometricPressureTrend(bps)
  }
}
```

Listing 3-3

```
public class TemperatureSensor extends Observable
{
  public TemperatureSensor(AlarmClock ac,
                           StationToolkit st)
  {
    itsImp = st.makeTemperature();
  }
  private TemperatureSensorImp itsImp;
}
```

Clock. Once again we will employ the BRIDGE pattern to separate the AlarmClock abstraction that is meaningful to the Weather Monitoring Applications, from the implementation that supports the hardware platform.

Figure 3-11 shows the new `AlarmClock` structure. The `AlarmClock` now receives `tic` messages through its `ClockListener` interface. These messages are sent from the appropriate derivative of the `AlarmClockImp` class in the API.

Figure 3-12 shows how the `AlarmClock` gets created. The appropriate `Station-Toolkit` derivative is passed into the constructor of the `AlarmClock`. The `AlarmClock` directs it to create the appropriate derivative of `AlarmClockImp`. This is passed back to the `AlarmClock`, and the `AlarmClock` registers with it so that it will receive `tic` messages from it.

Once again, this has an effect upon the main program in Listing 3-4. Notice that now there is only one line that is platform dependent. Change that line, and the entire system will use a different platform.

This is pretty good; but in Java we can do even better. Java allows us to create objects by name. The main program in Listing 3-5 does not need to be changed in order to make it work with a new platform. The name of the `StationToolkit` derivative is simply passed in as a command line argument. If the name was correctly specified, the appropriate `StationToolkit` will be created, and the rest of the system will behave appropriately,

Putting the classes into packages.

There are several portions of this software that we would like to release and distribute separately. The API and each of its instantiations are resusable without the rest of the application, and may be used by the testing and quality assurance teams. The UI and Sensors should be seperate so that they can vary indepedently. After all, newer prod-

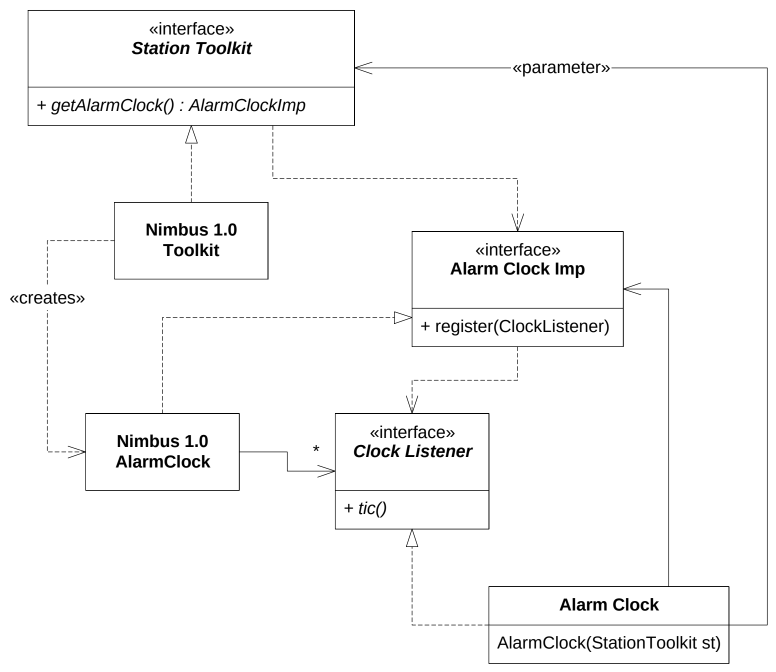


Figure 3-11
Station Toolkit and Alarm Clock

ucts may have better UI's on top of the same system architecture. In fact, Phase II will be the first example of this.

In general, the rules for seperating classes into packages involve three principles[1].

1. The Common Closure Principle (CCP) states that we want to put the classes together that are likely to change together when the requirements change. This principle tries to minimize the number of packages that will change when the requirements change. The ideal is to get this number down to 1.
2. The Reuse Release Equivalency Principle (REP) states that the granule of release is the same as the granule of reuse. That is, it is impractical to reuse anything smaller than a package. Therefore a package should contain one or more releasable units.

1. [Granularity96]

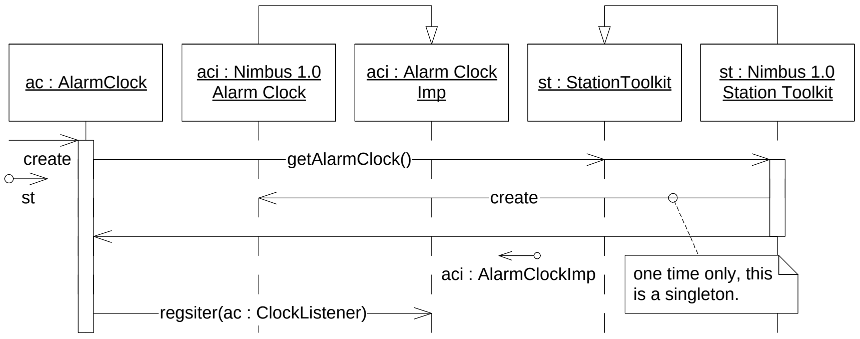

ac : AlarmClock
aci : Nimbus 1.0 Alarm Clock
aci : Alarm Clock Imp
st : StationToolkit
st : Nimbus 1.0 Station Toolkit
create
st
getAlarmClock()
create
aci : AlarmClockImp
one time only, this is a singleton.
regsiter(ac : ClockListener)

Figure 3-12
Creation of the Alarm Clock

Listing 3-4

```
public class WeatherStation
{
  public static void main(String[] args)
  {
    StationToolkit st = new Nimbus1_0Toolkit();
    AlarmClock ac = new AlarmClock(st);
    TemperatureSensor ts =
      new TemperatureSensor(ac,st);

    BarometricPressureSensor bps =
      new BarometricPressureSensor(ac,st);

    BarometricPressureTrend bpt =
      new BarometricPressureTrend(bps)
  }
}
```

Listing 3-5

```
public class WeatherStation
{
  public static void main(String[] args)
  {
    try
```

Listing 3-5 (Continued)

```
      {
        Class tkClass = Class.forName(args[0]);
        StationToolkit st =
          (StationToolkit)tkClass.newInstance();

        AlarmClock ac = new AlarmClock(st);

        TemperatureSensor ts =
          new TemperatureSensor(ac,st);

        BarometricPressureSensor bps =
          new BarometricPressureSensor(ac,st);

        BarometricPressureTrend bpt =
          new BarometricPressureTrend(bps)
      }
      catch (Exception e)
      {
      }
    }
  }
```

3. The Common Reuse Principle (CRP) states that all the classes in a package should be reused together. If it is possible for a client to reuse only a subset of classes in a package, then those classes should be removed into a seperate package.

Figure 3-13 shows a package structure for phase I. This package structure nearly falls out of the classes we have designed so far. There is one package for each platform, and the classes in `those` packages derive from the classes in the `API` package. The sole client of the API package is the `WeatherMonitoringSystem` package, which holds all the other classes.

Even though phase I has a very small UI, it is unfortunate that it is mixed in with the `WeatherMonitoringSystem` classes. It would be better to put this class in a seperate package. However, we have a problem. As things stand, the `WeatherStation` object creates the `MonitoringScreen` object, but the `MonitoringScreen` object must know about all the sensors in order to add its observers through their `Observable` interface. Thus, if we were to pull the `MonitoringScreen` out into its own package, there would be a cyclic dependency between that package and the `WeatherMonitoringSystem` package. This vio-

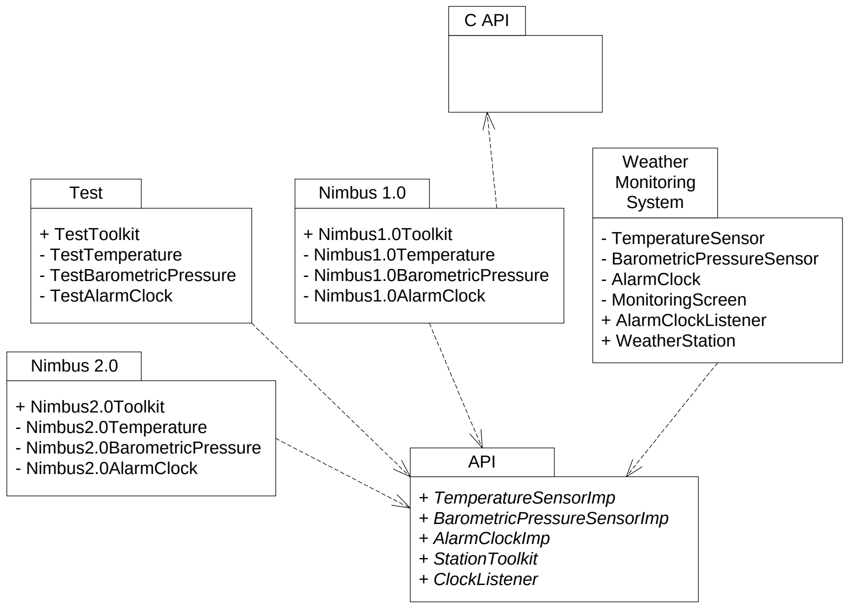


Figure 3-13
Phase I Package Structure

lates a principle known as the Acyclic Depencencies Principle (ADP)[1]. This would make the two packages impossible to release independently of each other.

We can fix this by pulling the main program out of the `WeatherStation` class. `WeatherStation` still creates the `StationToolkit` and all the sensors, but does not create the `MonitoringScreen`. The main program will create the `MonitoringScreen` and the `WeatherStation`. The main program will then pass the `WeathersStation` to the `MonitoringScreen` so that the `MonitoringScreen` can add its observers to the sensors.

How does the `MonitoringScreen` get the sensors from the `WeatherStation`? We need to add some methods to the `WeatherStation` that allow this to take place. See Listing 3-6 to see what this looks like.

Now we can redraw the package diagram as shown in Figure 3-14. We have omitted most of the packages that aren't concerned with the `MonitoringScreen`. This

1. [Granularity96]

Listing 3-6

```
public class WeatherStation
{
  public WeatherStation(String tkName)
  {
    //create station toolkit and sensors as
before.
  }

  public void addTempObserver(Observer o)
  {
    itsTS.addObserver(o);
  }

  public void addBPObserver(Observer o)
  {
    itsBPS.addObserver(o);
  }

  public void addBPTrendObserver(Observer o)
  {
    itsBPT.addObserver(o);
  }

// private variables...
  private TemperatuerSensor itsTS;
  private BarometricPressureSensor itsBPS;
  private BarometricPressureTrend itsBPT;

}
```

looks pretty good. Certainly the `UI` can be varied without affecting the `WeatherMonitoringSystem`. However, the dependency of the `UI` upon `WeatherMonitoringSystem` will cause problems whenever the `WeatherMonitoringSystem` changes.

Both `UI` and `WeatherMonitoringSystem` are concrete. When one concrete package depends upon another the Dependency Inversion Principle[1] (DIP). This principle states that dependencies should point at abstract entities. In this case, it would be

1. [DIP96]

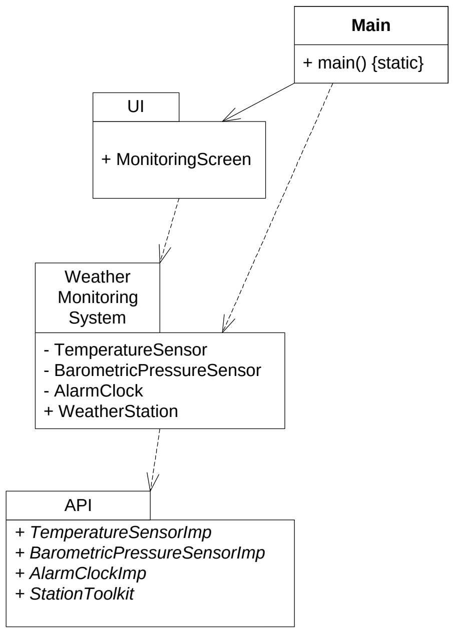


Figure 3-14
Package Diagram with Cycle Broken

better if the `UI` depended upon something abstract rather than the `Weather-MonitoringSystem`.

We can fix this by creating an interface that the `MonitoringScreen` can use, and that the `WeatherStation` derives from. See Figure 3-14.

Now, if we put the `WeatherStationComponent` interface into its own package, then we will achieve the separation we want. See Figure 3-16. Notice that now the UI and the WeatherMonitoringSystem are completely decoupled. They can both vary independently of each other. This is a good thing.

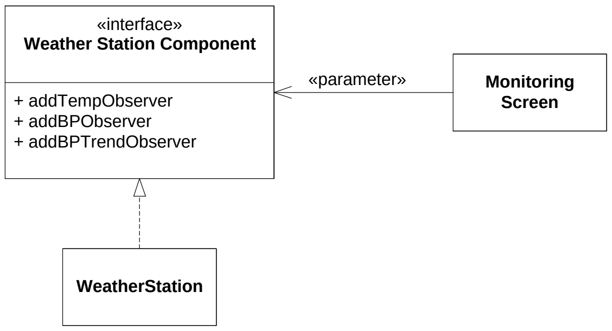


Figure 3-15
WeatherStation abstract interface

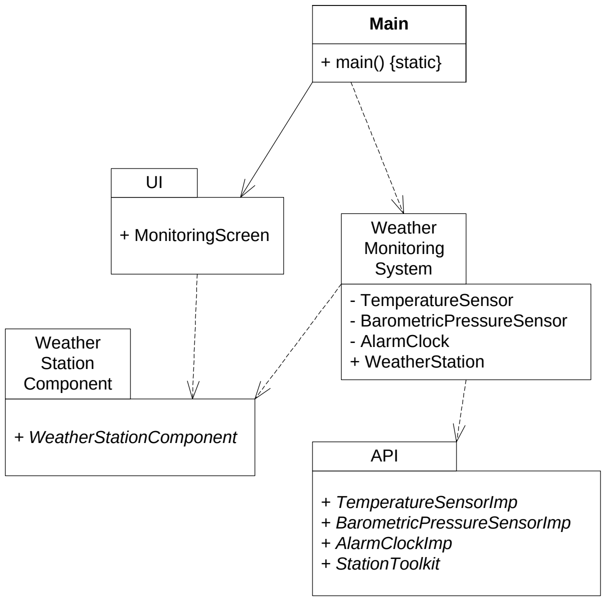


Figure 3-16
Weather Station Component Package Diagram

24 Hour History and Persistence.

Points four and five of the phase I Deliverables section (see page 110) of the Construction plan talk about the need for maintaining a persistent 24 hour history. We know that both the Nimbus 1.0 and Nimbus 2.0 hardware have some kind of non-volatile memory (NVRAM). On the other hand, the test platform will simulate the non-volatile memory by using the disk.

We need to create a persistence mechanism that is independent of the indivitual platforms, while still providing the necessary functionality. We also need to connect this to the mechanisms that maintain the 24 hour historical data.

Clearly the low level persistence mechanism should be defined as an interface in the API package. What form should this interface take? The Nimbus I C-API provides calls that allow blocks of bytes to be read and written from particular offsets within the non-volatile memory. While this is effective, it is also somewhat primitive. Is there a better way?

The Persistent API. The Java language provides the facilities that allow any object to be immediately converted into an array of bytes. This process is called *serialization*. Such an array of bytes can be reconstituted back into an object through the process of *deserialization*. It would be convenient if our low level API allowed us to specify an object, and a name for that object. Listing 3-7 shows what this might look like.

Listing 3-7

```
package api;
import java.io.Serializable;
import java.util.AbstractList;
public interface PersistentImp
{
  void store(String name, Serializable obj);
  Object retrieve(String name);
  AbstractList directory(String regExp);
};
```

The `PersistentImp` interface allows you to `store` and `retrieve` full objects by name. The only restriction is that such objects must implement the `Serializable` interface; a very minimal restriction.

24 Hour History. Having decided upon the low level mechanism for storing persistent data; lets look at the kind of data that will be persistent. Our spec says that we must keep a record of the high and low readings for the previous 24 hour period. Figure 3-24 on page 102 shows a graph with this data. This graph does not seem to make a lot of sense. The high and low readings are painfully redundant. Worse, they come from the last 24 hours on the clock, and not from the previous calendar day. Meteorologically, when we want the last 24 hour high and low reading, we want it for the previous calendar day.

Is this a flaw in the spec, or a flaw in our interpretation? It will do us no good to implement something according to the spec, if the spec is not really what the customer wants.

A quick verification with the stakeholders shows our intuition to be correct. We do indeed want to keep a rolling history of the last 24 hours. However, the historical low and high need to be for the previous calendar day.

The 24 hour high and low. The daily high and low values will be based upon real-time readings of the sensors. For example, very time the temperature changes, the 24 hour high and low temperatures will be updated appropriately. Clearly this is an observer relationship. Figure 3-17 shows the static structure, and Figure 3-18 shows the relevant dynamic scenarios.

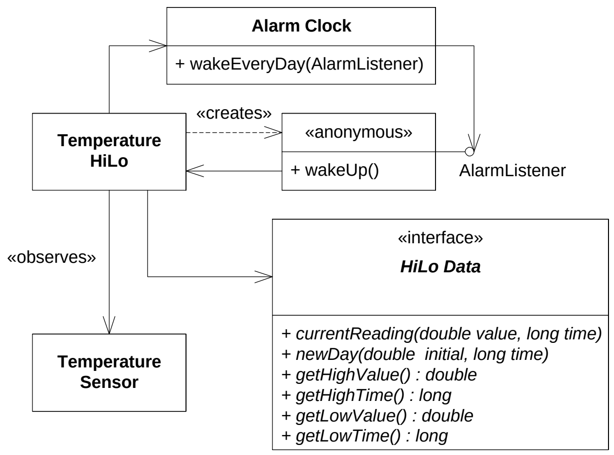


Figure 3-17
Temperature Hi Lo structure.

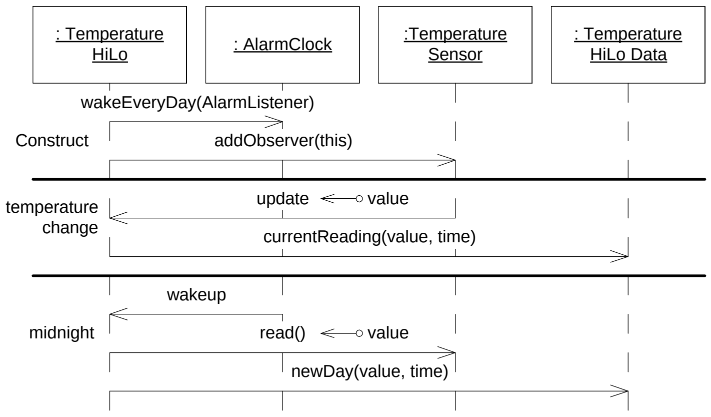


Figure 3-18
Hi Lo Scenarios.

We have chosen to show the OBSERVER pattern using an association marked with the «observes» stereotype. The details of this pattern were shown back in Figure 3-6 on page 68. We have created a class called `TemperatureHiLo` that is woken up by the `AlarmClock` every day at midnight. Notice that the `wakeEveryDay` method has been added to `AlarmClock`.

Upon construction of the `TemperatureHiLo` object it registers with both the `AlarmClock` and with the `TemperatureSensor`. Whenever the temperature changes, the `TemperatureHiLo` object is notified through the OBSERVER pattern. `TemperatureHiLo` then informs the `HiLoData` interface using the `currentReading` method. `HiLoData` will have to be implemented with some class that knows how to store the high and low values for the current 24 hour calendar day.

We have separated the `TemperatureHiLo` class from the `HiLoData` class for two reasons. First of all, we wanted to separate the knowledge of the `TemperatureSensor` and `AlarmClock` from the algorithms that determined the daily highs and lows. Secondly, and more importantly, the algorithm for determining the daily highs and lows can be reused for barometric pressure, wind speed, dew point, etc. Thus, though we will need `BarometricPressureHiLo`, `DewPointHiLo`, `WindSpeedHiLo`, etc to observe the appropriate sensors; each will be able to use the `HiLoData` class to compute and store the data.

At midnight, the `AlarmClock` sends the `wakeup` message to the `TemperatureHiLo` object. `TemperatureHiLo` responds by fetching the current temperature from the `TemperatureSensor` and forwarding it to the `HiLoData` interface. The implementation of HiLoData will have to store the previous calendar day's values using the PersistentImp interface, and will also have to create a new calendar day with the initial value.

PersistentImp accesses objects in the persistent store using a string. This string acts as an access key. Our HiLoData objects will be stored and retrieved with strings that have the following format: "<type>+HiLo+<MM><dd><yyyy>". For example: "temperatureHiLo04161998".

Implementing the HiLo algorithms.

How do we implement the HiLoData class? This seems pretty straightforward. Listing 3-8 shows what the Java code for this class looks like.

Listing 3-8
Implementation of HiLoData interface.

```
public class HiLoDataImp
      implements HiLoData
                ,java.io.Serializable
{
  public HiLoDataImp(StationToolkit st, String type,
                     Date theDate, double init,
                     long initTime)
  {
    itsPI = st.getPersistentImp();
    itsType = itsType;
    itsStorageKey = calculateStorageKey(theDate);
    try
    {
      HiLoData t =(HiLoData)itsPI.retrieve(
                              itsStorageKey);
      itsHighTime =  t.getHighTime();
      itsLowTime =   t.getLowTime();
      itsHighValue = t.getHighValue();
      itsLowValue =  t.getLowValue();
      currentReading(init, initTime);
    }
    catch (RetrieveException re)
    {
      itsHighValue = itsLowValue = init;
      itsHighTime = itsLowTime = initTime;
    }
  }
```

Listing 3-8 (Continued)
Implementation of HiLoData interface.

```
  public long   getHighTime()  {return itsHighTime;}
  public double getHighValue() {return itsHighValue;}
  public long   getLowTime()   {return itsLowTime;}
  public double getLowValue()  {return itsLowValue;}

  // Determine if a new reading changes the
  // hi and lo and return true if reading changed.
  public void currentReading(double current,
                              long time)
  {
    if (current > itsHighValue)
    {
      itsHighValue = current;
      itsHighTime = time;
      store();
    }
    else if (current < itsLowValue)
    {
      itsLowValue = current;
      itsLowTime = time;
      store();
    }
  }

  public void newDay(double initial, long time)
  {
    store();
    // now clear it out and generate a new key.
    itsLowValue = itsHighValue = intial;
    itsLowTime = itsHighTime = time;
    // now calculate a new storage key based on
    // the current date, and store the new record.
    itsStorageKey = calculateStorageKey(new Date());
    store()
  }

  private store()
  {
```

Listing 3-8 (Continued)
Implementation of HiLoData interface.

```
    try
    {
      itsPI.store(itsStorageKey, this);
    }
    catch (StoreException)
    {
      // log the error somehow.
    }
  }

  private String calculateStorageKey(Date d)
  {
    SimpleDateFormat df =
            new SimpleDateFormat(“MMddyyyy”);
    return(itsType + “HiLo” + df.format(d));
  }
  private double itsLowValue;
  private long   itsLowTime;
  private double itsHightValue;
  private long   itsHighTime;
  private String itsType;
  // we don’t want to store the following.
  transient private String itsStorageKey;
  transient private api.PersistentImp itsPI;
}
```

Well, maybe it wasn’t all *that* straightforward. Let’s walk through this code to see what it does.

At the bottom of the class you’ll see the private member variables. The first four variables are expected. They record the high and low values, and the times at which those values occurred. The `itsType` variable remembers the type of readings that this HiLoData is keeping. This variable will have the value “`Temp`” for temperature, “`BP`” for barometric pressure, “`DP`” for dew point, etc. The last two variables are declared transient. This means that they will not be stored in the persistent memory. They record the current storage key and a reference to the `PersistentImp`.

The constructor takes five arguments. The `StationToolkit` is needed to gain access to the `PersistentImp`. The `type` and `Date` arguments will be used to build the storage key used for storing and retrieving the object. Finally, the `init` and `initTime` arguments are used to initialize the object in the event that `PersistentImp` cannot find the storage key.

The constructor tries to fetch the data from `PersistentImp`. If the data is present, it copies the non transient data into its own member variables. Then it calls `currentReading` with the initial value and time to make sure that these readings get recorded. Finally, if `currentReading` discovered that there was a change in the high or low data, it will return true, and the `Store` function will be invoked to make sure that the persistent memory is updated.

The `currentReading` method is the heart of this class. It compares the old high and low value with the new incoming reading. If the new reading is higher than the old high, or lower than the old low, it replaces the appropriate value, records the appropriate time and stores the changes in persistent memory.

The `newDay` method is invoked at midnight. First it stores the current `HiLoData` in persistent memory. Then it resets the values of the `HiLoData` for the beginning of a new day. It recomputes the storage key for the new date, and then stores the new `HiLoData` in persistent memory.

The `Store` function simply uses the current storage key to write the `HiLoData` object into persistent memory through the `PersistentImp` object.

Finally, the `calculateStorageKey` method builds a storage key from the type of the `HiLoData`, and the date argument.

Ugliness. Certainly the code in Listing 3-8 is not too difficult to understand. However there is ugliness for another reason. The policy embodied in the functions `currentReading` and `newDay` have to do with managing the high-low data and are independent of persistence. On the other hand, the `store`, and `calculateStorageKey` methods, the constructor, and the `transient` variables are all specific to persistence and have nothing to do with the management of the highs and lows. It seems a shame to comingle these concepts in a single class. After all, you don't see the plumbing in your house!

In its current comingled state, this class is the makings of a maintenance nightmare. If something fundemental about the persistence mechanism changes, to the extent that the `calculateStorageKey` and `store` functions become inappropriate, then new persistence facilities will have to be grafted into the class. Functions like newDay and currentReading will have to be altered to invoke the new persistence facilities.

Decoupling persistence from policy. We can avoid these potential problems by decoupling the high-low data management policy from the persistence mechanism. using the PROXY[1] pattern.

Persistence is one of those issues that generates heat in conference rooms. Decoupling policy from persistence is certainly a desirable thing to achieve. However, it is never trivial. The conference rooms heat up because the risks of comingling are severe but deferred; while the cost of decoupling is high and immediate. It is often difficult to pay a high price for protection that you won't need for many months, and hope you won't need at all.[2]

The Nimbus 2.0 project represents a certain level of volatility. The company is trying for a new product in a new market. We can expect the requirements to remain in flux for some time to come.[3] Therefore we do not feel that it is wise to hope that the persistence mechanism will remain unaffected. We had better decouple persistence from policy.

Creating a Persistence Interface Layer. One of the most common techniques for seperating persistence from policy the division of the softare into layers that contain policy and persistence, and the interposition of a persistence interface layer between the two. However, what is often neglected about such a structure is the direction of the relationships. See Figure 3-19.

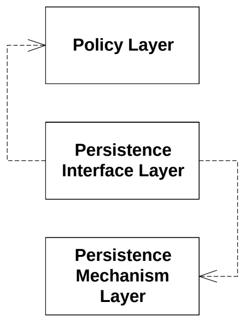


Figure 3-19

1. [GOF95] p. 207
2. It's rather like buying life insurance. You probably won't need it for years, and you hope you never need it.
3. The only projects that enjoy stable requirements are projects without customers.

The structure we desire is one in which the policy and mechanism layers have no dependence at all upon the interface layer. Rather the interface layer depends upon both. This gives us a significant amount of insulation. Changes in the persistence mechanism have no direct effect upon the policy layer. Schema changes or logic changes that do not affect the business logic do not require alterations in the policy layer. And, by the same token, changes in the policy layer have no direct effect upon the persistence mechanism either.

On the other hand, the persistence interface layer is subject to changes in both other layers. This layer is a nightmare, changing every time either of the other two layers changes. This may sound like a disadvantage, but it is not. We *want* to know where are nightmares live.

If we did not create the interface layer, then the nightmares would still exist; but they would be intertangled with our business rules and policies. The nightmares would leak out and contanimate everything.

How can we build a persistence interface layer with the appropriate dependencies? That is what the PROXY pattern is all about. See Figure 3-20.

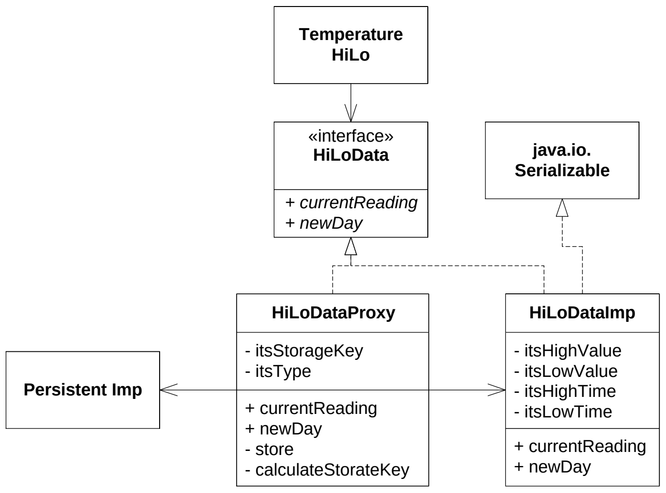


Figure 3-20
Proxy pattern applied to HiLo persistence.

Figure 3-20 differs from Figure 3-17 on page 85 by the addition of the `HiLoData-Proxy` class. It is the proxy class that the `TemperatureHiLo` object actually holds a reference to. The proxy in turn holds a reference to a `HiLoDataImp` object, and delegates calls to it. Listing 3-9 shows the implemention of the critical functions of both `HiLoDataProxy` and `HiLoDataImp`.

Listing 3-9
Snippets of the Proxy solution

```
class HiLoDataProxy implements HiLoData
{
  public boolean currentReading(double current,
                                long time)
  {
    boolean change;
    change = itsImp.currentReading(current, time);
    if (change)
      store();
    return change;
  }

  public void newDay(double initial, long time)
  {
    store();
    itsImp.newDay(initial, time);
    calculateStorageKey(new Date(time));
    store();
  }

  private HiLoDataImp itsImp;
}

class HiLoDataImp implements HiLoData
                            ,java.io.Serializable
{
  public boolean currentReading(double current,
                                long time)
  {
    boolean changed = false;
    if (current > itsHighValue)
    {
      itsHighValue = current;
      itsHighTime = time;
      changed = true;
```

Listing 3-9 (Continued)
Snippets of the Proxy solution

```
      }
      else if (current < itsLowValue)
      {
        itsLowValue = current;
        itsLowTime = time;
        changed = true;
      }
      return changed;
    }

    public void newDay(double initial, long time)
    {
      itsHighTime = itsLowTime = time;
      itsHighValue = itsLowValue = initial;
    }
  };
```

Notice how the `HiLoDataImp` class has no inkling of persistence. Notice also that the `HiLoDataProxy` class takes care of all the persistence ugliness and then delegages to the `HiLoDataImp`. This is nice. Furthermore, notice how the proxy depends upon both `HiLoDataImp` (the policy layer) and `PersistentImp` (the mechanism layer). This is exactly what we were after.

But all is not perfect. The astute reader will have caught the change that we made to the `currentReading` method. We changed it to return a `boolean`. We need this boolean in the Proxy so that the Proxy knows when to call `store`. Why don't we call store every time `currentReading` is called? There are many varieties of NVRam. Some of them have an upper limit on the number of times you can write to them. Therefore, in order to prolong the life of the NVRam, we only store into it when the values change. Real life intrudes, yet again.

Factories and Initialization. Clearly we don't want `TemperatureHiLo` to know anything about the proxy. It should know only about `HiLoData (See` Figure 3-20.) Yet somebody is going to have to create the `HiLoDataProxy` for the `TemperatureHiLo` object to use. Also, someone is going to have to create the `HiLoDataImp` that the proxy delegates to.

What we need is a way to create objects, without knowing exactly what type of object we are creating. We need a way for `TemperatureHiLo` to create a `HiLoData` without knowing that it is really creating a `HiLoDataProxy` and a `HiLoData-`

`Imp`. Mechanisms like this are commonly implemented with the ABSTRACT FACTORY[1] pattern. SeeFigure 3-21.

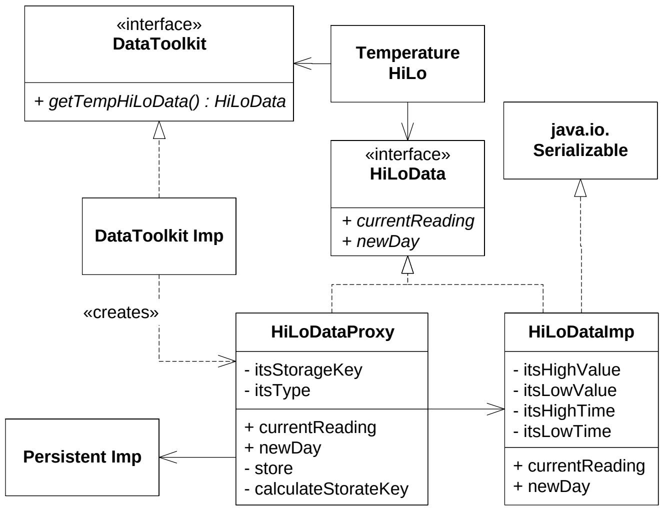


Figure 3-21
Using Abstract Factory to create the Proxy.

`TemperatureHiLo` uses the `DataToolkit` interface to create an object that conforms to the `HiLoData` interface. The `getTempHiLoData` method gets deployed to a `DataToolkitImp` object which creates a `HiLoDataProxy`, whose type code is “`Temp`”, and returns it as a `HiLoData`.

This solves the creation problem nicely. `TemperatureHiLo` does not need to depend upon the `HiLoDataProxy` in order to create it. But how does `TemperatureHiLo` gain access to the `DataToolkitImp` object. We don’t want `TemperatureHiLo` to know anything about `DataTookitImp` because that would cerate a dependency from the policy layer to the mechanism layer.

Package Structure. To answer this question, lets look at the package structure in Figure 3-22. The abbreviation WMS stands for the Weather Monitoring System package that was described in Figure 3-16 on page 83.

1. [GOF95] p. 87

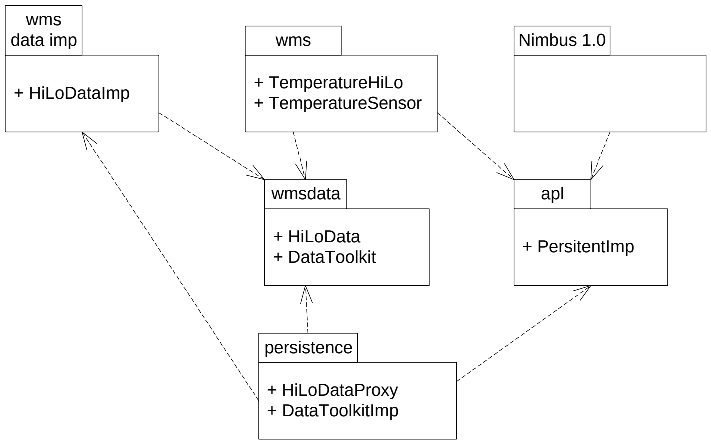


Figure 3-22
Proxy and Factory package structure

Figure 3-22 reenforces our desire for the Persistence Interface Layer to depend upon the policy and mechanism leyars. It also shows how we have deployed the classes into the packages. Notice that the abstract factory: `DataToolkit`, is defined in the `WMSData` package along with `HiLoData`. `HiLoData` is implemented in the `WMS-DataImp` package, whereas `DataToolkit` is implemented in the `persistence` package.

Who creates the factory? Now, we ask the question once again. How does the instance of `wms.TemperatureHiLo` gain access to an instance of `persis-tence.DataToolkitImp` so that it can call the `getTempHiLoData` method and create instances of `persistence.HiLoDataProxy`?

What we need is some statically allocated variable, accessible to the classes in `wms-data`, that is declared to hold a `wmsdata.DataToolkit` but which is initialized to hold a `persistence.DataToolkitImp`. Since, in Java, all variables, including static variables, must be allocated in some kind of class, we can create a class named `Scope` that will have the static variables that we need. We will put this class in the `wmsdata` package.

Listing 3-10 and Listing 3-11 show how this works. The `Scope` class in `wmsdata` declares a static member variable that holds a `DataToolkit` reference. The `Scope` class in the `persistence` package declares an `init()` function that creates a

`DataToolkitImp` instance and stores it in the `wmsdata.Scope.itsData-Toolkit` variable.

Listing 3-10

```
package wmsdata;

public class Scope
{
  public static DataToolkit itsDataToolkit;
}
```

Listing 3-11

```
package persistence;

public class Scope
{
  public static void init()
  {
    wmsdata.Scope.itsDataToolkit =
                      new DataToolkit();
  }
}
```

There is an interesting symmetry between the packages and the `scope` classes. All the classes in the `wmsdata` package, other than `Scope`, are interfaces that have abstract methods and no variables. But the `wmsdata.Scope` class has a variable and no functions. On the other hand, all the classes in the `persistence` package, other than `Scope`, are concrete classes that have variables. But `persis-tence.Scope` has a function and no variables.

Figure 3-23 shows how this might be depicted in a class diagram. The Scope classes are «utility» classes. All the members of such classes, whether variables or functions, are static. Thus, a final element to the symmetry. It would appear that packages that contain abstract interfaces tend to contain utilities that have data and no functions. Whereas packages that contains concrete classes tend to contain utilities that have functions and no data.

So, who calls `persistence.Scope.init()`? Probably the `main()` function. The class that holds that main function must be in a package that does not mind a dependency upon `persistence`. We often call the package that contains main the `root` package.

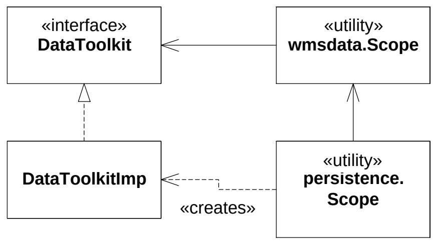


Figure 3-23

But you said... The persistence implementation layer should not depend upon the policy layer. However, a close inspection of Figure 3-22 shows a dependency from the `persistence` to `wmsDataImp`. This dependency can be traced back to Figure 3-21 in which `HiLoDataProxy` depends upon `HiLoDataImp`. The reason for this dependency is so that `HiLoDataProxy` can create the `HiLoDataImp` that it depends upon.

In most cases, the proxy will not have to create the imp because the proxy will be reading the imp from persistent store. That is, the `HiLoDataImp` will be returned to the Proxy by a call to `PersistentImp.retrieve`. However, in those rare cases where the retrieve function does not find an object in the persistent store, `HiLoDataProxy` is going to have to create an empty `HiLoDataImp`.

So, it looks like we need another factory that knows how to create `HiLoDataImp` instances, and that the proxy can call. This means more packages and more `Scope` classes, etc.

Is this really necessary? Probably not in this case. We created the factory for the proxy because we wanted `TemperatureHiLo` to be able to work with many different persistence mechanisms. Thus we had a solid benefit to justify the `DataTookit` factory. But what benefit would be obtain from interposing a factory between `HiLoDataProxy` and `HiLoDataImp`? If there could be many different implementations of HiLoDataImp, and if we wanted the proxy to work with them all, then we might be justified.

However, we don't believe that the requirements are quite that volatile. The `wmsDataImp` package contains weather monitoring policies and business rules that have remained unchanged for quite awhile. It seems unlikely that they will be changing any time in the future. This may sound like famous last words, but you have to draw the line somewhere. In this case, we have decided that the dependency from the proxy to the imp does not represent a big maintenance risk; and we will live without the factory.

Conclusion

The design so far sets the stage for keeping the rest of the historical data in persistent store. We will create other proxies and imps that will continue the separation of persitence mechanisms and policy.

The reader should take note of our use of design diagrams and code as a way to analyze the problem. We have pursued the understanding of this problem using any and every tool at our disposal, including class diagrams, sequence diagrams, and even code. This is normal and healthy.

But the time for exploratory diagrams has passed. We have a good understanding of the issues facing us, and it is now time to start developing in earnest. We will continue with phase II of the weather monitoring system in another chapter.

(The code for Phase I of the Weather Monitoring System is available at http://www.oma.com/ooadwa3/code/wm1)

Bibliography

[GOF95]: Design Patterns, Gamma, et. al., Addison Wesley, 1995

[Meyer97]: Object Oriented Software Construction, 2d. ed. Bertrand Meyer, Prentice Hall, 1997

[JAVA98]: The Java Programming Language, 2d. ed., Ken Arnold and James Gosling, Addison Wesley, 1998

[Granularity96]: Granularity, Robert C. Martin, C++ Report, Nov-Dec, 1996

[DIP96]: The Dependency Inversion Principle, Robert C. Martin, C++ Report, May, 1996

Nimbus-LC Requirements

Usage Requirements

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

- Wind speed and direction
- Temperature
- Barometric pressure
- Relative Humidity
- Wind chill
- Dew point temperature

The system shall also provide an indication of the current trend in the barometric pressure reading. The three possible values include stable, rising, and falling. For example, the current barometric pressure is 29.95 inches of mercury (IOM) and falling.

The system shall have a display which continuously indicates all measurements, as well as the current time and date.

24-Hour History

Through the use of a touch screen the user may direct the system to display the 24 hour history of any of the following measurements:

- Temperature
- Barometric Pressure
- Relative Humidity

This history shall be presented to the user in the form a line chart (see Figure 3-24

User Setup

The system shall provide the following facilities to the user to allow the station to be configured during installation.

- Setting the current time, date, and time zone.
- Setting the units that will be displayed (english or metric)

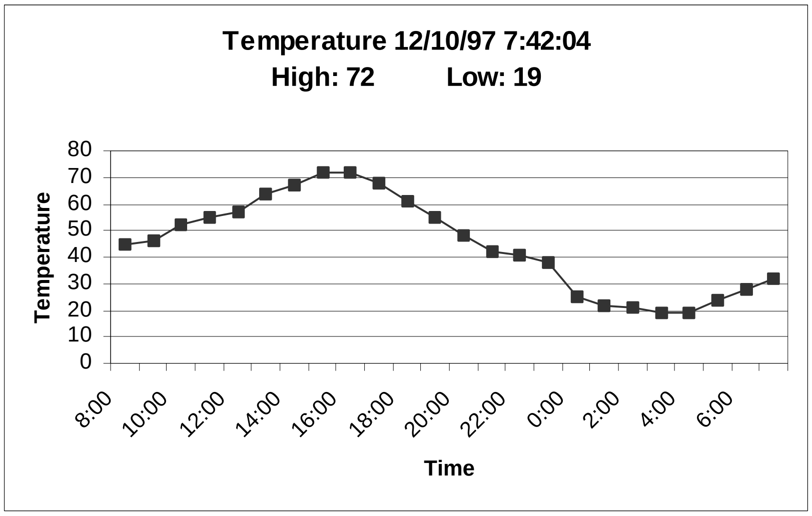


Figure 3-24
Temperature History

Administrative Requirements

The system shall provide a security mechanism for access to the administrative functions of the weather station. These functions include:

- Calibrating the sensors against known values
- Resetting the station

Nimbus-LC Elaboration Phase

Introduction

This document describes the deliverable for the elaboration phase of the Weather Monitoring Station project. This phase of the project takes attempts to transform the requirements document into a set of use cases. The deliverable for this phase of the project are as follows:

- Actors

  This phase of the project identifies the actors for the weather station.
- Use Cases (primary and secondary)

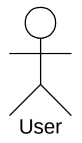
  The use cases specify the interaction that occurs between the actors and the system. It is important to note that use cases do not describe the details of how the system will accomplish the task.

Referenced Documents

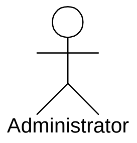
- Weather Monitoring Station - Requirements
- Weather Monitoring Station - Hardware Description

Actors

In this system there are two distinct roles played by users.



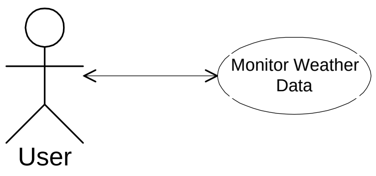
User. This actor views the real-time weather information that the station is measuring. It also interacts with the system to display the historical data associated with the individual sensors.



Administrator. The role played by this actor is one of administering the system. This administration includes controlling the security aspects of the system, calibrating the individual sensors, setting the time/date, setting units of measure, and resetting the station when required.

Use Cases

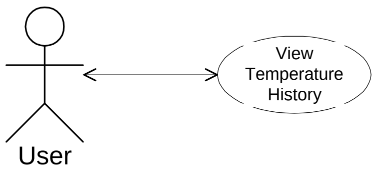
Once the actors have been defined the next step is to synthesize this information with the requirements document to describe the interaction between the newly identified actors and the system.



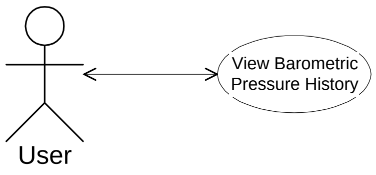
Use Case #1: Monitor Weather Data. The system will display the current temperature, barometric pressure, relative humidity, wind speed, wind direction, wind chill temperature, dew point, and barometric pressure trend.

Measurement History

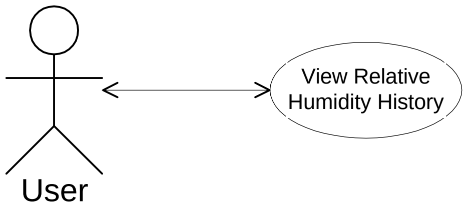
The system will display a chart depicting the previous 24 hours of readings from the sensors in the system. The type of chart is line oriented. In addition to the chart the system will display the current time and date and the highest and lowest readings from the previous 24 hours.



Use Case #2: View Temperature History. The system will display the history of the temperature readings.

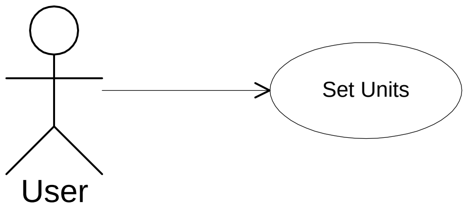


User Case #3: View Barometric Pressure History. The system will display the history of the barometric pressure readings.

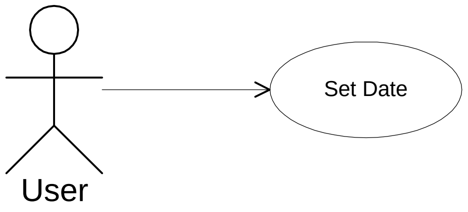


Use Case #4: View Relative Humidity History. . The system will display the history of the relative humidity readings.

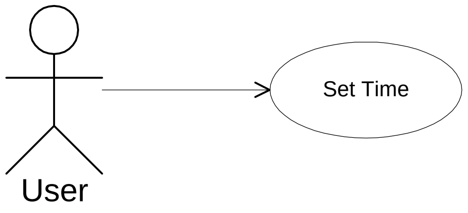
Setup



Use Case #5: Set Units. The administrator sets the type of units that will be displayed. The choices are between english and metric values. The default is metric.

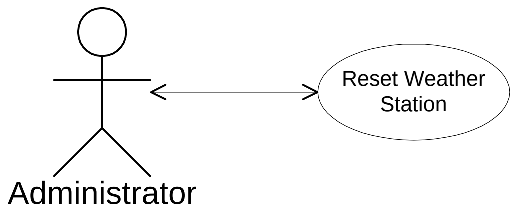


Use Case #6: Set Date. The administrator will set the current date.

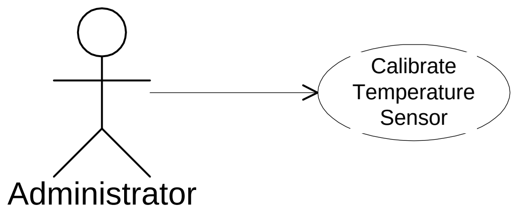


Use Case #7: Set Time. The administrator will set the current time and time zone for the system.

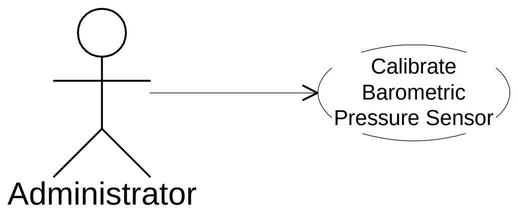
Administration



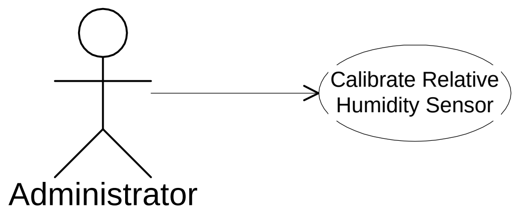
Use Case #8: Reset Weather Station. The administrator has the ability to reset the station back to it's factory default settings. It is important to note that this will erase all of the history that is stored in the station and remove any calibration that may have occurred. As one last check it will inform the administrator of the consequences and prompt for a go/no go to reset the station.



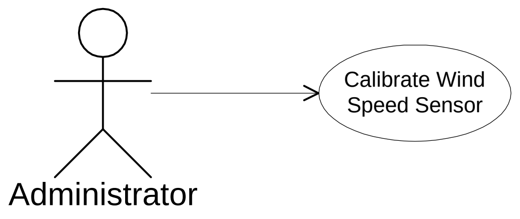
Use Case #9: Calibrate Temperature Sensor. The administrator using a known good source for the temperature will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring. For a detailed look at calibrating the sensors see the hardware description document.



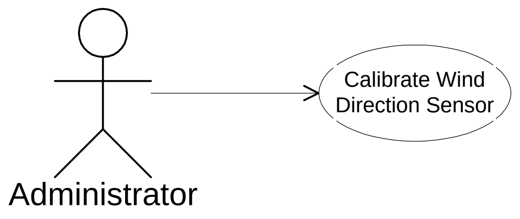
Use Case #10: Calibrate Barometric Pressure Sensor. The administrator using a known good source for the pressure will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring.



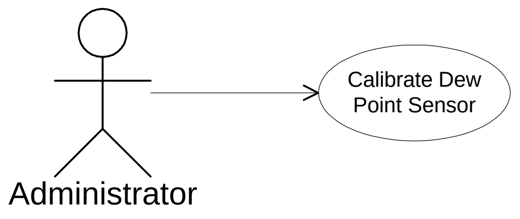
Use Case #11: Calibrate Relative Humidity Sensor. The administrator using a known good source for the humidity will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring.



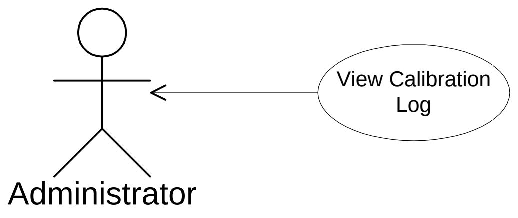
Use Case #12: Calibrate Wind Speed Sensor. The administrator using a known good source for the wind speed will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring.



Use Case #13: Calibrate Wind Direction Sensor. The administrator using a known good source for the wind direction will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring.



.Use Case #14: Calibrate Dew Point Sensor. The administrator using a known good source for the dew point will enter that value into to the system. The system shall accept the value and use it internally to calibrate that actual reading with the readings it is currently measuring



Use Case #15: Calibration Log. The system will show the administrator the calibration history of the unit. This history includes the time and date of the calibration, the sensor calibrated, and the value that was used to calibrate the sensor.

Nimbus-LC Construction Plan

Introduction

The implementation of the weather station will be done in a series of iterations. Each iteration will build on what has been done previously until we have provided the functionality which is required for release to the customer. This document outlines three construction phases for this project. Some argue there should be a strong correlation between the explicit use cases and the outputs of the construction phases. This is often at odds with I believe the main goal of the small construction phases, the reduction of risk. As is the case in most real projects there is a mixture between the two. The first phase of this project will complete no use cases in their entirety. However, it does reduce what is believed to be the main set of risks for the project.

Construction Phase I

The first phase of construction has two goals. The first is to create an architecture that will support the bulk of the application in a manner that is independent of the Nimbus hardware platform. The second goal is to manage the two biggest risks.

1. Getting the old Nimbus 1.0 API to work on the processor board with a new operating system.

   This is certainly doable, but it is very hard to estimate how long this will take because we cannot anticipate all the incompatibilities.
2. The Java Virtual Machine.

   We have never used a JVM on an embedded board before. We don't know if it will work with our operating system; or even if it correctly implements all of the Java byte codes properly. Our suppliers assure us that everything will be fine, but we still percieve a significant risk.

The integration of the JVM with the touch screen and graphics subsystem is proceeding in parallel with this construction phase. It is expected to be complete prior to the beginning of the second phase.

Risks

1. Operating System upgrade - We currently use an older version of this OS on our board. In order to use the JVM we need to upgrade to the latest version of the OS. This also requires us to use the latest version of the development tools.

2. The OS vendor is providing the latest version of the JVM on this version of the OS In order to stay current we want to use the 1.2 version of the JVM. However, V1.2 is currently in beta and will change during the construction of the project.
3. Java Native Interface to the board level “C” API needs to be verified in the new architecture.
4. Basic changes in the Java language and libraries that may occur as part of the process from beta to released version of the JVM, which should occur in the middle of the year.

Deliverable(s)

1. Our hardware running the new OS along with the latest version of the JVM.
2. A streaming output which will display the current temperature and barometric pressure readings *(throw away code not used in final release)*
3. When ther is a change in the barometric pressure the system will inform us if the pressure is risong, falling or stable.
4. Every hour the system will display the past 24 hours of measurements for the temperature and barometric pressure. This data will be persistent in that we can cycle the power on the unit and the data will be saved.
5. Every day at 12:00 AM the system will display the high and low temperature and brometric prerssure for the previous day.
6. All measurements will be in the metric system.

Construction Phase II

During this phase of the project the basis for the user interface is added to the first construction phase. No additional measurements are added. The only change to the measurements themselves is the addition of the calibration mechanism. The primary focus in this phase is on the presentation of the system. The major risk is the software interface to the LCD panel/Touch Screen. Also, since this is the first release that will display the UI in a form that can be shown to the user we may begin to have some churn in the requirements. In addition to the software, we will also be delivering a specification for the new hardware. This is the main reason for the addition of the calibration to this phase of the project. This API will be specified in Java.

Use Cases Implemented

- #2 - View Temperature History
- #3 - View Barometric Pressure History

- #5 - Set Units
- #6 - Set Date
- #7 - Set Time/Time Zone
- #9 - Calibrate Temperature Sensor
- #10 - Calibrate Barometric Pressure Sensor

Risks

1. The LCD Panel/Touch Screen interface to the Java Virtual Machine needs to be tested on the actual hardware.
2. Requirements Changes
3. Changes in the JVM. Along with changes in the Java Foundation Classes as they proceed from beta to released form.

Deliverable(s)

1. A system that executes and provides all of the functionality specified in the use cases listed above.
2. The Temperature, Barometric Pressure, and Time/Date portion of Use Case #1 will also be implemented.
3. The GUI portion of the software architecture will be completed as part of this phase.
4. The administrative portion of the software will be implemented to support the temperature and barometric pressure calibrations.
5. A specification for the new hardware API specified in Java instead of “C”.

Construction Phase III

This is the final construction phase prior to customer release of the product. In this phase we round out the implementation providing the balance of the functionality required to release the product.

Use Cases Implemented

- #1 - Monitor Weather Data
- #4 - View Relative Humidity History
- #8 - Reset Weather Station

- #11 - Calibrate Relative Humidity Sensor
- #12 - Calibrate Wind Speed Sensor
- #13 - Calibrate Wind Direction Sensor
- #14 - Calibrate Dew Point Sensor
- #15 - Calibration Log

Risks

1. Requirements Changes - It is expected as more of the product is completed there may be changes required.
2. Completion of the entire product may indicate changes to the Hardware API that was specified at the end of Construction Phase II.
3. Limits of Hardware - As we complete the product we may run into limitations of the hardware (i.e. memory, CPU, etc.)

Deliverable(s)

1. The new software running on the old hardware platform.
2. A specification for the new hardware that has been validated with this implementation.