A Device Polling Problem

Objective Engineering, Inc.

699 Windsong Trail
Austin, Texas 78746
512-328-9658
FAX: 512-328-9661
ooinfo@oeng.com
http://www.oeng.com

© Objective Engineering, Inc., 1999-2007.

Photocopying, electronic distribution, or foreign-language translation of this document is permitted for personal and classroom use, provided this document is reproduced in its entirety and accompanied by this notice and by its copyright. Copies and translations may not be used or distributed for profit or commercial advantage without prior written approval from Objective Engineering, Inc.

Part I: The Problem

You must develop software that allows clients to periodically check for changes in the status of devices in a network. Specifically, a client must be able to create a monitor that will periodically check a particular device in the network. If the state of that device has changed since the monitor last checked the device, the monitor should write an event to a global event log.

A device is uniquely designated by its device id. When making its initial monitoring request, the client specifies the id of the device to be monitored and the monitoring period. At that point, the monitor is initialized but has not yet been started. The client makes a subsequent request to start the monitor. The client should be able to start and stop the monitor at any time.

Each device has an interface to poll its current state, although the precise interface differs from device to device. As an example, to poll a D1 device, you invoke its `poll` method, whereas to poll a D2 device, you call its `currentState` method. The various Device classes are provided by different vendors, so you are not permitted to change the interfaces of those classes.

The components that make up the states of different devices may also differ. For example, the state of a D1 device is defined by three floating point values. A D2 device's state, on the other hand, is represented by five integer values and a floating point value.

Many clients may have monitors running, and different clients may be monitoring the same device at different intervals. For example, one client may be monitoring a device every five minutes while another client monitors that same device every seven minutes. Any solution must therefore permit multiple monitors on the same device. (You need not worry about "locking" devices to guarantee that the polling operation is atomic, however.)

Assume the existence of an Event Log and an Event class. The Event Log class defines a method, `logEvent`, that takes an Event as an argument and places that Event in the Log. You should write a specific type of Event, a Device Change Event, that includes the id of the device.

Part II: A Solution

This problem calls for a mechanism that permits the periodic polling of devices. A complication is that the various devices have different interfaces as well as different internal components that define their states. The solution presented here includes both a model of the requirements and a design.

The requirements model consists of a use case diagram and a textual description of each use case. While the former adheres to UML notation, UML provides no syntax for documenting the details of use cases. This document employs simple English paragraphs for use case descriptions.

The design described here includes a static model in terms of a class diagram, as well as interaction diagrams that form a partial dynamic model.

A Requirements Model

You can create a simple use case diagram that models the functional requirements of the device monitoring facility. A client program uses this facility to monitor devices. Specifically, such a program can make four types of requests:

a) It can define a monitor.

b) It can start a monitor.

c) It can stop a monitor.

d) It can discard a monitor.

A client program is one type of actor. Because the actual devices and the event log are provided to you, you can also model those entities as actors. Creating and stopping a monitor does not require any interaction with the device. Starting a monitor, on the other hand, requires an initial and periodic polling of the device. Because the polling of a device happens in the background (from the client's point of view), you could treat that function as a separate use case. Figures 1a and 1b contain the resulting use case diagram using UML 1.1/1.2 and UML 1.3 notation, respectively.

The Stop Monitor use case extends Discard Monitor because, if a client program requests that a running monitor be discarded, that monitor must first be stopped. The textual descriptions of these use cases are:

Define Monitor. A client program asks that a monitor be defined, specifying a device identifier indicating a specific device, as well as a monitoring period. A monitor is defined. An exceptional condition occurs when the client specifies a device identifier that is unknown to the system.

© 1999-2007 by Objective Engineering, Inc.

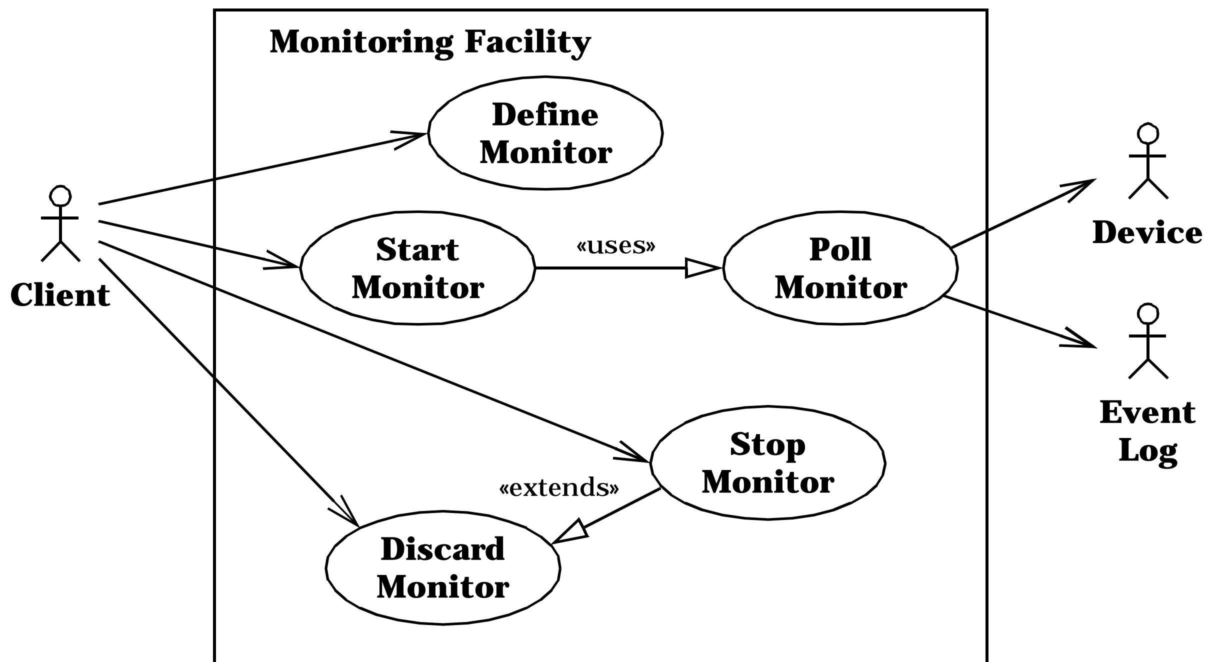

Figure 1a: A UML 1.1/1.2 use case diagram for device monitoring.

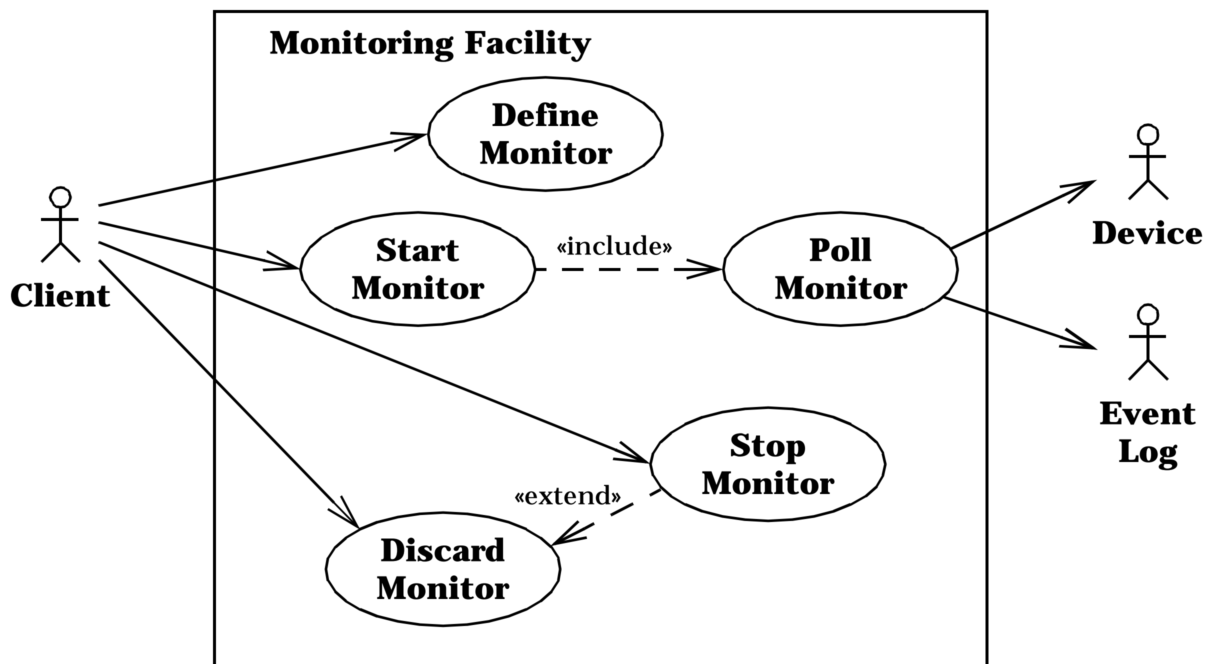

Figure 1b: A UML 1.3 use case diagram for device monitoring.

© 1999-2007 by Objective Engineering, Inc.

Start Monitor. A client program asks that a monitor be started, specifying a particular monitor. The Poll Device use case is employed to periodically poll the device. (Control must return to the client immediately, and the polling must occur in the background.) An exceptional condition arises when the client tries to start a monitor that already has been started.

Stop Monitor. A client program asks that a monitor be stopped, specifying a particular monitor. The monitor is de-activated and its polling ceases. An exceptional condition occurs when the client tries to stop a monitor that already has been stopped.

Discard Monitor. A client program asks that a monitor be discarded. The client is indicating that the monitor will not be used again, and so all vestiges of the monitor should be removed from the system. If the monitor is running when the request is issued, the monitor should be stopped before it is discarded. It is an error to attempt to start or stop a monitor that has been discarded.

Poll Device. The device is initially polled, after which the following steps are executed repeatedly. A timer is started. When the timer expires, the device is polled, and its current state is compared to its previous state. If the two states differ, a *device change event* is written to the event log.

The Poll Device use case includes both the triggering condition for polling the device and the behavior for polling the device. To allow the reuse of the polling behavior in situations that do not employ time-based monitoring, the triggering and polling behaviors could be factored into distinct use cases.

The Poll Device use case requires a repeating sequence of actions. To add clarity to this requirements model, you could draw an activity diagram for that use case. (Such an activity diagram is not included here.)

A First Design

First, consider what interface is to be provided to client programs. One possible interface is a monitoring *facade* [GHJ&V, pp.185-193] that offers an API-level interface for all of the capabilities required by a client. In particular, the facade is a class that includes methods to define, start, stop, and discard a monitor. When a client program creates a monitor, the facade must return an identifier that the client can use to subsequently manipulate the monitor. A possible facade class is shown in Figure 2.

A minor disadvantage of this approach is the need to create and maintain an identifier for each monitor. A second, more significant disadvantage stems from the fact that a client is permitted to restart a monitor. The facade must include a method by which the client can dispose of a monitor, as a monitor cannot be discarded by the monitoring facility simply because it has been stopped. This

© 1999-2007 by Objective Engineering, Inc.

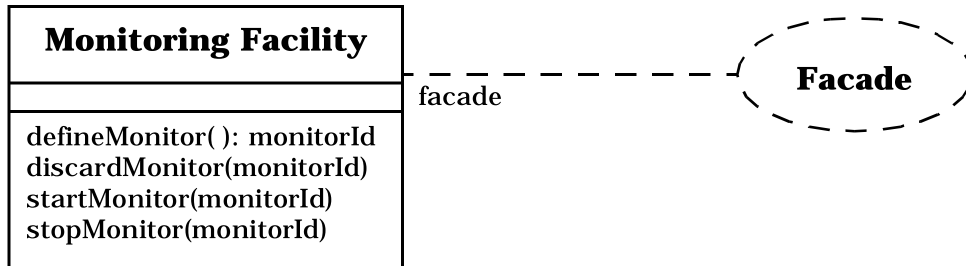

Figure 2: A monitoring facade class.

implies that the client must remember to inform the facade to dispose of its monitors. (Failure to do so means that a monitor will hang around consuming memory indefinitely.)

Because it must maintain the current state of each monitor, the Monitoring Facility in Figure 2 must employ some sort of Monitor object behind scenes. As an alternative interface, why not allow a client to create and interact with that instance directly? To define a monitor, a client creates an instance of a Monitor class, providing the device identifier and the monitoring period. To start (or resume), stop (or pause), or discard a Monitor, a client invokes the Monitor's `start`, `stop`, or `discard` method, respectively. Figure 3 depicts the Monitor class.

By creating (and starting) multiple Monitor instances, a client may have multiple Monitors running at once. In fact, a client can have multiple Monitors monitoring the same device (perhaps at different periods). The sequence diagram in Figure 4 illustrates a typical interaction between a client and a Monitor object.

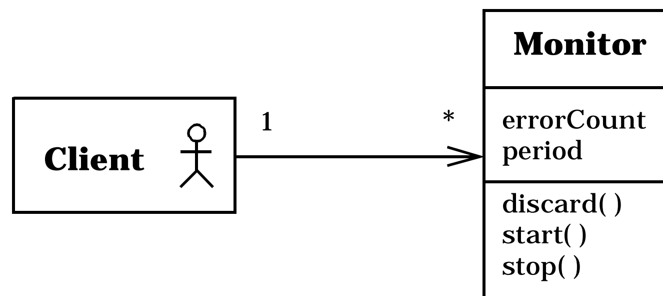

Figure 3: The Monitor class.

© 1999-2007 by Objective Engineering, Inc.

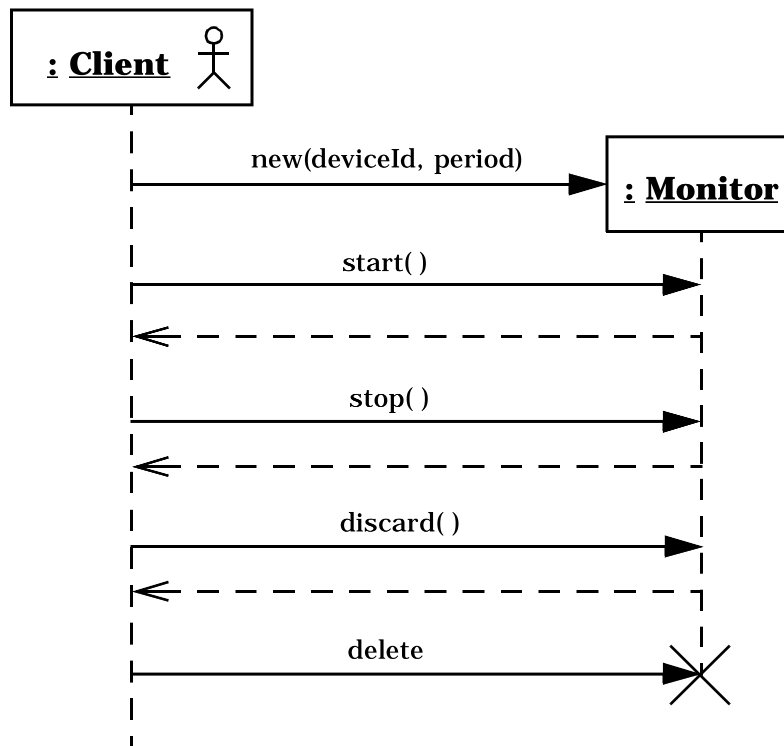

Figure 4: A client's interactions with a Monitor instance.

Figure 4 assumes an implementation in a language that supports the explicit deletion of objects. (For an implementation language void of that feature, the figure would not include the invocation of `delete`.) The developer must take care to guarantee the correct semantics when a client deletes a Monitor that has not been previously discarded. (In C++, for example, the Monitor destructor should discard the Monitor if necessary.)

When created, a Monitor must be able to find the specific device to be monitored. The design must therefore maintain a list of devices, each of which is identified by a device identifier. For the moment, assume the existence of an abstract Device class, a generalization of the specific device classes (such as D1 and D2). A Device Registry can be used to look up each Device instance using a device identifier as a qualifier (i.e., a key), as shown in Figure 5. Figure 5 also illustrates the association from the Monitor class to the Device Registry. The `static` constraint indicates that the Monitor class will hold the Device Registry reference in a class variable (e.g., a static field in Java, a static data member in C++, or a class variable in Smalltalk).

The Monitor class will also employ a class variable to hold the reference to the global Event Log. Its relationship with the Event Log, as well as the definition of the Event classes, are included in Figure 6.

© 1999-2007 by Objective Engineering, Inc.

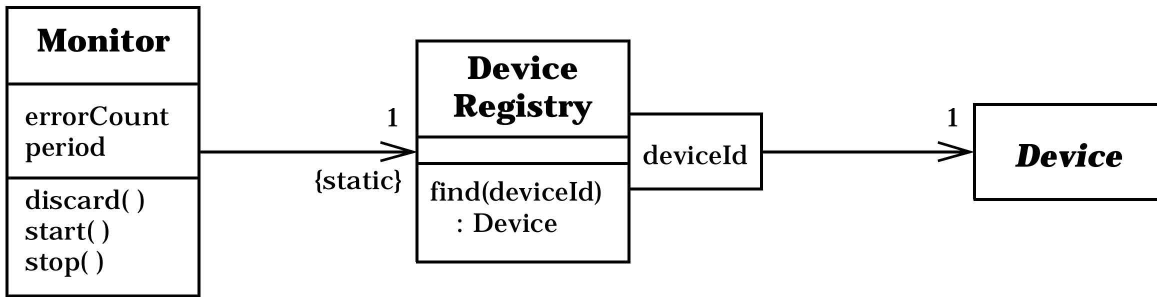

Figure 5: A Device Registry class.

Figure 7 contains the composite design to this point. Two problems remain to be solved:

1. Different devices have different polling interfaces, so a Monitor must somehow accommodate these variations.
2. The components that make up the states of different devices vary. A Monitor must also accommodate these variations.

To solve the problem of varying device interfaces, you can apply the Adapter pattern [GHJ&V, pp. 139-150]. An *object adapter* is placed atop each device instance. That adapter defines a uniform polling interface, a `poll` method that takes no arguments and returns the state of the device. The `poll` method polls

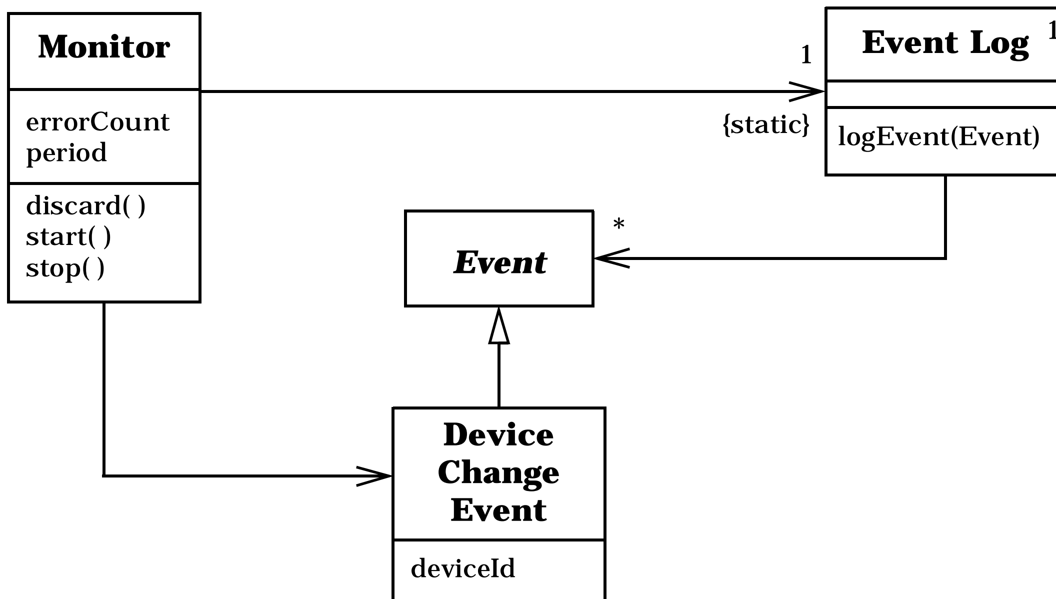

Figure 6: A Monitor's relationship with the Event Log and Events.

© 1999-2007 by Objective Engineering, Inc.

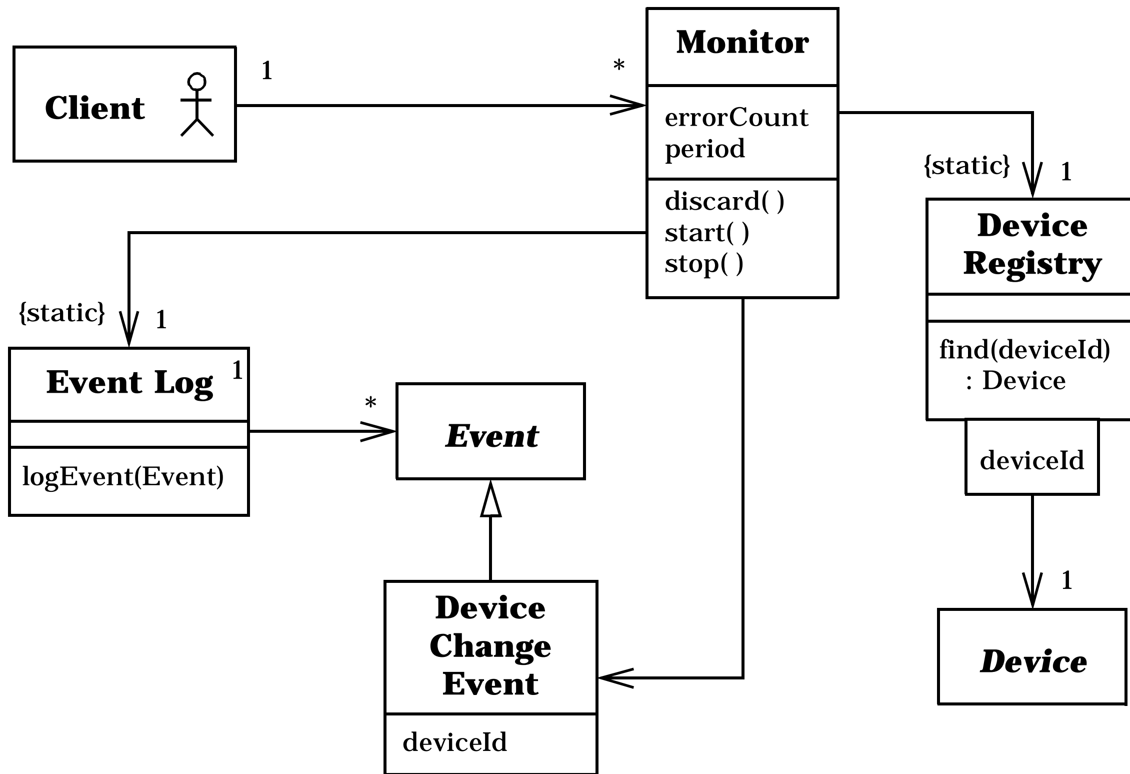

Figure 7: The design to this point.

its underlying device, using the device's specific interface, and returns the state of the device. All adapter classes are derived from a common Device Adapter interface class that defines the common polling interface. The adapter classes for devices D1 and D2 are included in Figure 8.

Observe that the figure includes an explicit indication of the application of the Adapter pattern. The pattern is depicted as a collaboration, the notation for which is a dashed oval labeled with the collaboration (pattern) name. Dependencies relate the pattern to its participating classes, and each class is labeled with its role in the pattern. (The D1 class plays the role of an adaptee, for example.) Although omitted from the figure due to space limitations, the D2 Adapter and D2 classes also act as adapter and adaptee, respectively.

The Device Adapter class in Figure 8 is labeled with the `«interface»` stereotype. Recall that such an interface class contains only public, abstract methods. The class name (Device Adapter) and its method name (`poll`) are not italicized in the figure because, by virtue of being an interface class, the class and its methods must be abstract (and so the italics are superfluous).

© 1999-2007 by Objective Engineering, Inc.

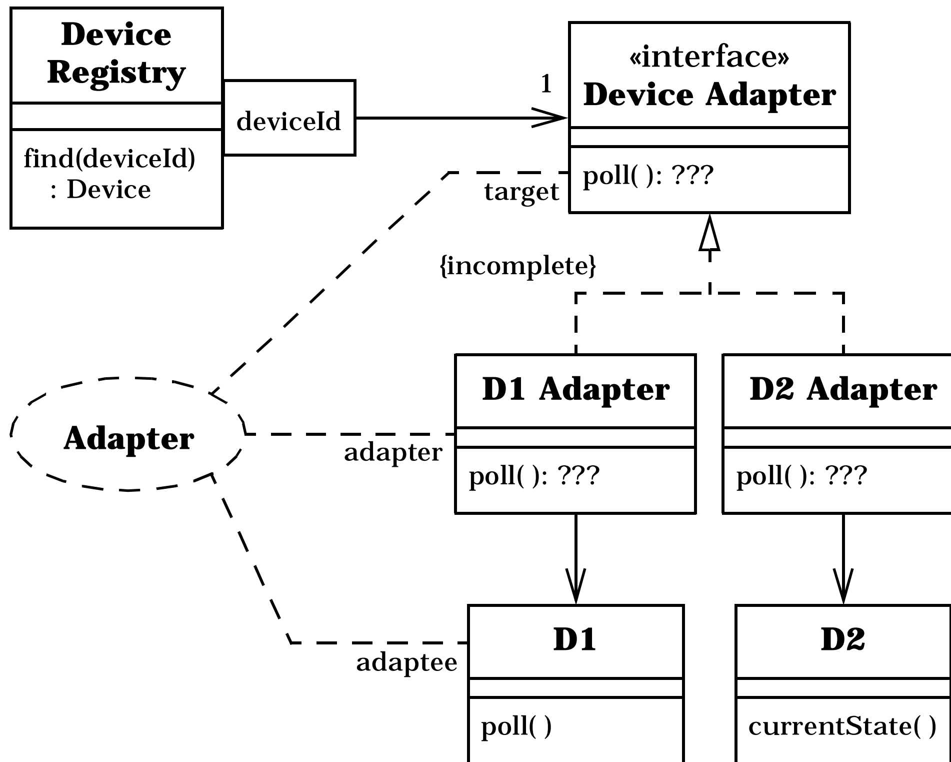

Figure 8: The adapter classes.

Figure 8 also refines the definition of the Device Registry used by a Monitor to locate a device. When asked to locate a device, the Registry will return a reference to the appropriate Adapter instance for that device.

Although such a class is not shown in Figure 8, the device classes may specialize a Device class that defines properties common to all devices, such as a device identifier.

The specification of the `poll` method defined in the adapter classes in Figure 8 does not include a return type. What should the return type be? Recall that the states of different devices have different constituent parts. To handle the varying parts of different device states, and to divorce the Monitor class from knowledge about how to compare the current and previous states of devices, you can apply a variation of the Memento pattern [GHJ&V, pp. 283-291].

In general, a *memento* encapsulates the state of an *originator*. In this design, Poll Token classes define mementos for the various types of devices (the originators). In other words, each type of device has a corresponding Poll Token class with instance variables that define the state of the device. Because the state of a D1 device is defined by three floating point values, for example, a D1

© 1999-2007 by Objective Engineering, Inc.

Poll Token contains instance variables for those three values. Conversely, a D2 Poll Token holds the five integer values and one floating point value that define the state of a D2 device.

When polled, each Adapter instance creates the appropriate Poll Token instance for that device. A Device Poll Token interface class defines the interface that all Poll Tokens have. In particular, it specifies an (abstract) `equals` method that takes another Device Poll Token as an argument and returns a `boolean`. That method should return `true` if its Token is of the same type as the passed Device Poll Token (e.g., if they are both *D1* Poll Tokens), *and* if its state equals the state of the passed Device Poll Token. It should return `false` if either condition does not hold.

> **Note:** If you implement this design in Java, the `equals` method defined in the Device Poll Token interface is identical to the standard `equals` method in the Object class, except that the standard one takes an Object reference, not a Device Poll Token, as an argument. *Overloading* the `equals` method is a bad programming practice, in that *which* `equals` implementation is invoked is determined based on the compile-time type of the argument. For example, if you calling a D1 Poll Token's `equals` method, passing it a D1 Poll Token that has been cast to an Object reference, the `equals` method in Object will be called.
>
> For a Java implementation, the `equals` method in Device Poll Token should *override* Object's `equals` method. This implies that it takes as a parameter a reference to an Object. In such a case, the Device Poll Token interface need not define the `equals` method at all. (That is, the interface serves simply as an abstract type, the return type of an adapter's `poll` method, but it defines no methods itself.) You can invoke `equals` on any reference in Java, even if the type of the reference is an interface that does not define `equals` explicitly.
>
> The equality test might be implemented in C++ by overriding the == operator, or in Smalltalk through the use of the #= operator.

Figure 9 describes the design of the Poll Token classes. The figure also illustrates (using the `«creates»` dependencies) that each specific Adapter instance creates a corresponding Poll Token instance.

> **Note:** This figure and others in this solution assume that an Adapter returns a Java-style or Smalltalk-style superclass reference. For C++, you would return a base class pointer.)

Figure 10 contains an elaboration of the basic design of the Monitor as shown in Figure 7. The Device Adapter interface class in Figure 10 replaces the Device class in Figure 7. A Monitor calls the Device Registry's `find` method, passing a device identifier, and the Registry returns the Device Adapter for the device with

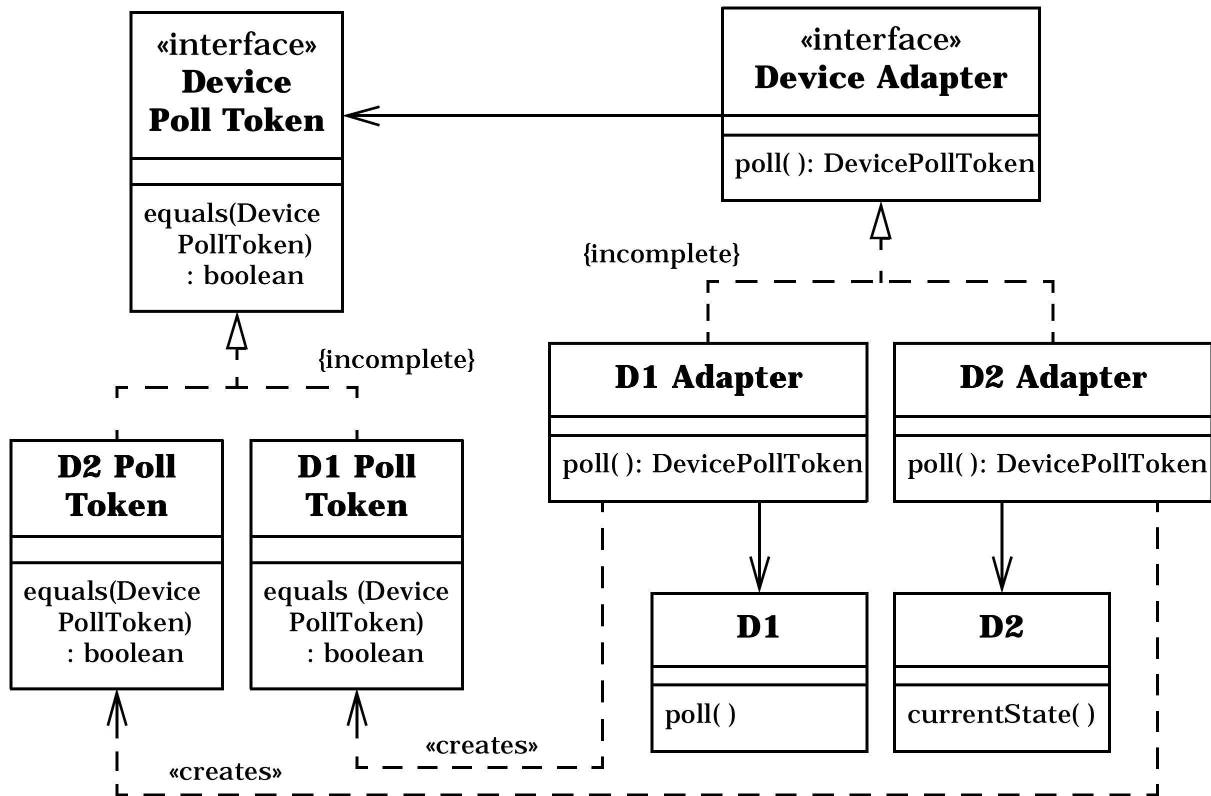

Figure 9: The poll token classes.

that device identifier. The Device Change Event class in Figure 10 does not define a `deviceId` attribute, as shown in Figure 7, but rather holds a reference to a Device Poll Token that contains the entire state of the device.

The Monitor in Figure 10 holds references to as many as two Device Poll Tokens. One is the Token returned on the last call to its Device Adapter's `poll` method. When its timer expires and it calls `poll` again, it temporarily holds a second, current Token. It then asks one Token if it is equal to another, and, in the case where the Tokens are not equal, it creates and logs a Device Change Event.

A typical polling scenario is:

1. The Client starts a Monitor that is monitoring a D2 device.
2. The Monitor polls the D2 Adapter. The Adapter in turn polls its underlying D2 device (by calling its `currentState` method), then creates a Device Poll Token that holds the state of the D2 device. The Adapter returns that Token to the caller (the Monitor).
3. The Monitor starts a timer for *period* minutes.

© 1999-2007 by Objective Engineering, Inc.

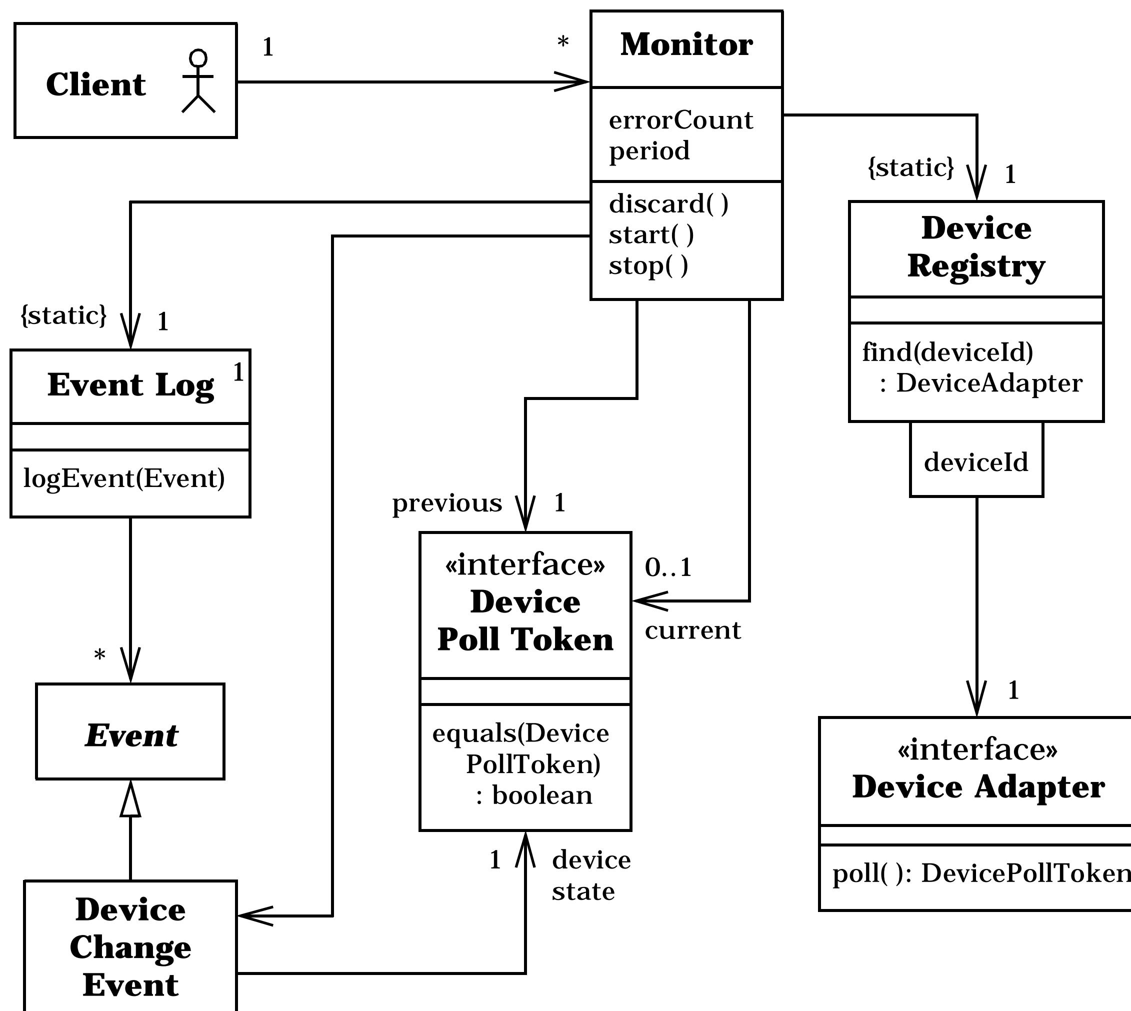

Figure 10: An elaborated design of the Monitor.

4. The timer expires, at which point the Monitor polls the D2 Adapter again. The Adapter polls its D2 device, then creates and returns a D2 Poll Token that contains the current state of the D2 device.

5. The Monitor asks the current D2 Poll Token if its equals the previous D2 Poll Token. The current Token compares the two and returns `true` (indicating the two are equal).

6. The Monitor throws away the previous Token, establishes the new token as the previous one, and starts a timer for `period` minutes. (When the timer expires, step 4 will repeat.)

© 1999-2007 by Objective Engineering, Inc.

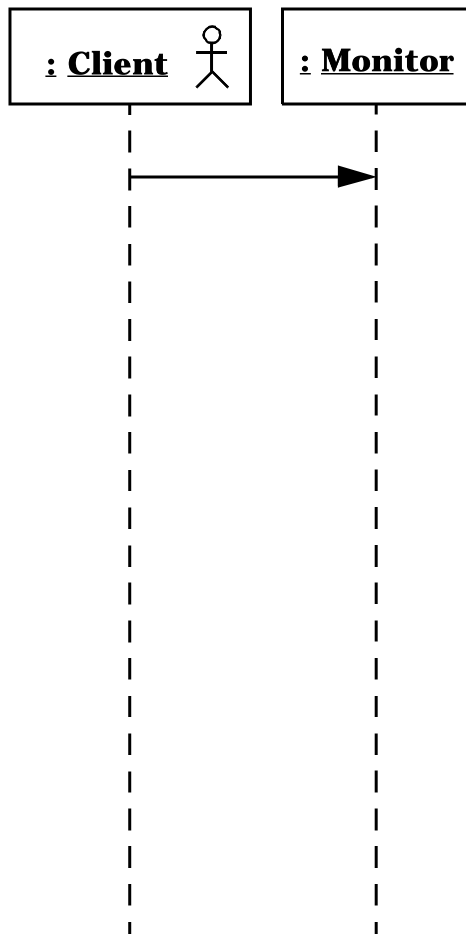

Figure 11 holds a sequence diagram for this scenario. In a C++ implementation of this design, the Monitor will delete the initial D2 Poll Token at the point in the diagram at which it establishes `t2` as the previous token. In Java and Smalltalk, the Monitor will simply throw away its reference to the Token. The figure does not include Token destruction and therefore reflects this latter view.

Figure 12 contains an abbreviated sequence diagram for the case where the most recent Token differs from the previous one. When this occurs, a Device Change Event must be created and written to the Event Log.

Observe that Figures 11 and 12 omit the concurrency aspects of this design. When a Monitor is started, that Monitor must create a separate thread that handles the polling. The timer itself might run as a separate thread, or the polling thread might simply put itself to sleep for the required period.

© 1999-2007 by Objective Engineering, Inc.

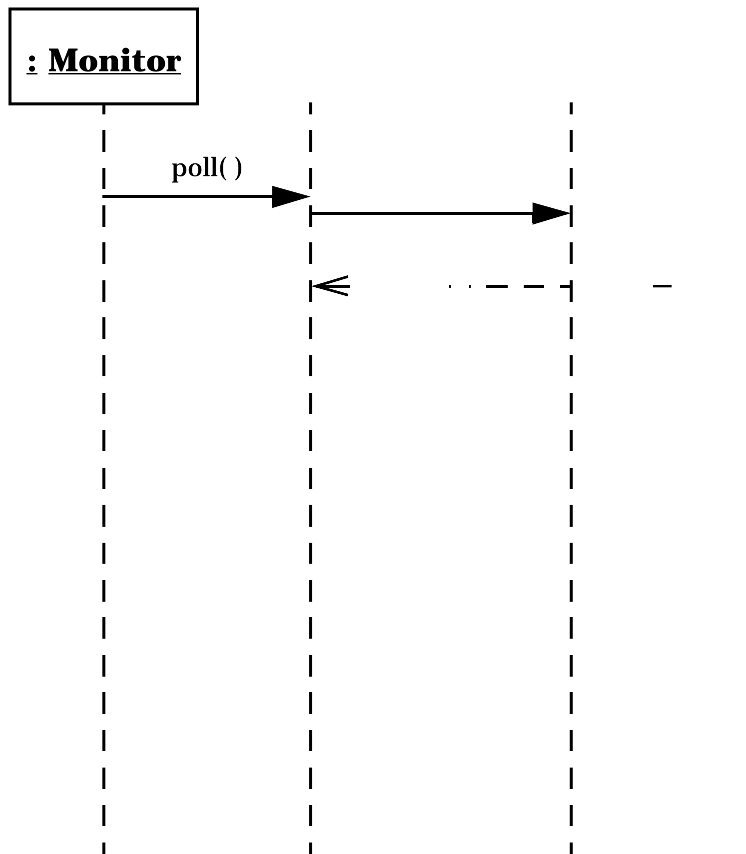

Figure 13 depicts the design of a Polling Thread class. It inherits from a Thread class (such as the Thread class in Java). The `run` method in that class defines what a Thread instance does when its thread of control is initiated. The Polling Thread class overrides that method to include the periodic polling of the device. When started, a Monitor will create a Polling Thread instance and invoke its `start` method. That method will initialize the thread and call the `run` method. (The sequence diagrams in Figures 11 and 12 could be extended to show the creation and initiation of a Polling Thread instance.)

A Monitoring Framework

Consider the class diagram in Figure 10 again. It depicts the design of a polling mechanism for devices. With a small degree of generalization, this design can be transformed into a framework for monitoring anything that can be polled.

To allow the monitoring of non-devices, the Device Adapter and Device Poll Token interfaces become Pollable and Poll Token interfaces, respectively. The Device Poll Token's `equals` method is revised to take a generic Poll Token as an

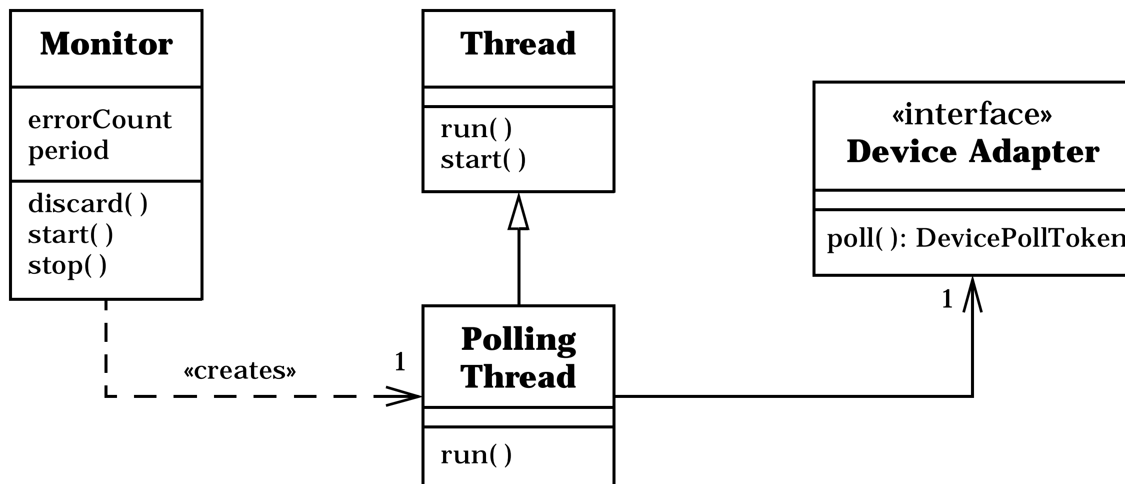

Figure 13: The introduction of a Polling Thread class.

argument. (In a Java implementation, where `equals` takes an Object reference as an argument, only the *name* of the Device Poll Token interface class changes.) In the Device Adapter interface, however, the `poll` method must be altered to return a generic Poll Token (rather than a Device Poll Token). The Device Registry becomes a Pollable Registry that, given a Pollable identifier, returns a Pollable reference. Figure 14 contains these generalizations.

Figure 14 also illustrates the generalization of the mechanism used to create Event instances. The framework defines an Event Factory interface class that must be specialized when specializing the framework. This interface class defines a single method, `createChangeEvent`, that takes a Poll Token as an argument and returns an Event. The Monitor uses this Factory to create the required Event instance when the current and previous Poll Tokens differ. (The Monitor then logs the resulting Event.)

An Alternative Design

Consider a situation in which you have many clients monitoring each device, but most devices change their states infrequently. In such circumstances, the design just presented requires many interactions with devices for relatively few changes of state. Suppose a device requires significant resources to respond to polling requests. For example, the device might expend many machine cycles determining its current state, or perhaps the device must synchronize itself, ignoring all other requests while it handles the polling request. In such a situation, the monitors in the previous design may overtax a device by consuming too many device resources.

© 1999-2007 by Objective Engineering, Inc.

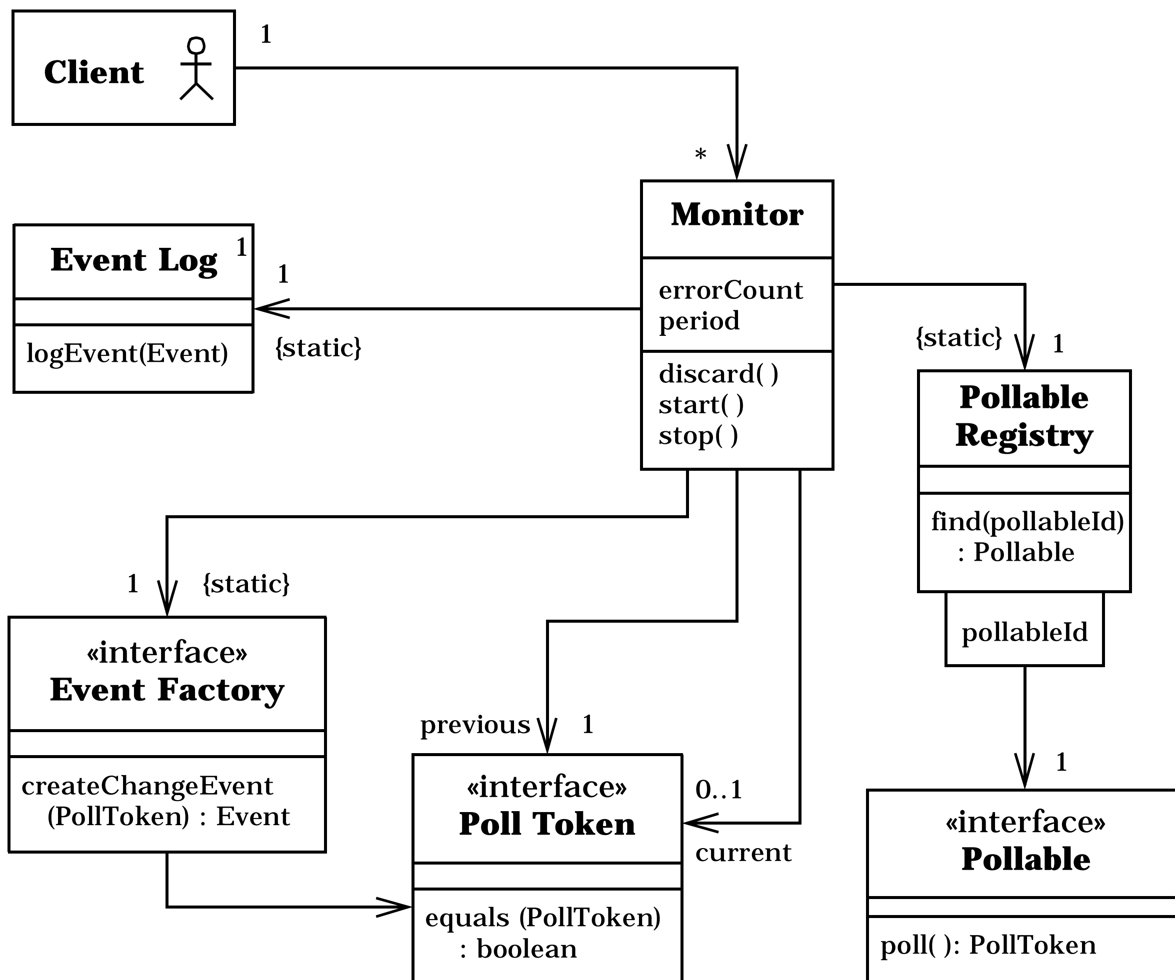

Figure 14: A monitoring black-box framework.

The problem specification states that you are not permitted to alter the definitions of the device classes. If each such class provides an interface to register “listeners” or “observers” to watch for state changes in the device, however, you have a second design option available to you. To reduce the overhead incurred by the devices, you could employ a design in which each device “pushes” it changes to observing parties.

What were device adapters in the previous design become device observers (or listeners) in this design. Each observer registers itself with its particular device as an observer of that device. Each time a device’s state changes, the device informs all registered observers of that change. When informed of a change of state, the observer records the new state of the device.

© 1999-2007 by Objective Engineering, Inc.

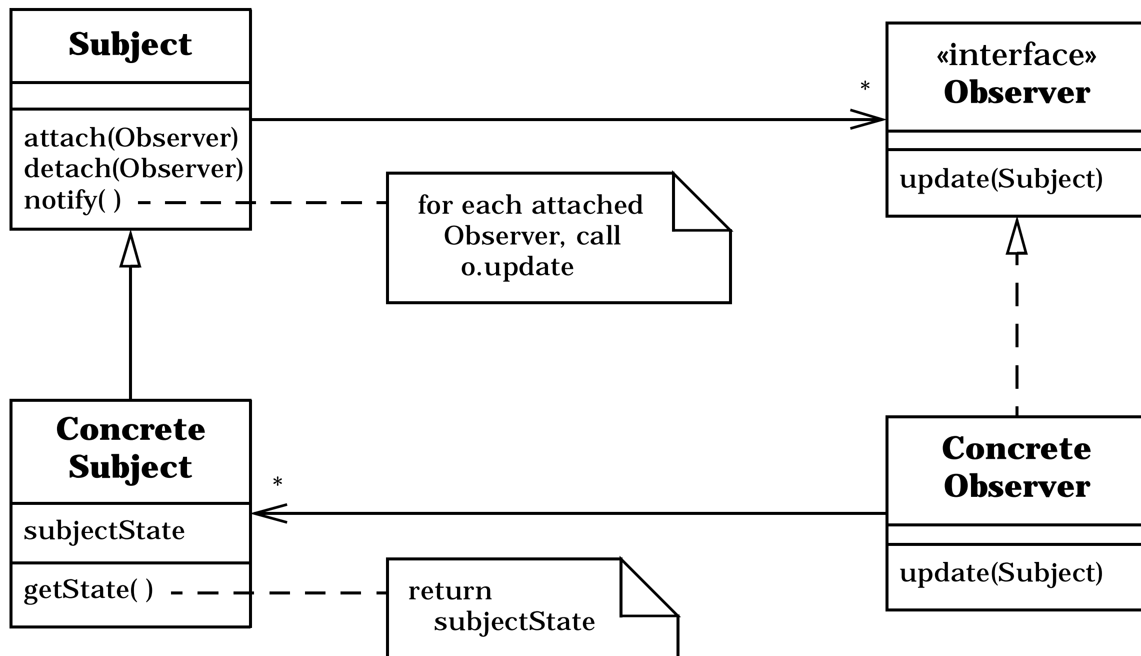

Figure 15: The Observer design pattern.

Such an arrangement is an application of the Observer design pattern [GHJ&V, pp. 293-303]. Figure 15 contains the generic class diagram for that pattern. An instance of a class derived from the Observer interface is watching for changes in one or more instances of classes derived from the Subject class.

The devices are obviously the concrete subjects in the pattern, but who are the observers who are watching devices for changes? One possible answer is that the Monitors themselves act as observers. This implies, however, that a Monitor must contain the machinery both to retain the current state of a device (in response to an `update` call), and to carry out the periodic inspection of that state. The result is an incohesive Monitor class. (What if, for example, a later design of another application must retain the state of a device, but does not include periodic polling? Reusing the device observer portion of the Monitor class entails also reusing or else removing the monitoring code.)

You might instead employ a set of distinct device observer classes, each of which observes a particular type of device. A Monitor instance uses a device identifier to locate a specific device observer instance (the observer for the device with that device identifier), after which the Monitor periodically polls that device observer instance. The remaining portions of the design are unchanged. Figure 16 illustrates this design for device D2. The design assumes the observer

© 1999-2007 by Objective Engineering, Inc.

classes and methods exactly as described by the Observer pattern. Note that a Monitor views a device observer (such as D2 Observer in Figure 16) as a generic Device reference with a single `poll` method.

You could also generalize this design to obtain a black-box framework. The framework would be identical to the previous framework, except that it would also include the Observer interface and the Subject class.

References

[GHJ&V] E. Gamma, R, Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Trademarks

Java is a registered trademark of Sun Microsystems. Inc.

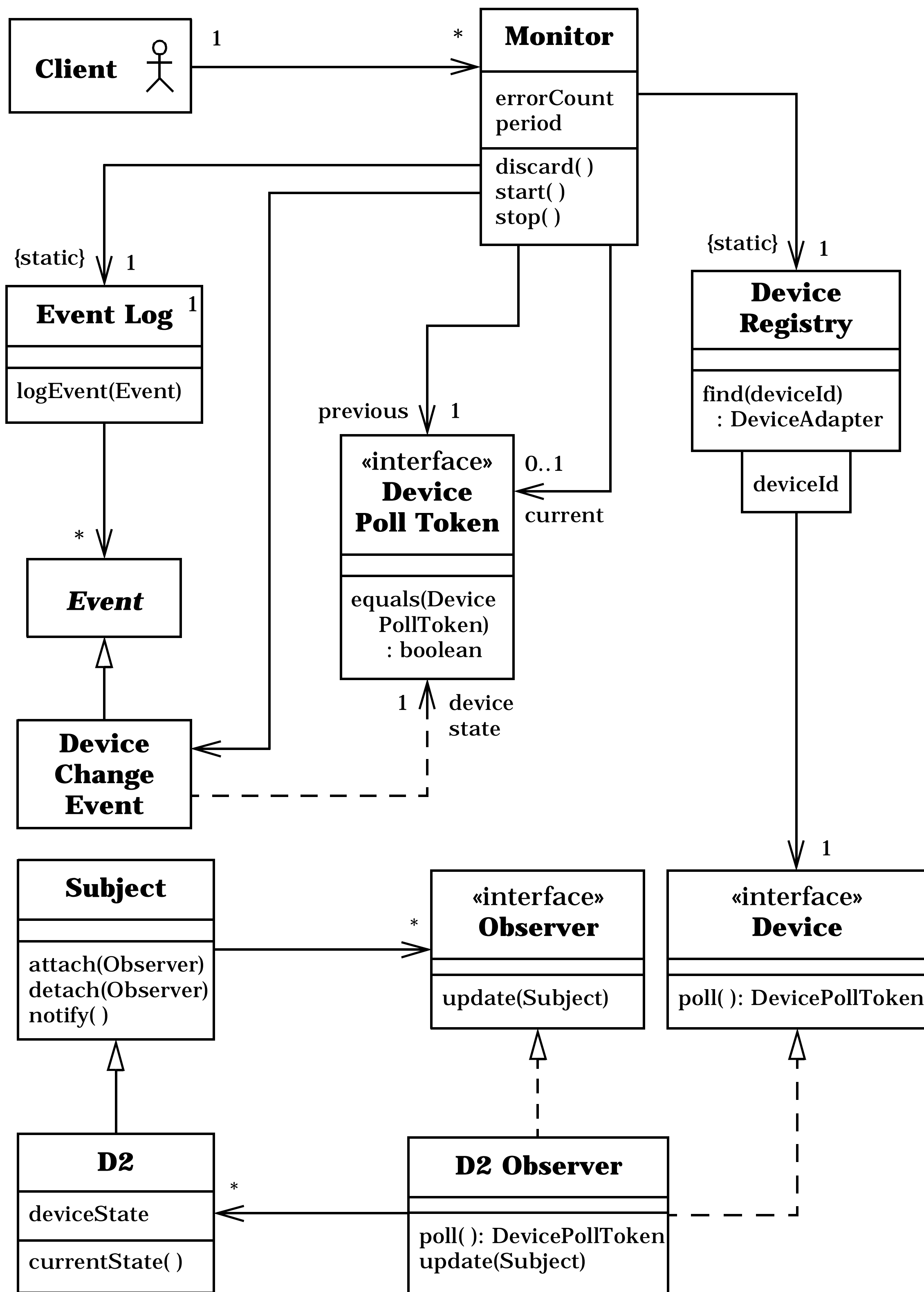

Figure 16: A class diagram for the second design.

© 1999-2007 by Objective Engineering, Inc.