INTELLIGENT SUBJECT – adapting OBSERVER with push model and filters to handle divergent update needs

Paul G. Austrem
Dept. of Information Science and Media Studies
University of Bergen, Norway
paul.austrem@infomedia.uib.no

Abstract

The OBSERVER design pattern is one of the most widely used patterns from the original GoF book [1]. With the proliferation of mobile devices in worklife and information systems serving data to such devices is paramount to maintaining data integrity in a work process. The idiosyncracies of mobile devices have placed new requirements on the mechanisms for updating resource limited clients[1] with an OBSERVER style solution. This work provides an adapted pattern named INTELLIGENT SUBJECT that allows for a SUBJECT side filtering mechanism to avoid propagating all updates to all OBSERVERS if the cost of notification is high. This cost could be either due to resource or network constraints. OBSERVERS define threshold values, and are only notified when the data value is changed beyond their individual threshold. The pattern introduces slightly more complexity, but allows for a separation of concerns on the SUBJECT side and a life of blissful ignorance on the OBSERVER side.

Introduction

Currently mobile devices are being increasingly used as integral parts of day-to-day operations in many business areas, being employed by healthcare workers [2], ticket takers on trains, as well as suggested uses for construction workers [3]. This is complimented by the increased development of mobile devices supporting constant network connectivity (through technologies such as Wi-Fi and/or HSDPA) for broadband data transfer speeds, along with GPS technology for location data[4] . These technologies pave the way for *mobile knowledge workers* to utilize information on-the-go to improve their workday, as they can now access a centralized information system or data source through their network connectivity, and enrich information retrieval techniques with contextual information through the use of GPS location data. This work uses the definition of a mobile knowledge worker as a person who does not have a stationary workplace and who is dependant on updated information in order to perform their work tasks. Note however, that the domain of mobile knowledge workers is not normative for the pattern, it is exemplary. The pattern may of course be used in other situations and contexts, the specific domain is applied here because it brings forth many of the benefits of this adaptation of the original OBSERVER pattern.

How data is used by different applications on a mobile device may vary. For instance maybe you are running several different applications simultaneously on your mobile device. On a mobile device (hereafter referred to as a `Client`) with constrained resources, there should be an aim to minimize unneccessary inter-process calls such as with applications actively polling a shared resource on the `Client`[5, 6].

Both [5] and [6] offer an alternative to making the `Client` responsible for retrieving location data in a data pull-manner. In this paper, the OBSERVER pattern [1] is used to allow `Client` applications to register with, for example, a `LocationManager` and receive either periodic

[1] Devices that are battery powered, have limited memory or have reduced processing power.

updates [5], or updates whenever the user has moved beyond a set proximity [6]. The pattern has also been applied in the Symbian OS for mobile devices as part of the MVC pattern. It is in this context used to notify views of updates/changes to the model.

The OBSERVER pattern is used to offer this functionality. The pattern is one of the most widely applied patterns in software today. It allows a system to achieve consistency among objects whilst maintaining loose coupling between them. This gives you a system that is flexible and extendable with loose coupling without breaking the OPEN-CLOSED PRINCIPLE [7] (page 57).

The purpose of this paper is to present a filtering mechanism to avoid propagating all updates to all observers. To help illustrate this, we may use an analogy to a news publisher within a niche market. The news publisher charges his subscribers on a per news update delivered basis. Suffice to say not all subscribers wish to receive all the niche market news updates, thus the publisher has decided to offer custom subscription packages, where one can subscribe and receive all news updates, or only headline / breaking news updates depending on the individual needs of the subscribers.

Offering this functionality introduces a new challenge, the GoF [1] state this as the ”Push or Pull” model. In the ”Pull” model the `Subject` merely issues a notification of change without providing any extra information to the `Observers`. This means the `Observers` themselves must discover what has changed, and whether it is relevant for them. Conversely in the ”Push” model the `Subject` ”pushes” extra data to the `Observers`. Essentially the `Observers` get the change information served directly to them parametrically[2].

The example of the news publisher being the `Subject`, whereas all the customers are the `Observers` shows how to place responsibility onto the `Subject`, This goes beyond just using a push model, in addition the `Subject` must deal with what is analogous to the ”subscription type” of the `Observer`. The reason for using this model is to avoid unneccessary memory usage if dealing with ”heavy-objects”. A different pattern that resolves similar issues locally in an application is the VIRTUAL PROXY [1] and the ”Lazy” family of patterns (LAZY INITIALIZATION [8], LAZY LOAD [9]) for datalayer to businesslayer retrieval.

If the `Subject` is a provider of large and/or complex objects this will naturally take up significant amounts of memory on the `Observer` devices and induce performance issues if transferred over a network. If these complex objects are not necessarily needed by the `Observers` then this is a waste of resources.

On mobile, resource limited devices, a design should strive to constrain the memory footprint and inter-process calls of an application to a minimum. The initial memory footprint is affected by how many classes are loaded at initialization; for instance, loading entire libraries such as `System.Graphics.*` is wasteful if you do not actually need all the classes in the package. Secondly, the number of objects initialized and allocated will affect the memory footprint. Thirdly, all method calls will incur some overhead; although this is barely noticeable in intra-process calls it may have an effect on inter-process calls. These should therefore be minimized.

[2] A variation of this is the ”Event Listener” pattern wherein an Observer when registering with the Subject passes in a reference to an object that implements a pre-agreed method signature. The method signature contains a subclass of an abstract Event Class as an in parameter. This way the Event information is pushed to the Observer. This approach is used extensively in the Java.AWT and Swing components.

The original pattern of the GoF is named OBSERVER, this pattern has been named INTELLIGENT SUBJECT to emphasize the dominating role played by the Subject.

Intent

Relieve the `Observer` of all duties and avoid unnecessary resource usage when dealing with solutions where data passing from `Subject` to `Observer` is costly and `Observers` have divergent update needs by extending the `Subject` and giving it added responsibility.

Problem

You are faced with multiple clients each with differing needs for updates. Their needs may be based on limited resources, etc. thus they may only require updates when changes have gone beyond a certain level, or *threshold*. These requirements are individual. How do you accommodate varying needs in update frequencies in clients but still make this transparent for both the *subject* and the *observers*?

Motivation

If we break down the previously defined phrase 'mobile knowledge workers', we can tentatively motivate the use of the pattern. *Mobile* devices may imply limitations on resources in terms of capabilities or performance, or due to the cost of use or portability. *Knowledge* implies that the *workers* are dependant on information in order to do their job; for example, a healthcare worker or train ticket collector. The workers must have timely information available in order to do their job correctly, or the results could be less than agreeable. However, depending on the accuracy needs of the application the data may not need to be updated constantly. For instance, not all `Observers`, whether they be applications or different `Clients`, may require the same accuracy or timeliness.

Applying our real-world analogy, a person who is a news subscriber may choose to subscribe to only the headline news *if the cost of the subscription is too high* for a full news subscription. Similarly, an application on a resource limited device could opt to only receive updates when the data has changed by a pre-defined amount if notifications are costly. Note that although the example of a mobile knowledge worker *motivates* the pattern, it does by no means limit the applicability of the pattern to resource limited devices of the domain of mobile information systems.

An example of this could be in a financial application wherein certain `Clients` (`Observers`) are so resource limited that they cannot receive updates too frequently seeing as the updates are costly. Thus they only desire updates when values change beyond a certain limit or threshold.

This implies that the INTELLIGENT SUBJECT pattern gives the `Subject` additional responsibilities. Due to the fact the `Subject` must actively handle which `Observers` are to receive notifications anytime the data changes.

Forces

You are dealing with situations where there is a real need to provide updated information to different observers with varying requirements to data freshness. The solution must be stable in its interface and easy to bind to for observers, but at the same time it must be flexible and capable of accommodating differing needs. This creates an overarching force of providing a static interface while allowing for dynamic behaviour.

Applicability

The INTELLIGENT SUBJECT pattern can be applied in the following scenarios:

- Use the pattern when a `Subject` object needs to notify a dynamic list of unknown `Observer` objects without inducing strong coupling between the objects *and the `Subject` must handle divergent update needs from the `Observers`.*
- You need to utilize a ”push” model[3], however the cost of passing the `eventData` is too high to justify it being passed when the value of the data is of no significance to the receiver; for instance, because the data value change is too fine. Frequently pushing the `eventData` will lead to unacceptable performance. The performance cost can reside with the `Subject` or the `Observers`, or even both. On the `Subject` side, the cost may be associated with network constraints; for instance, messages fail to reach the intended Object, forcing the `Subject` to resend the message even though the `eventData` in the message is of no interest to the `Observer`. Contrarily, the cost may reside with the `Observer` if the cost of processing the received `eventData` is high in terms of computational power (which on a battery-powered device would translate directly into draining the battery). In which case it would be preferable for the `Observer` to only receive updates that are relevant.
- The `Observers` need to do cascading updates to many different aspects / objects upon receiving `eventData`. This is costly. Avoid this by defining upfront limits/thresholds for when to receive updates.

Solution

Create a separate class that handles the notification to only those observers that require it based on their individual thresholds. The Subject does not know, nor does it care, which of the observers actually receive its updates. Similarly the Observers do not know whether there have been sent out updates that they have not received, they are only notified whenever a change happens that exceeds their personal threshold. The Observers are thus able to create their own universe of state, or sphere if you will; which is not intruded or “contaminated” with unnecessary or uninteresting data. The following sections present the solution in more detail, starting with a structural view. This is followed by a behavioural view and a presentation of the participants, before finally a code sample and implementation guideline is provided.

[3] Perhaps because the client devices do not have pull capabilities.

Structure

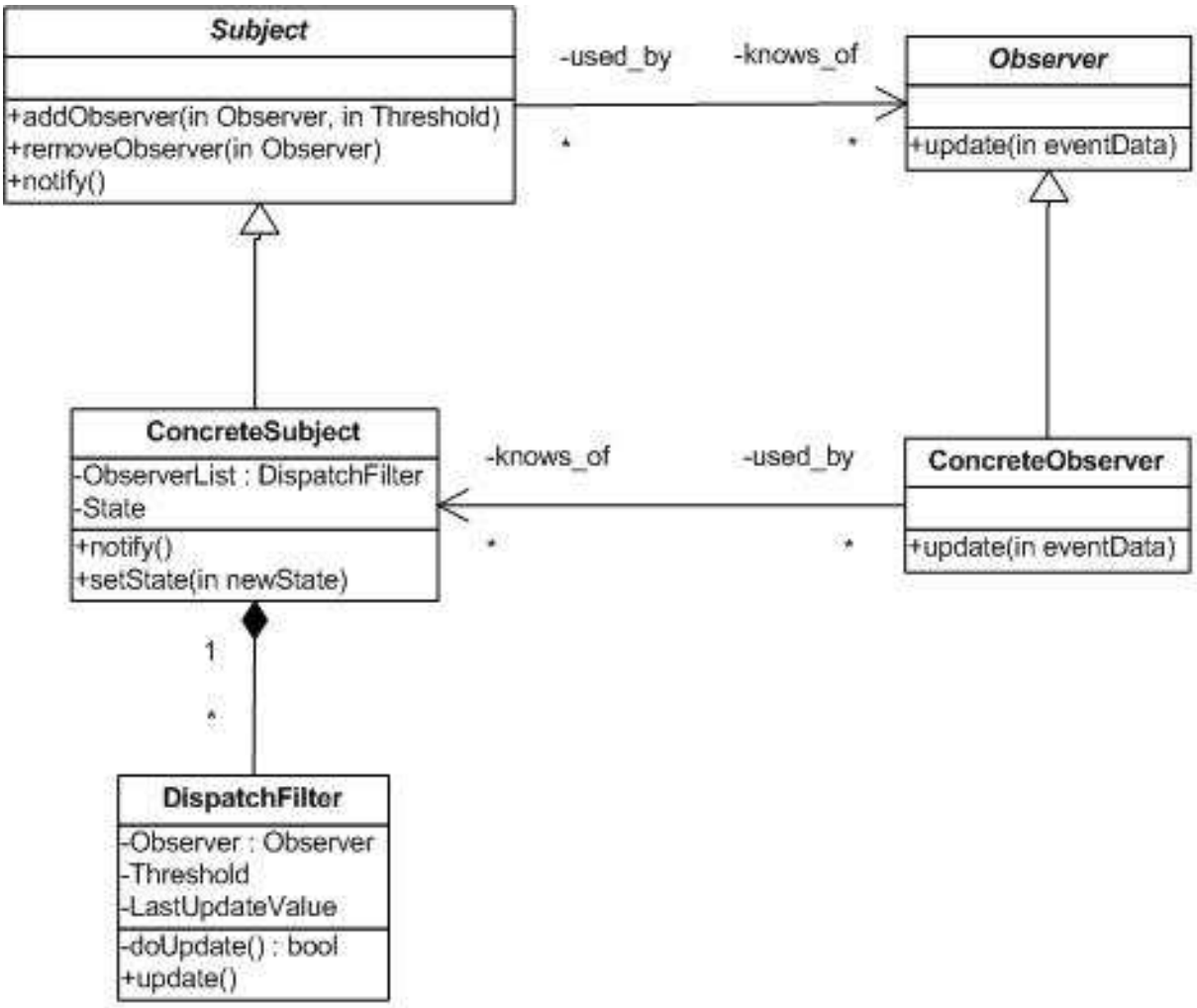

Figure 1: Class diagram of the INTELLIGENT SUBJECT.

The difference between the original OBSERVER pattern, and INTELLIGENT SUBJECT is in the addition of the `DispatchFilter` class. Though we previously stated that more responsibility is placed on the `Subject`, in that context, the term `Subject` was only considered conceptually. As we can see from the class diagram in figure 1, the pattern uses ”part-whole” composition because the `DispatchFilter` is contained within the `ConcreteSubject`. A filter could per se exist without the `ConcreteSubject`, however there would be little point in this since the `DispatchFilter` is uniquely associated with the `eventData` values of each individual `Subject`.

Another shift of responsibility is that the `ConcreteSubject` no longer invokes methods on the `Observers` directly. This task is delegated to the `DispatchFilters` contained in the `ObserverList` attribute. This accomplishes two things; the `ConcreteSubject` now has no knowledge of the specific needs of any `Observers`, nor should it. The `ConcreteSubject` knows only how many `Observers` are registered at any given time in its list, but that is all the knowledge it has. Additionally, this separation enforces the LAW OF DEMETER[4].

[4] Essentially, the law of Demeter states that a method *M* of object *O* may only invoke the methods of closely connected/related objects.

The separation of `DispatchFilter` into a separate class is crucial to avoiding DIVERGENT CHANGE, one of the many malodorous symptoms described by Fowler [10]. We are dealing with two distinct behaviors, lumping them both in with the `Subject` class is unattractive. The `Subject` class deals only with receiving notifications from the `Client` (assuming this is the notification model used), wrapping the whole event up in an `eventData` object, and notifying each of the members in the `ObserverList.` Only a reference to the `eventData` is passed to the `DispatchFilter` objects which save the load of a possibly large `eventData` object being unneccesarily transfer. This leads us on to the second behavior, namely the evaluation and propagation of `eventData` to the registered `Observers`. The task of evaluation is closely tied into the threshold values of the individual `Observers`, thus it should be performed by the `DispatchFilter` which is object that is composed of the `Observer` and threshold value. Additionally, the `DispatchFilter` must handle the computation of the difference between the new `eventData` value and the `lastValue` of the `Observer`. If the difference is greater than the threshold then an update will be initiated.

As a comment on Figure 1, it is plausible to create the object `observerList` class as a generically derived class parameterized with `<Filter>`, which the `Subject` class would then bind to. In such a case, we would be using an association between `DispatchFilter` and `Subject` instead of composition between `ConcreteSubject` and `DispatchFilter`. The advantages of this would be that a layer of indirection would be removed (the `ConcreteSubject` class) and we would enforce type-checking, and also ensure that the *addObserver* and *removeObserver* methods are correctly implemented by the `DispatchFilter` class. However, this is only applicable in certain strongly typed languages (although many languages do now support it with Java Generics and C# Templates), and does have consequences in terms of ”code bloat in [for example] C++” p. [11] .

The INTELLIGENT SUBJECT adaption of the original OBSERVER is reminiscent of the MEDIATOR pattern as described in the GoF book [1]. However, whereas the MEDIATOR pattern is concerned with centralized control of complex interactions between objects, in order to decrease the coupling between them, the INTELLIGENT SUBJECT is concerned with centralized control of divergent update needs. Concisely stated; MEDIATOR handles centralized control of cascaded / dependant updates, INTELLIGENT SUBJECT handles divergent update needs.

Participants

- **Subject**
  - knows of its `Observers` and offers all the method signatures needed by `Observers`, to add and remove themselves.
- **Observer**
  - Is an interface used by concrete `Subject` objects to update the registered `Observers`.
- **ConcreteSubject**
  - This participant extends the `Subject` base class. It also stores the state that is the basis for all `Observer` updates.
- **ConcreteObserver**
  - Knows of the `ConcreteSubject` so that it can attach itself and remove itself from the Subject’s list of `Observers`. Implements / Extends the `Observer` supertype to stay synchronized with the methods and signatures used for updates.

- **DispatchFilter**
  - This class is delegated the task of directly invoking the *update* method of all `Observers` where the `eventData` value exceeds their threshold value. It is also responsible for retrieving the state from the `ConcreteSubject`.

Collaborations

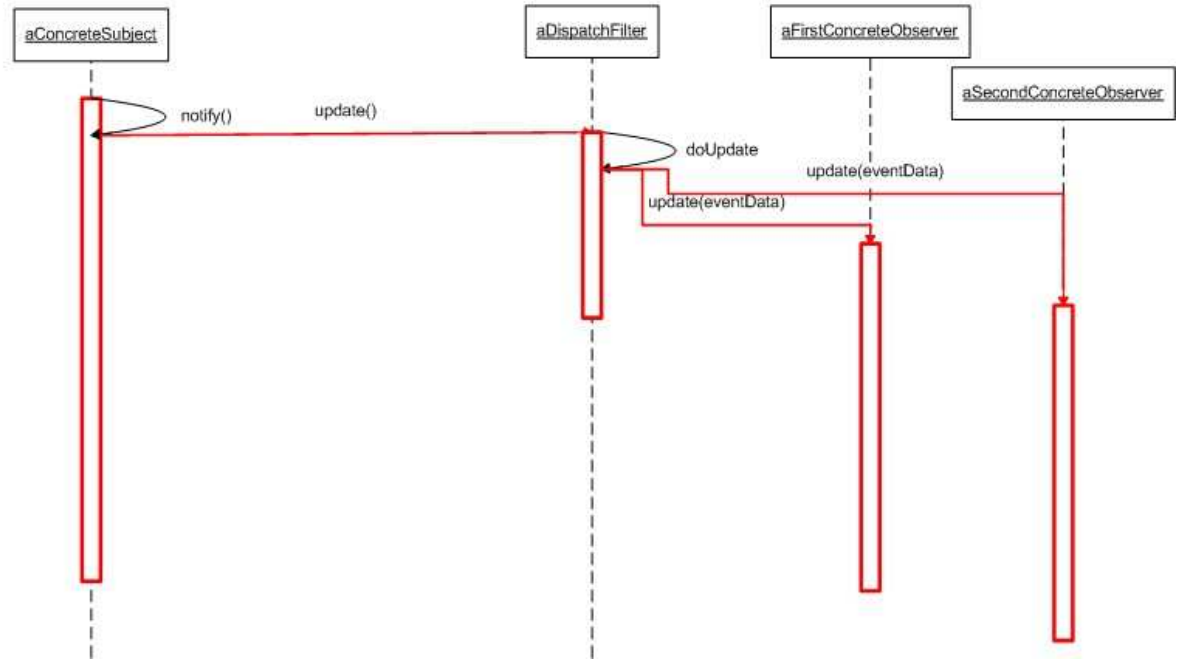

Figure 2: Sequence diagram of the object interactions during an update.

The sequence diagram shows the interactions between the objects during an update event. Initally the object `aConcreteSubject` will receive a message call to its *notify* (this is not shown in the above diagram). This will prompt the object `aConcreteSubject` to invoke its *iterateList* method. Essentially this is where the responsibility of the `concreteSubject` object ends. It invokes the *update* method on each `DispatchFilter` object in its `observerList`, it is then the `DispatchFilters` responsibility to decide whether or not the `Observer` object contained in the Filter object is to be updated with new `eventData`. This is depicted as the self-call of *doUpdate* the `aDispatchFilter` object-lifeline in figure 2. The *doUpdate* returns a boolean value after having compared the difference between the specific `Observer` object's `lastValue` attribute and the new `eventData` value against the `threshold` attribute value of that `Observer`. This is accomplished through operator overloading, since depending on the type of the `eventData` the operator symbols of greater than and less than may not natively be supported. In which case the `Observer` must overload those operators to function with the `eventData` type.

Code Sample and Implementation Guidelines

The following code samples show a C# skeleton implementation and the mechanisms behind the INTELLIGENT SUBJECT pattern.

Below is the *IObserver* interface which all classes that wish to observer the *Subject* must implement. For simplicity's sake, we are merely using an int object[5] as the eventData

[5] since the example is written in C# where all types ultimately derive from System.Object, as opposed to Java where int is an immutable primitve disconnected from the object model.

parameter that gets passed to the observers. Note that in a real implementation the eventData object would be more complex, and consequentially could be passed as a struct which is slightly more efficient as reported by [12].

```
public interface IObserver
{void Update( Object state );}
```

The class below is the *ConcreteObserver* which implements the *IObserver* interface.

```
public class ConcreteObserver : IObserver
{
   public void Update(Object state )
   { Console.WriteLine(id + " updated with " + state.ToString());}
   . . . .
}
```

The following (partial) abstract class is the Subject supertype, correlates to the Subject in the participant list. It also has methods for allowing observers to remove themselves and for notification.

```
public abstract class Subject
{
   private const int arrayno = 10;
   protected static DispatchFilter[] observerList = new
   DispatchFilter[arrayno];

   private int counter = 0;

   public void addObserver( IObserver observer, int threshold)
   {
         if(counter < arrayno)
         {
            observerList[counter] = new DispatchFilter(observer,
            threshold);
            counter++;
         }
         else
         {//throw an exception here}
   }
}



public class ConcreteSubject : Subject
{
   private int State;

   public void setState(int s )
   {
      this.State = s;
      notify();
   }


   private void notify( )
   {
         for (int arrayIterator = 0; j < counter; arrayIterator++)
         {observerList[arrayIterator ].update(ref State);}
   }
}
```

Listing 1: Code sample showing skeleton of the Intelligent Subject pattern.

The code snippet above, `ConcreteSubject` is the class which extends the abstract class `Subject`. As we can see it offers no method for `getState()` as the original OBSERVER pattern does, this is because INTELLIGENT SUBJECT enforces a push model, thus there is no need for Observers to be able to programmatically retrieve state since it is pushed to them as a parameter in the `Update` method.

Whenever the state is set, the `notify()` method is called. Note that a caveat about the sample above is that calling the `notify()` method sequentially with the state setting operation is not advisable. This is because in a real-life implementation the state setting procedures may be complex involving many steps, and multiple calls to the `setState()` method. Therefore the `Client` would not want to call the notification `notify()`until after the `setState()` method had been called for the last time. In practice, this is easy to implement; simply extract the call to `notify()` and place it in an overridden `notify()` method call. Thus the `Clients` could call `notify()` to run the updates. The only reason we didn't was to simplify the example.

Below in listing 2, the code that handles the filtering and `Update()` calls to observers is presented.

```
public class DispatchFilter
{
      private IObserver Observer;
      private Object Threshold;
      private Object LastUpdateValue;

      public Filter(IObserver observer, Object threshold) {
         this.Observer = observer; this.Threshold = threshold;
         LastUpdateValue = 0;
      }

      public void update(ref Object state ) {doUpdate(ref state);}

      private void doUpdate(ref Object state) {
         if (LastUpdateValue != 0 && beyondThreshold(state))
               Observer.Update(state);

         else if(LastUpdateValue == 0)
         {
               LastUpdateValue = state;
               Observer.Update(state);
         }
      }


      public bool beyondThreshold(Object state )  {
         return (getDifference(state) > Threshold);
      }

      public int getDifference(Object state ) {
       return state - LastUpdateValue; }
}
```

Listing 2: Code for the Filtering class which handles the *Update()* calls to the conditioned Observers

The `DispatchFilter` class encapsulates the behavior required to update the Observers and handle the task of filtering out which Observers are to receive updates. The `ConcreteSubject` class will invoke the `update()` method, of `DispatchFilter` objects maintained in its `ObserverList`, and pass in the `eventData` object (in our vanilla example this is just a simple Object) as a reference. Note that the `ConcreteSubject` *does this for all the* `DispatchFilter` *objects in its list*. It must be done this way to enforce the separation of concerns, and encourage the high cohesion of the Subject and DispatchFilter classes. Note that this approach (in certain languages) is not costly since passing `eventData` as a reference in-process is performance wise economical, and allows a higher cohesion in the `ConcreteSubject` class. This pass-by-reference approach is idiomatic to the C# programming language, and is also doable in C++, however it will not be possible in for example the Java language. In which case there might be a slight performance penalty, but in-process passing-by-value is not overly costly, so the message passing architecture is not bound to any specific programming languages.

The `DispatchFilter` object will then check whether the new `eventData` difference value exceeds the threshold of the individual `Observer`. If so, then the `Observer's Update()` method is called.

Note that this design lends itself well to Meyer's NON-REDUNDANCY PRINCIPLE [7] in the constructor of `DispatchFilter` and in the `addObserver()` method of `Subject`.

The NON-REDUNDANCY PRINCIPLE states that ”under no circumstance shall the body of a routine ever test for the routine’s precondition” [7] (page 343). We see that although the *addObserver* method in Listing1 does do a check on the size of the array, it does not do any checks on the integrity on the parametric data passed in. This affects the responsibility distribution, essentially the code sample operates with ”demanding pre-conditions” [7] (page 343).. The responsibility is to a larger degree shifted to the `Observer` which must ensure that any data passed when registering is correct, in this code sample if the data is not correct and the registration fails the `Observer` will not receive any notification of this. This approach goes against the paradigm of ”defensive programming”, and Meyer argues that the NON-REDUNDANCY PRINCIPLE allows for reduced complexity and increased reliability; this is called ”the zen-style paradox…: that to get *more* reliability the best policy is often to check *less*” [7] (page 345).

Consequences

A consequence of the INTELLIGENT SUBJECT pattern is the shift of responsibility between the `Subject`/`DispatchFilter` dyad. The `Subject` becomes a class that holds a list of all registered `Observers` under the guise of `DispatchFilter` objects, which handle the tasks of adding and removing `Observers`. However, the `Subject` no longer has the responsibility of communicating with `Observers` to Update them, this is now delegated to the `DispatchFilter` class. Compared to the original OBSERVER pattern, this variation is more complex as you use delegation to provide the filtering mechanism through a separate filter object. Additionally, there is transparency between the classes; for instance, the `Observers` do not know that there is a separate `DispatchFilter` class that updates them with new `eventData`. If they at times are bypassed, it is because the `eventData` change is below their threshold value, causing them to remain completely oblivious to the change. Therefore a chance of *data disalignment* between `Observers` can occur. This can be troublesome if the `Observers` in a different part of the system cooperate or collaborate and their data is not the same because they have different threshold values registered with the `Subject`. Thus they may have received a different number of updates, in which case one of the `Observers` would have more accurate and more timely data than the other. This could be solved by timestamping the `eventData` so that the `Observer` with the freshest data would trump the `Observer` with stale data.

Known uses

The traditional OBSERVER pattern (and minor variations on it) have been widely used in object-oriented event driven software designs. In the .Net and Java frameworks, the traditional OBSERVER pattern is utilized extensively in their delegate-event models [13, 14]. It is also used in the architecture of Symbian S60 platform for mobile devices [15]. The traditional OBSERVER has been used in the Java packages java.awt and javax.swing for handling notifications between graphical artefacts, event-triggers and the event listeners which handle the business logic.

The adoption presented here is viable in domains of resource limited devices, or in systems where any `eventData` is propagated over a network with limited bandwidth. Propagating this data to `Observers` who do not need it should be avoided. Concepts from the INTELLIGENT SUBJECT pattern have been used as part of Google’s Android Location API framework [16] wherein it is possible to register (an `Observer`) with a

`LocationProvider` (`Subject`) with a set `ProximityAlert` (conceptually a `threshold`), thus the `Observer` will not receive updates all the time, only when the proximity alert is triggered.

Related patterns

The original OBSERVER and INTELLIGENT SUBJECT are high-level design patterns. It would be feasible to use other patterns such as FACTORY METHOD to create Filters, or to use SINGLETON to ensure there is only one list object containing the Observers. As mentioned previously, the MEDIATOR pattern is similar to INTELLIGENT SUBJECT in its functional aims. Concisely stated, MEDIATOR handles centralized control of cascaded / dependant updates, whereas INTELLIGENT SUBJECT handles divergent update needs. The SASE OBSERVER [17] variation is similar, it allows the `Observers` to register with the subject, and at registration time identify themselves, register which events they care about, and register what event data they request when the event fires, and also possibly what they should do with the event data. Although very similar in many of its intents, the SASE pattern gives a different distribution of responsibility. It allows the `Subject` to dictate the `Observers` response and processing of events, whereas INTELLIGENT SUBJECT enforces an opaqueness between the `Subject` and its `Observers`; thus, the `Subject` does not know and does not care what happens to the data after it has been pushed to the `Observers`.

Another option is that the `DispatchFilter` could implement the STRATEGY pattern allowing for interchangeable algorithms to be applied to handle the filtering. Thus, the mechanisms that go into differentiating between which `Observers` receive updates could be run-time pluggable. This could allow the `DispatchFilter` to take on a policy-enforcer approach to notification, thus information could be disseminated not only based on which `Observers` that have registered for it, but also based on internal policies set forth by the `Subject` as to which `Observers` qualify as recipients (maybe based on the sensitivity of the information).

Finally, Niblett and Graham propose in the IBM Systems journal [18] a pattern called the NOTIFICATION PATTERN, also known as the SOA NOTIFICATION PATTERN. This pattern is manifested in the WS-Base Notification specification. The pattern is an alternative to INTELLIGENT SUBJECT as it allows for a `filter` to determine which `Observers` are to receive messages, thus not all notifications are propagated to all registered `Observers`. However the main difference is in INTELLIGENT SUBJECT dealing with compounded results, in the form of triggering thresholds as a mechanism for filtering. The NOTIFICATION PATTERN is more concerned with direct conditional limitations, such as topic based limitations. Furthermore INTELLIGENT SUBJECT is closer to the original OBSERVER [1] in that detachment can only be done by direct OBSERVER initiation, whereas NOTIFICATION PATTERN allows for temporal subscription based detachment so that an `Observer` may detach at a predefined time in the future.

Acknowledgements

I would like to thank my shepherd Maurice Rabb for his insights and contributions to improving this pattern. He provided strong technical advice, and many pointers to relevant materials, along with encouragement and support. I would also like to thank Jim Siddle for his oversight and suggestions, and also Andreas L. Opdahl for his comments on early versions of the paper. Finally I would like to thank my group at Euro-PLoP for their great feedback. .

References

1. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994: Addison-Wesley Professional. 416.
2. Christensen, C.M., J. Kjeldskov, and K.K. Rasmussen, *GeoHealth: a location-based service for nomadic home healthcare workers*, in *Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments*. 2007, ACM: Adelaide, Australia.
3. May, A., et al., *Opportunities and challenges for location aware computing in the construction industry*, in *Proceedings of the 7th international conference on Human computer interaction with mobile devices \& services*. 2005, ACM: Salzburg, Austria.
4. Patel, N. *Mobile World Congress Roundup: Cellphone Mania*. [Webpage] 2008 11/2-2008 at 10:16pm [cited 2008 11/2-2008]; Webpage summarizing all the cellphones released during the first week of Mobile World Congress 3GSM in Barcelona]. Available from: http://www.engadget.com/2008/02/11/mobile-world-congress-roundup-cellphone-mania/.
5. Mahmoud, Q.H. *J2ME and Location-Based Services*. 2004 [cited 2008 10/1-2008]; Available from: http://developers.sun.com/mobility/apis/articles/location/.
6. Android, G. *Location-based Service APIs*. 2007 [cited 2008 10/1]; Available from: http://code.google.com/android/reference/android/location/LocationManager.html.
7. Meyer, B., *Object-Oriented Software Construction*. 2nd Edition ed. 1997, Upper Saddle River, New Jersey: Prentice Hall PTR. 1254.
8. Beck, K., *Smalltalk Best Practice Patterns*. 1996: Prentice Hall PTR. 240.
9. Fowler, M., *Patterns of Enterprise Application Architecture*. 11th printing ed. The Addison-Wesley Signature Series. 2003, Boston: Pearson Education. 530.
10. Fowler, M., *Refactoring: Improving the design of existing code*. Object Technology Series, ed. J. Booch, Rumbaugh. 1999, Reading, Massachusettes: Addison-Wesley. 431.
11. Fowler, M. and K. Scott, *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*. Second Edition ed. Object Technology Series, ed. J. Booch, Rumbaugh. 2000: Addison-Wesley.
12. Kayun, C. and S.-a. Chonawat, *Energy conscious factory method design pattern for mobile devices with C\# and intermediate language*, in *Proceedings of the 3rd international conference on Mobile technology, applications \& systems*. 2006, ACM: Bangkok, Thailand.
13. Richter, J. *An Introduction to Delegates*. MSDN Magazine The Microsoft Journal for Developers 2004 [cited 2008 22/1]; Available from: http://msdn.microsoft.com/msdnmag/issues/01/04/net/.
14. Microsoft. *Implementing Observer in .NET*. Microsoft Patterns and Practices 2003 [cited 2008 20/1]; Available from: http://msdn2.microsoft.com/en-us/library/ms998543.aspx.
15. Developer Community Wiki, N. *Design Patterns in Symbian*. Developer Community Wiki 2008 [cited 2008 15/1]; Available from: http://wiki.forum.nokia.com/index.php/Design_Patterns_in_Symbian#Observer_Pattern.
16. Google. *Google Android android.location.LocationManager*. 2007 [cited 2008 22nd of March 2008]; Available from: http://code.google.com/android/reference/android/location/LocationManager.html.
17. Brown, K. *Understanding inter-layer communication with the Self-Addressed Stamped Envelope (SASE) pattern*. [Webpage] 1998 18th of March 1998 [cited 2008 27/3-

2008]; Available from: http://members.aol.com/kgb1001001/Articles/SASE/sase2.htm.
18. Niblett, P. and S. Graham, *Events and service-oriented architecture: The OASIS Web Services Notification specifications*. IBM Systems Journal, 2005. **44**(4): p. 17.