On Using the Observer Design Pattern

Constantin Szallies
Energotec GmbH, Germany
szallies@energotec.de

21. Aug 1997

Abstract

The so-called Observer design pattern is introduced in [1]. The following article describes this pattern, tries to explain why the pattern is helpful and takes a closer look at several consequences of using the pattern. Code examples are given in a java-like syntax.

I have tried to avoid repeating things about the pattern already described in [1], the reader should be familiar with the particular section. You can find an HTML version of the Observer section under *http://www.inf.fu-berlin.de/ bokowski/Observer.html*.

I would like to thank Luther Hampton (hampton.luther@ist.mds.lmco.com) for reviewing this paper and for his valuable comments.

1 The Observer Pattern

The Observer design pattern, or more accurately some implementation of this pattern, is used to assure consistency between different objects. For example, let the state of an object "B" depend on the state of an object "A". Whenever the state of object "A" changes, "B" has to recompute its state in order to remain consistent with "A". The observer design pattern supports such dependencies and at the same time tries to reduce the coupling between the object that changes (the subject) and the object that needs a change notification (the observer).

Here's an example of the interfaces in java-like code:

```
public abstract interface Subject {
public abstract void addObserver(Observer observer);
public abstract void removeObserver(Observer observer);
public abstract void notify();
}

public abstract class Observer {
public abstract void update();
}
```

The subject calls its "notify" operation whenever it changes. "Notify" calls the "update" operation on each observer. The observers then call back to the subject and update themselves accordingly.

An observer may wish to remove itself from the observer set as part of the "update" method. If this is allowed, make sure this removal doesn't affect the iteration over the observer set. If an iterator object is used, make sure the iterator can handle modifications to the set during iteration. If not, make a shallow copy of the container object first and iterate over the copy.

The observer design pattern can be used for purposes other than keeping two objects consistent. Since the focus of this article is on maintaining state dependencies by using the Observer pattern, other uses (for example, notifications of events other than "state did change") are not discussed.

2 Why Use the Pattern?

State dependencies (or, more formally, "consistency conditions") between objects are unfavorable for several reasons. If we could model our problem domain as a set of independent classes, we would get a software system without state dependencies. Coding and especially maintaining such a system would be much easier. The programmer could focus on single classes only.

One of the basic principles of object-oriented design is to encapsulate data in objects. To achieve good encapsulation, data managed by different objects should be loosely coupled while data managed within an object should be highly cohesive [2].

A class with high cohesion should have the following properties:

1. The object's data should somehow "belong" together.
2. There should be consistency conditions between the data.
3. The consistency conditions should be enforced by the object's methods.
4. The data plus the methods which operate on the data should form one abstraction.

Using too few classes often leads to a situation violating point four. On the other hand, if you put everything into the same object, you end up with a set of global variables and functions and therefore with no encapsulation at all.

Loose coupling between classes means:

1. There are few consistency conditions between the data of any two objects.
2. Each class should know few other classes. If the class knows some other class, it should use not only a few methods but the entire abstraction.

Point 2 is interesting: If one part of the class interface is used in some context and the other part is only used in some other context, then the class may contain two different abstractions. Think about defining two interfaces and inherit from both of them.

If state dependencies are bad, we could view our problem domain as a sea of data with a set of consistency conditions on this data. Then we could wrap objects around the data in such a way as to minimize inter-object dependencies. This would result in loosely coupled but highly coherent classes. Would this strategy result in "good" design as well? Maybe, but probably not!

Consider the following:

- Consistency conditions tend to define a tightly connected network. However, you have to partition your data somewhere, otherwise your classes would be too "fat".
- The problem domain and therefore the consistency conditions tend to change. You may start with a good design and end up with garbage.
- In my opinion, data encapsulation is not a "natural" concept that is experienced in "real" life. If a modeler tries to model "things" in the real problem domain as classes, this may or may not result in a well-encapsulated design. Modelling the real world may result in a class diagram/ interaction diagram that is easy for people familiar with the problem domain to understand—but will this lead to code that is easy to understand and easy to maintain? This is at least questionable.

It's sometimes better to break the rules. Think about a domain object displaying itself on the screen. Someone familiar with object-oriented principles but lacking real-world coding experience will find this natural. And it is! But real world software uses the Model-View-Controller (MVC) architecture for this [3]. MVC separates the responsibilities for maintaining data and displaying data. This is done for flexibility and reusability reasons but it weakens encapsulation and additional state dependencies, dependencies between the model and view classes, are added to the system.

In [1] the MVC architecture motivates the Observer pattern. But even if we don't separate data and behavior for this purpose, most reasonable designs of larger size will include situations where data and behavior are separated. Therefore a well understood approach to manage state dependencies is valuable.

3 Inter-Object Assertions

B. Meyer introduced so-called assertions in the programming language Eiffel [5]. Assertions help to ensure program correctness by letting the programmer add consistency conditions for an object's state and its state transitions. Even if it's not easy to capture all correct transitions by using pre- and post-conditions, assertions are a big help for program correctness and they add some formal class level documentation as well.

Because of encapsulation, it's easier to keep a single object consistent than a set of objects. But to verify that a system is in a consistent state, it's not enough to test each object alone. A global view onto the system is required — a problem that shows up in object-oriented testing as well.

Unfortunately, assertions in Eiffel don't handle consistency conditions between different objects!

If you use Eiffel and assertions, your bugs will show up as failed assertions at runtime or as inconsistencies between objects. If you are a rocket scientist for the ESA as well, maybe your next Eiffel application will blow up an Ariane rocket due to some inconsistency between two objects!

4 The Simple Case

In this section, we focus on the simple case: One subject with one or more observers, where the observers are completely isolated from each other.

If the state of object "B" depends on the state of object "A", it should be "B"'s responsibility to synchronize its state whenever "A" changes. How does "B" know when "A" changes state?

The first possibility is that "A" notifies "B" of any change, as in the previous code snippet. The second possibility would be that "B" checks periodically to see if "A" has changed. A third possibility would be that "B" is notified by some other object "C" and "C" is notified using one of the first two methods.

4.1 Notification

In this implementation option, "A" holds a pointer to object "B" and whenever "A" changes state, it calls some notification method of "B".

The first problem with this approach is that "A" must know the interface definition of "B". This reduces reusability of class "A" because if you need class "A", you need class "B" as well.

If "A" and "B" are closely related abstractions, the interface dependency will do no harm because the granule of reuse is the package level (packages in the UML sense) and "A" and "B" probably belong to the same package.

If "A" and "B" are independent abstractions, we can turn the interface dependency in the same direction as the state dependency by using the "dependency inversion principle" [4]:

In the subject's package, we define some interface Observer that represents the abstraction of some object that is state-dependent on the subject and that needs a notification of any state change.

Now let our class "B" inherit this interface so that it becomes an observer. "A" must only know the "Observer" interface, not the interface to "B".

If we view this strategy as an independent concept, we get the Observer design pattern. The concept can be supported by some general-purpose OO library, allowing reuse. If the update of the observer is not triggered by the subject automatically, we need an additional interface for a "subject". The only method in this interface is the "update" method that tells the subject to notify its observers.

By using the Subject/Observer interfaces, we can decouple classes representing different abstractions.

Using this approach and trying to reuse a class from a different framework may cause problems. The other class might not be aware of the "observer" concept; we may have to subclass it to make it "observer aware". Depending on the implementation this might be easy, hard or impossible. And if it works (how do you verify that it works without source code and without overwriting all methods?) performance may suffer.

What if the other class uses the pattern but has defined its own Subject/Observer interfaces? Then you have different interfaces for the same abstraction. Then again you have to subclass and, if these interfaces don't diverge too much, it will work.

This is really a problem in it's own right: the existence of different interfaces for similar or equivalent abstractions. Did you ever try to (re)use two frameworks from two different vendors that use different container classes? From my point of view, the "one abstraction — two interfaces" problem is one of the problems that makes "reuse in the large" very difficult, if not impossible — at least with the current object oriented software technology.

4.1.1 State Change Scopes

[1] discusses the following two problems using the Observer pattern:

1. The subject's state must be consistent when the observer queries it.

2. Spurious / redundant updates

If you want your subject to call its "notify" operation after any state-changing method has been executed, you may find the following extension useful to solve the above problems:

```
public abstract interface SubjectWithChangeScope
                    extends Subject{
public abstract void beginChange();
public abstract void endChange();
}
```

Two methods were added to support a "change scope". Instead of using the "notify" method, the subject calls "beginChange" on itself at the beginning of every state modifying method and "endChange" at the end. The subject keeps a "change count" that is increased for every "beginChange" and decreased for every "endChange" invocation. If the change count drops to zero, "endChange" calls "notify" to trigger the update.

Because the SubjectWithChangeScope inherits from Subject, classes using this interface can be used as normal subjects as well. An invocation of "notify" will force an update even if there is a change scope pending.

If both change scope methods are declared public, other objects can access them. This introduces the risk that the invocations will not be called symmetrically. If both methods are defined as protected, maintaining correct change scopes is the classes and subclasses responsibility only.

If you use C++ as an implementation language, you can use a "ChangeScope" class to handle change scopes automatically. The only purpose of this class is to create and destroy *automatic* instances.

- The constructor of the ChangeScope class has a reference to a SubjectWithChangeScope object as an argument. The implementation of the constructor calls "beginChange" on the subject.

- The destructor of the ChangeScope class calls "endChange" on the subject.

Note that the resulting code is exception safe [6].

[1] recommends calling update in template methods to avoid the problem of updates on inconsistent subjects. This means that you have to implement two methods for every state changing operation — one that triggers the update and one that doesn't.

With the change scope extension, you can implement equivalent behavior without doubling the size of your interface. By using the interface above, you can:

1. Implement new state changing methods

2. Overwrite any state changing method, calling the overwritten method in the implementation or not.

3. Invoke any sequence of state changing methods and trigger only one update notification.

4. If both methods are declared public, clients of the subject can call several state-changing methods nested in a change scope to avoid redundant updates.

If you use C++, just create an automatic ChangeScope object at the beginning of each method.

4.1.2 Comparing Subject States

To enhance performance, the subject's "notify" method should not call "update" on its observers blindly. Instead, the subject should first check whether any change to its visible(!) state has occurred since the last "notify" invocation. If no change has occurred, "update" is not invoked.

If the update methods change the subject's state as part of its implementation, we are in trouble:

1. Different observers will see different states of the same object.

2. Change notifications will be lost

This also may happen if an observer is itself a subject and there's a cycle in the dependency graph. See also section 5.

We could solve the second problem by making "notify" itself a state changing operation with a change scope. Depending on the situation, this will solve the first problem as well. There is some chance however that we will introduce an infinite loop were the subject toggles between different states.

I would recommend that changing the subject's state as part of the "update" implementation should be not be allowed.

4.2 Polling

In this implementation option, the observer "B" queries the subject "A" to see if there was a state change.

The advantage of this approach would be that "A" is completely independent of "B". "A" neither needs an "Observer/Subject" interface nor a pointer to "B" (or some set of observers).

In this case, there is no need for the subject to be "observer-ready". The subject doesn't have to maintain old states or call "notify" in state changing methods. The responsibility for consistency shifts completely to the observing object, increasing the subject's reusability and maintainability.

From this point of view, the polling implementation is more favorable. Two main disadvantages of polling come to mind:

1. Who notifies the observer to poll its subject(s)? How do you assure that this doesn't happen too early/too late. If the subject is polled too early, the observer may update on an old or illegal state. If the subject is polled too late, transitions may be lost.

2. Because the subject is polled even if it doesn't change, overhead is introduced.

Since both problems are significant, the polling option is undesirable for most situations. But sometimes it's the better option.

4.2.1 GUI frameworks

Let's say your observer is some part of a graphical user interface (GUI). A GUI will have something called either an "event loop" or a "main loop". This loop will receive and dispatch events form various sources, for example the mouse, the keyboard, etc.

An object can receive control periodically by registering itself at the event loop (you can use the Observer interface for this). After each processed event, the event loop object calls "notify" on itself, which in turn calls "update" on each observer. Each observer then queries its subject as part of its "update" implementation and recomputes its state.

Because update is called on each interested object after every user initiated event, this method is very powerful for controlling GUI elements that need to update after the user has manipulated some part of the interface.

OpenStep [7] uses this method to update menu items and other elements in GUI-based applications; for example, in its "update" implementation, the menu cell locates the object to be called when the menu is activated by using the Chain-of-Responsibility pattern [1]. This object is asked if the menu cell's operation is currently allowed. Depending on the result, the menu cell enables or disables itself.

The polling implementation of the Observer pattern is a very powerful tool within the graphical user interface domain because GUI applications frequently have the following characteristics:

1. The observer's state (the GUI's state) depends on the current state of the subject but not on it's state history. In these cases the observer doesn't need to update on every state change of the subject.

2. Because GUI elements need to update only if the GUI is reactive (can respond to a new event), the update can be deferred to the end of the current event. Most of the time, this deferral is actually desirable.

Think of a listbox that is automatically redisplayed every time an item is added or removed. In the case of automatic redisplay, adding a sequence of items will result in redundant redisplays. Where there is no automatic redisplay, then it is the client's responsibility to trigger the screen refresh.

An alternative to these two approaches is the described polling implementation of the

Observer pattern. Here, the client of the listbox doesn't have to worry about calling any redisplay operation. The listbox can redisplay itself only once after all modifications have finished. No further modifications can take place because the current event processing has finished.

4.2.2 Reducing polling overhead

Because polling introduces a fixed overhead for each event, the programmer has to assure that query operations are cheap.

If query operations are too expensive, you can make them cheaper by maintaining some additional state in the subject that signals relevant changes.

For example, you can increment a counter in every state changing method. The observer can query, store and compare this counter during updates. If, for example, the observer needs to call four methods on the subject to update itself, you save three method calls if the subject remains unchanged between two update notifications.

The disadvantage of this approach is that you have to modify/subclass the subject in order to create the counter mechanism.

4.3 Polling on demand

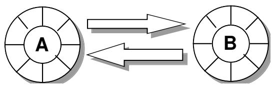
In a different implementation approach, the observer queries the subject whenever the observer itself is queried. If the observer is completely encapsulated, no one knows whether the observer is synchronized or not. This would assure consistency but may increase the overhead if the observer is queried much more often than the subject changes state.

In it's most extreme form, the observer would not keep any dependent state at all but would query the subject every time data is needed. In this case, you don't need the Observer pattern in the first place, because there are no state dependencies :-)

5 The Complex Case

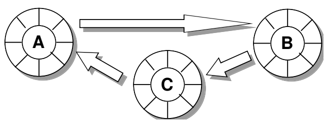
If you have an observer that is itself a subject or if observers influence each other, you have what is called the complex case.

The following pictures use circles to denote objects and directed "is-state-dependent-on" arcs to denote a state dependency between two objects.



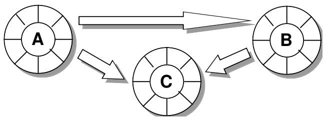
If, as in the above picture , "A" is state dependent on "B" and "B" is state dependent on "A", there is a cycle in the dependency graph. In this case, even the Observer pattern is not helpful. The application might hang, since "A" and "B" might call each other's "update" methods recursively. Even if the update cascade can be stopped, resources can be wasted due to redundant updates.

Moving control to a third object "C" which is responsible for preserving consistency between "A" and "B" will solve this problem, but now "A" and/or "B" might not be reusable without "C" or without detailed knowledge of how to preserve consistency between both objects.



Cycles covering more than two objects, as in the above picture, are even harder to find.

Even if the original design doesn't contain such cycles, they are frequently introduced in the maintainance phase, especially if the implementer changes or the software is poorly documented.



In the above picture, object "A" depends on object "B" and both "A" and "B" depend on "C". There is an undirected cycle in dependency graph. Depending on the update strategy, this could result in incorrect results and/or redundant updates as well.

For example, if "C" changes and "A" is notified first and the subject "A" changes its state as part of the update implementation, "B" will receive an update call twice.

In the implementation section of the Observer pattern [1], a ChangeManager is introduced. A ChangeManager is a singleton were "is-state-dependent-on" pathes can be registered and were updates are triggered. This "central" object can indeed handle undirected cycles because it can overlook the dependency graph and trigger the updates in a sequence that results in a minimum number of updates. If a change scope is introduced, the ChangeManager can further optimize by notifying observers with multiple subjects only once.

In section 4, I mentioned that calling "update" on an observer should not alter the subject's state because otherwise different observers will see different subject states. Unfortunately, an update invocation may not be "side-effect free" if there is a cycle in the subject/observer graph.

But even if there is no such cycle, side-effects may result since some other object may change the subject during an "update" call. A strictly cycle free association graph helps in avoiding these situations. Unfortunately such a strictly hierarchical design tends to concentrate behavior in the "upper" part of the class graph leaving only data in the leaves.

I can't offer a general solution to the problem of cycles (directed or undirected) in the dependency graph. In my experience, "sophisticated" ChangeManagers are not often useful, because they can't be used in many situations and better and cheaper specialized solutions can frequently be found. (If some readers have had satisfactory experience with ChangeManagers, I would like to hear about it and I would include the responses in an appendix.)

6 State-Dependencies and Design

Every larger OO software system has state dependencies between different objects. Sometimes these dependencies are introduced to enhance certain software qualities like reusability, sometimes they are introduced accidentally or during the maintainance phase.

Because of this, the Observer pattern and its different implementation forms can help to enhance the correctness and quality of the software product. But like every powerful technique, there's a certain risk of misuse:

- If your classes are closely related abstractions which are always reused together, there's no need to decouple them using the Observer pattern.
- The Observer pattern introduces an additional level of indirection and blurs state dependencies between objects. This increases flexibility but decreases the understandability and performance of the code.

References

[1] *E. Gamma et al*: Design Patterns, Elements of reusable Object-Oriented Software, Addison-Wesley, 1995

[2] *A. J. Riel*: Object-Oriented Design Heuristics, Addison-Wesley, 1996

[3] *G. E. Krasner et al*: A cookbook for using the model view controller user interface paradigm in Smalltalk-80, JOOP, Aug/Sep 1988

[4] *R. Martin*: The Dependency Inversion Principle, Engineering Notebook, C++ Report, May 1996. Also available under *http://www.oma.com/PDF/dip.pdf*

[5] *B. Meyer*: Object-ortiented Software Construction, Prentice Hall, 1988

[6] *B. Milewski*: Resource management in C++, JOOP, March/April 1997

[7] Openstep Appkit Reference. Also available under *http://devworld.apple.com/dev/Rhapsody/OPENSTEP/ApplicationKit/Welcome.html*