Concise paper

The Observer Pattern Revisited

Andrew Eales
Wellington Institute of Technology
Wellington, New Zealand
andrew.eales@weltec.ac.nz

ABSTRACT

Software design patterns describe elegant solutions to commonly encountered software design problems. The observer pattern is a behavioural object-oriented design pattern first codified by a group of IBM researchers (Gamma, 1995). Observers become listeners by registering their interest with observables that notify all registered listeners when an update is required. Java implementations of the observer pattern may create undesirable run-time behaviour characterised by observers performing multiple updates, cyclic dependencies between objects, and dangling references that inhibit garbage collection. This paper examines the pitfalls and design tradeoffs encountered when implementing the observer pattern, and considers the deficiencies inherent in the native Java support for the pattern. A class that manages the lifecycle of observer and observable instances is introduced. This class recursively updates registered observers while detecting cycles and ensuring that memory leaks do not occur.

Keywords:
Observer, object-oriented, Java, pattern, software design

1. INTRODUCTION

The observer pattern is a behavioural object-oriented design pattern first codified by a group of IBM researchers (Gamma, 1995) that is also known as the broadcaster-listener or publisher-subscriber pattern. This object-oriented design scheme ensures that an observer object or a set of observer objects automatically perform appropriate actions when required to do so by an observable object. Observers register their interest with one or more observable objects and are notified when an event that satisfies this interest is recognised by an observable. The observer pattern also promotes a fundamental object-oriented design heuristic (Riel, 1996) by facilitating loose coupling between communicating objects. Unfortunately, the naive implementation of the basic pattern is not robust enough to cope with complex situations created by hierarchies of observers and observables such as the relationships found in the spatial dependencies between symbols in the Music Notation Toolkit (Eales, 2000). This paper examines the pitfalls and design tradeoffs encountered when implementing the observer pattern, and considers the deficiencies inherent in the native Java support for the pattern. The design of an observer manager class that provides an elegant solution to these problems is discussed.

2. JAVA IMPLEMENTATION

The Java programming language directly supports the observer pattern by providing a concrete Observable class and an Observer interface in package java.util. This implementation promotes a loosely coupled relationship where observers are implemented as an interface and observables must inherit from a base class. Thus observables are coupled to an interface rather than directly to observer instances. Different implementation schemes to the native Java implementation exist in the literature, some of which are discussed in the following sections.

2.1 Structural Implementation strategies

Implementing classes as either derived classes or interfaces creates four possible structural class relationships (Coad, 1998). Observer and observable can both be implemented as base

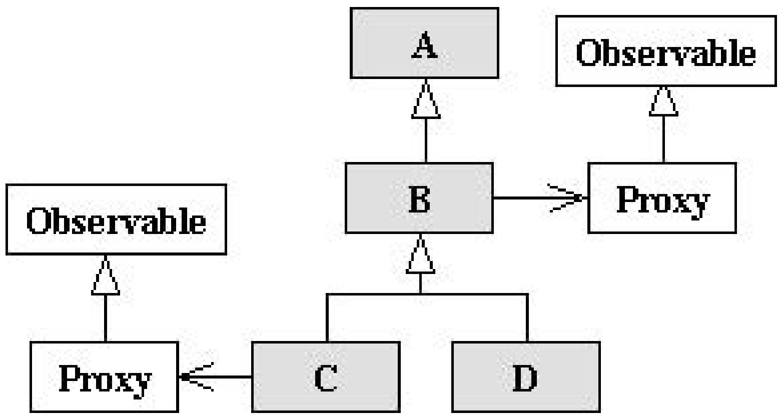

Figure 1. Observables implemented using proxy classes.

classes, interfaces, or, by mixing the two implementations so that one is a class and the other is an interface. The native Java implementation has been criticised by Coad (Coad, 1999) and Venners (Venners, 1998). Forcing Observables to inherit from a class in a single inheritance language can disrupt the inheritance hierarchy within the problem domain and, more seriously, distort the semantics of inheritance. Classes derived from class Observable in package java.util are not specializations of class Observable which functions solely as a utility class. If direct inheritance is to be avoided, observables must be implemented as an interface or by means of instance relationships (composition). An observable must have methods to manage its relationship with observers; in particular, it must be able to add and remove observers from a collection of dependent observers. Interfaces, by definition cannot be used to implement observables, as an interface may not store a collection of objects. Classes that function as observables must implement observable behaviour via composition (Coad, 1999) that delegates responsibility to a separate class. Figure one illustrates the use of proxy classes that make problem domain classes B and C observables.

This class hierarchy preserves the integrity of the problem domain in a single inheritance language; unfortunately adding to the complexity of the design. Modifying a class hierarchy can modify the implementation of structural patterns such as the composite pattern (Gamma, 1995) to fit the needs of a particular application. Attribute propagation and method visibility can be altered by carefully considering the design of the inheritance hierarchy. Behavioural patterns, such as the observer pattern, have a dynamic nature that is not easily modified or controlled by altering inheritance relationships. The observer pattern is a classic example of the problems caused by a single inheritance programming language. An elegant solution requires multiple inheritance, where both observables and observers can also function as problem domain or user interface objects.

2.2 Dynamic Implementation

Order of notification, design of methods and consideration of argument types can control the dynamic behaviour of the observer pattern. The native Java implementation will not notify observers in any specific order. Notifying observers in reverse order of registration enhances system performance. The native Java implementation allows an observable to pass a reference to itself to an observer, allowing observers to obtain additional information from an observable. This creates a 'pull' model where observers can obtain information via a reference from the event generator after the event. Complex hierarchies of relationships where an observer is also an observable cannot be expressed using the native Java implementation, since the two provided types are not convertible – an observer cannot be cast to an observable and vice versa. A class that inherits from class Observable and implements the Observer interface cannot dynamically discard one of its roles, which are fixed in the class design. The observer manager presented in section four preserves the spirit of the Java language that promotes type convertibility, by expressing both

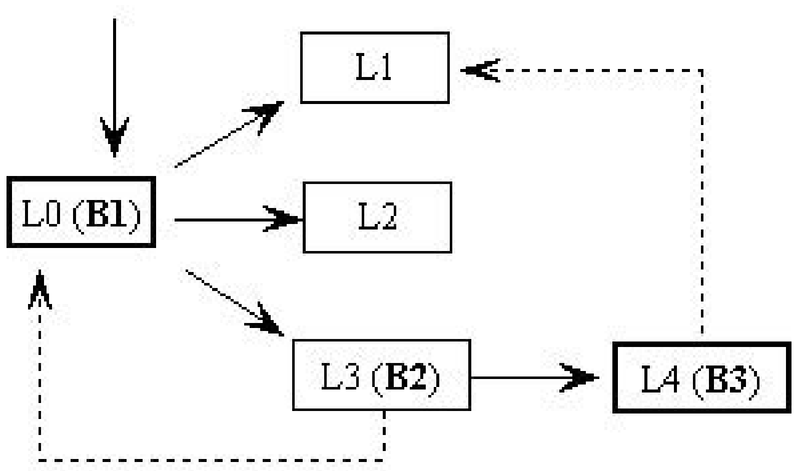

Figure 2. Cyclic dependencies and multiple observer updates.

observers and observables in terms of the Java base type Object.

3. IMPLEMENTATION PITFALLS

Observer implementations can easily result in undesirable run-time behaviour characterised by observers performing multiple updates and observer references creating dangling references that inhibit garbage collection. Complex interrelationships between observables and observers result in relationship hierarchies where an object can function as both an observer and an observable. Such an object generates the same events that it must respond to. For brevity and clarity, figure two uses the alternative denotation of broadcaster-listener for the observer pattern. Listeners one and three create a cycle by also being broadcasters. Listener one is dependent on two broadcasters which results in listener one receiving two notifications within a single update operation initiated by broadcaster one:

Although Java simplifies memory management for the programmer, it is still possible to create memory leaks via live references. Any reference to a listener that remains within a system after attempting to remove the listener will inhibit the Java garbage collector and create a memory leak. Ensuring that all references are removed where an observer may be referenced from multiple observables can be a non-trivial operation.

4. AN OBSERVER MANAGER

The proposed solution to the implementation challenges created by the observer pattern consists of a class that manages the lifecycles of observer and observable instances. Observers are added and removed to observables by the manager, which also ensures that updates are performed correctly. This manager class recursively updates registered observers while avoiding multiple updates and cyclical behaviour. The following methods add and remove objects:

```
void Register(Object anObservable,
Object anObserver)
If anObservable is registered with the
manager, add anObserver to the list
of observers for anObservable, other-
wise, register anObservable and add
anObserver to its list of observers.
void Deregister (Object anObject)  re-
moves all references to anObject from
the manager where anObject is an ob-
server, observable, or possibly both
of these.
void Deregister (Object anObservable,
Object anObserver)
Remove anObserver from the list of
observers for anObservable. If anOb-
server is the only observer, the ob-
servable object anObservable is also
removed.
```

Updates are controlled by the observer manager, which notifies all registered observers of a particular observable when an update is required. Where an observer is also an observable the method recursively processes all of its observables and their dependencies. Multiple updates and cycles are avoided by viewing the update process as a graph traversal which maintains a list of visited objects. An argument parameter and a reference to the observable implement both the push and pull models of event notification (listing 1) :

```
void notifyManager (Object anObserv-
able, Object arg)
{
    if ( anObservable s exists )
    {   for ( all observers of anOb-
servable )  // process in reverse or-
der of registration
        {  currentObserver.notify
(anObservable, arg)  // combine push
and pull models
            place the current observer
on the visited list
            if  ( currentObserver is
also an Observable )
     call notifyManager (currentOb-
server, arg)
        }
    }
 }
```

Listing 1. Updating all registered observers.

The observer manager is implemented using entries in a Java hash map consisting of an observable key and a corresponding list of observers. An iterator processes all observers for a particular observable, while the recursive call handles all cases where an observer also functions as an observable. This representation scheme allows complex dynamic relationships such as those found in the Music Notation Toolkit (Eales, 2000) to be effectively managed.

5. CONCLUSION

Observers facilitate the creation of a software architecture that allows active notification between loosely coupled objects. By forcing inheritance on observable implementations, the native Java implementation highlights the distortions created by object-oriented languages that do not support multiple inheritance. A single inheritance language is forced to achieve functionality at the expense of semantic clarity. Class Observable in java.util distorts the is-a relationship that defines inheritance. The dynamic nature of the pattern creates a number of potential problems for unwary software designers and programmers. These problems include relationships that create multiple updates or cycles, dangling references and inconvertible types when creating objects that are both observers and observables. Attempts to address these issues led to the design of an observer manager that controls the entire lifecycle of observer-observable relationships. The manager is able to elegantly address the potential pitfalls and implementation problems discussed in this paper. By providing a container that stores observable and observer instances as the Java type Object, the manager conforms to the regularity of design found in the design of the Java container classes.

REFERENCES

Coad, P. (1999) Java Design: Building Better Apps and Applets.Yourdon Press

Eales. A.A. (2000) The Music Notation Toolkit: A Study in Object-Oriented Development Proceedings of the 13th Annual conference of the NACCQ, Wellington. p99

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Addison-Wesley Professional Computing Series.

Riel.A.J. (1996) Object-Oriented Design Heuristics. Addison-Wesley

SUN Microsystems Java language API v1.4.2 http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observable.html

Venners (1998) The 'event generator' Idiom: how and when to make a Java class Observable. Java World, September 1998. Accessed April 20, 2005 http://www.javaworld.com/javaworld/jw-09-1998/jw-09-techniques.html