Patterns in C – Part 4: OBSERVER

By Adam Petersen <adampetersen75@yahoo.se>

Managing dependencies between entities in a software system is crucial to a solid design. In the previous part we had a look at the open-closed principle. This part of the series will highlight another principle for dependency management and illustrate how both of these principles may be realized in C using the OBSERVER pattern.

Dependencies arise

Returning to the examples used for the STATE pattern [2], they described techniques for implementing the behaviour of a simple digital stop-watch. Implementing such a watch typically involves the task of fetching the time from some kind of time-source. As the time-source probably will be tied to interactions with the operating system, it is a good idea to encapsulate it in an abstraction hiding the system specific parts in order to ease unit testing and portability. Similarly, the internals of the digital stop-watch should be encapsulated in a module of its own. As the digital stop-watch is responsible for fetching the time from the time-source (in order to present it on its digital display), it stays clear that there will be a dependency between the watch and the time-source. There are two obvious choices for the direction of this dependency.

Consider the Watch as a Client

It may seem rather natural to consider the watch as a client of the time-source. That is, letting the watch depend upon the time-source. Unfortunately, implementing the dependency in this direction introduces one obvious problem: how does the watch know if the time changes? The quick answer is: it doesn't. At least it doesn't unless it introduces some kind of polling mechanism towards the time-source. Just as important as it is to avoid premature optimization, one should also strive to avoid premature pessimization; even if the direction of the dependency seems correct, this solution is very likely to be extremely CPU consuming. In case the application needs to do more than updating a display, the problem calls for another solution.

Let the Time-Source update the Watch

The potential capacity problem described above may easily be avoided by reversing the dependency. The time-source may simply notify the watch as soon as its time changes. This approach introduces a dependency from the time-source upon the watch.

Listing 1: Code for the time-source

```
#include "DigitalStopWatch.h"
#include "SystemTime.h"

static DigitalStopWatchPtr digitalWatch;
static SystemTime currentTime;

/* Invoked once by the application at start-up. */
void startTimeSource()
{
   digitalWatch = createWatch();

   /* Code for setting up handlers for interrupts, or the like, in order to get notified each
millisecond from the operating system. */
}

/* This function is invoked each millisecond through an interaction with the operating system. */
static void msTick()
{
   /* Invoke a function encapsulating the knowledge about time representation. */
   currentTime = calculateNewTime();

   /* Inform the watch that another millisecond passed. */
   notifyChangedTime(digitalWatch, &currentTime);
}
```

The attractiveness of this approach lies in its simplicity. However, if scalability and flexibility are desired properties of the solution, the trade-offs are unacceptable. The potential problems introduced are best described in terms of the principles that this design violates.

The Open-Closed Principle

Having a quick recap on the open-closed principle, it is summarized as *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"* [4]. The code for the time-source above clearly violates this principle. In its current shape it supports only a single watch of one type. Imagine supporting other types of watches, for example one with an analogue display. The code for the time-source would, due to its hard-coded notification, explode with dependencies in all directions on all types of watches.

The Stable Dependencies Principle

During software maintenance or incremental development changes to existing code are normally unavoidable; even when applying the open-closed principle the design is just closed against certain modifications based upon assumptions by the original designer (it is virtually impossible for a software entity to be completely closed against all kinds of changes). The stable dependencies principle tries to isolate the impact of changes to existing code by making the recommendation that software entities should "depend in the direction of stability" [4].

In the initial approach, the watch itself fetched the time from the time-source. With respect to the stable dependencies principle, this was a better choice. A time-source is typically a highly cohesive unit and the more stable entity; simply encapsulating it in a proper abstraction, which hides the system specific details, makes it a good candidate to be packaged in a re-usable lower-level domain layer.
By having the time-source depend upon the higher-level digital watch, we violate the stable dependencies principle. This violation manifests itself by making the code for the watch difficult to update. Changes may actually have impact upon the code of the time-source itself!

Combining the dependency direction of the first approach with the notification mechanism of the second would make up an ideal design and the design pattern OBSERVER provides the extra level of indirection necessary to achieve the benefits of both solutions without suffering from any of their drawbacks.

OBSERVER

The OBSERVER pattern may serve as a tool for making a design follow the open-closed principle. *Design Patterns* [3] captures the intent of OBSERVER as "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically". Applying the OBSERVER pattern to our example, extended with a second type of watch, the "dependents" are the different watches. *Design Patterns* [3] defines the role played by **`TimeSource`** as a "concrete subject", the object whose state changes should trigger a notification and update of the dependents, i.e. observers.

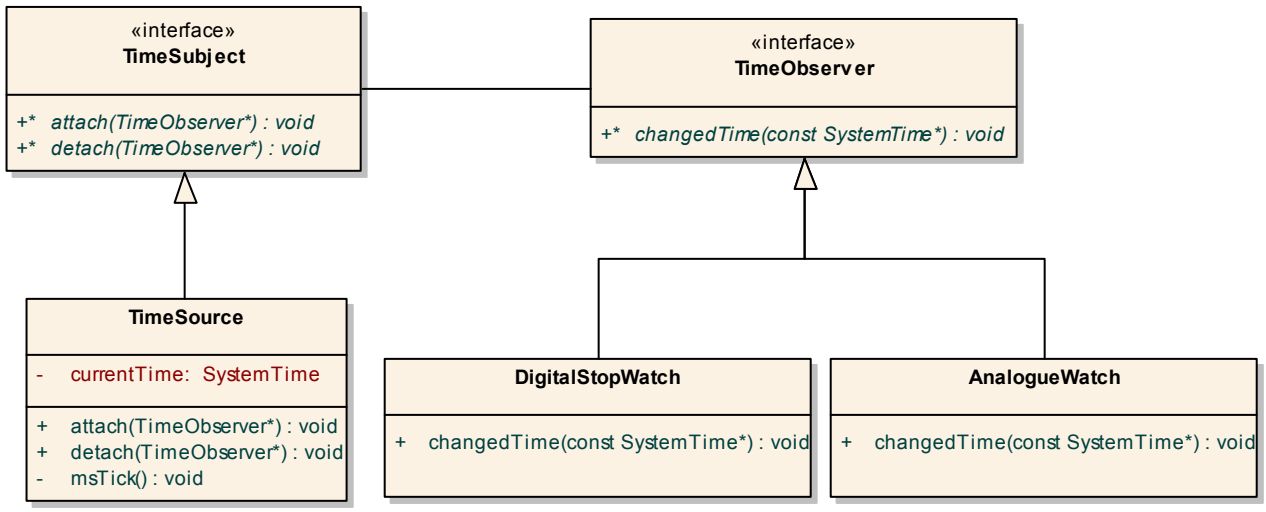

Figure 1: OBSERVER pattern structure

Implemenation Mechanism

In order to decouple the subject from its concrete observers and still enable the subject to notify them, each observer must correspond to a unique instance. The FIRST-CLASS ADT pattern [1] provides a way to realize this. However, as seen from the subject, all observers must be abstracted as one, general type sharing a common interface. In the C language, without language support for inheritance, generality is usually spelled **`void*`**; any pointer to an object may be converted to **`void*`** and back to its original type again without any loss of information. This rule makes a common interface possible.

Listing 2: Interface of the observers, `TimeObserver.h`

```
typedef void (*ChangeTimeNotification)(void* instance, const SystemTime* newTime);

typedef struct
{
   void* instance;
   ChangeTimeNotification notification;
} TimeObserver;
```

The interface above declares a pointer to a function, which proves to be an efficient technique for implementing dynamic behaviour; a concrete observer attaches itself, together with a pointer to a function, at the subject. As the subject changes, it notifies the observer through the attached function.

By using **`void*`** as abstraction, type-safety is basically traded for flexibility; the responsibility for the conversion of the observer instance back to its original type lies on the programmer. A conversion back to another type than the original may have disastrous consequences. Franz Kafka, although not very experienced in C programming, provided a good example of such behaviour: "When Gregor Samsa woke up one morning from unsettling dreams, he found himself changed in his bed into a monstrous vermin" [5]. Obviously, Gregor Samsa wasn't type-safe. In order to guard against such erroneous conversions, the **`TimeObserver`** structure has been introduced to maintain the binding between the observer and the function pointer. The implicit type conversion is encapsulated within the observer itself, as illustrated in the code below.

Listing 3: A concrete observer implemented as a FIRST-CLASS ADT [1]

```
/* Include files omitted. */
struct DigitalStopWatch
{
   Display watchDisplay;
   /* Other attributes of the watch, e.g. digital display. */
};

/* Implementation of the function required by the TimeObserver interface. */
static void changedTime(void* instance, const SystemTime* newTime)
{
   DigitalStopWatchPtr digitalWatch = instance;
   assert(NULL != digitalWatch);

   updateDisplay(digitalWatch, newTime);
}
```

Before an observer can get any notifications, it has to register for them. Listing 4 declares the interface required by the observers.

Listing 4: Interface to the subject, `TimeSubject.h`

```
#include "TimeObserver.h"

void attach(const TimeObserver* observer);
void detach(const TimeObserver* observer);
```

These functions are implemented by the concrete subject, the **`TimeSource`**, which has to keep track of all attached observers. The example below uses a linked-list for that task. Attaching an observer corresponds to adding a copy of the given **`TimeObserver`** representation to the list. Similarly, upon a call to **`detach`**, the node in the list corresponding to the given **`TimeObserver`** shall be removed and that observer will no longer receive any notifications from the subject. The code fragments below illustrate this mechanism.

Listing 5: Implementation of the subject, `TimeSource.c`

```
#include “TimeSubject.h”

struct ListNode
{
   TimeObserver item;
   struct ListNode* next;
};

static struct ListNode observers;
static SystemTime currentTime;

/* Local helper functions for managing the linked-list (implementation omitted). */

static void appendToList(const TimeObserver* observer)
{
   /* Append a copy of the observer to the linked-list. */
}

static void removeFromList(const TimeObserver* observer)
{
   /* Identify the observer in the linked-list and remove that node. */
}

/* Implementation of the TimeSubject interface. */

void attach(const TimeObserver* observer)
{
   assert(NULL != observer);
   appendToList(observer);
}

void detach(const TimeObserver* observer)
{
   assert(NULL != observer);
   removeFromList(observer);
}

/* Implementation of the original responsibility of the TimeSource (code for initialization, etc
omitted). */
static void msTick()
{
   struct ListNode* node = observers.next;
   /* Invoke a function encapsulating the knowledge about time representation. */
   currentTime = calculateNewTime();

   /* Walk through the linked-list and notify every observer that another millisecond passed. */
   while(NULL != node) {
      TimeObserver* observer = &node->item;
      observer->notification(observer->instance, &currentTime);
      node = node->next;
   }
}
```

The code above solves our initial problems; new types of watches may be added without any modification to the **`TimeSource`**, yet these watches, our observers, are updated in an efficient way without the need for expensive polling.

Observer Registration

An additional benefit of implementing the OBSERVER pattern as a FIRST-CLASS ADT [1] is the combination of loose dependencies with information hiding. The client neither knows nor depends upon a subject. In fact, the client doesn't even know that the **`DigitalStopWatch`** acts as an observer because the functions for creating and destructing the ADT encapsulate the registration handling. The code below, which extends Listing 3, illustrates this technique:

Listing 6: Encapsulation of the registration handling

```
static void changedTime(void* instance, const SystemTime* newTime)
{
   /* Implementation as before (Listing 3). */
}

DigitalStopWatchPtr createDigitalWatch(void)
{
   DigitalStopWatchPtr watch = malloc(sizeof *watch);

   if(NULL != watch){
      /* Successfully created -> attach to the subject. */
      TimeObserver observer = {0};
      observer.instance = watch;
      observer.notification = changedTime;

      attach(&observer);
   }
   return watch;
}

void destroyDigitalWatch(DigitalStopWatchPtr watch)
{
   /* Before deleting the instance we have to detach from the subject. */
   TimeObserver observer = {0};
   observer.instance = watch;
   observer.notification = changedTime;

   detach(&observer);
   free(watch);
}
```

Subject – Observer Dependencies

The introduction of the OBSERVER pattern results in loose dependencies between the subject and its observers. However, no matter how loose, the dependencies cannot be ignored and an often overseen aspect of the OBSERVER pattern is to ensure correct behaviour in case of changed registrations during a notification of the observers. The problem should be addressed in all OBSERVER implementations and is illustrated by investigating the notification loop coded earlier:

Listing 7: Code extracted from Listing 5

```
    /* Walk through the linked-list and notify every observer that another millisecond passed. */
    while(NULL != node) {
       TimeObserver* observer = &node->item;
       observer->notification(observer->instance, &currentTime);
       node = node->next;
    }
```

In case an observer decides to detach itself during a notification, the list containing the nodes may become corrupted. The solutions span between forbidding subject changes during notification and, at the other extreme, allow the subject to change and ensure it works.

The solution with forbidding subject changes during notification calls for a well-documented Subject interface. Further, the constraint may be checked at run-time using assertions.

Listing 8: Checking Subject constraints with assertions

```
static int isNotifying = 0;

void attach(const TimeObserver* observer)
{
   assert(0 == isNotifying);
   /* Code as before. */
}

void detach(const TimeObserver* observer)
{
   assert(0 == isNotifying);
   /* Code as before. */
}

static void msTick()
{
   struct ListNode* node = observers.next;

   /* Ensure that no changes are done to the subject during notification. */
   isNotifying = 1;

   while(NULL != node) {
      /* Loop thorugh the observers as before. */
   }
   /* All observers notified, allow changes again. */
   isNotifying = 0;
}
```

The solution at the other side of the spectrum depends upon the actual data structure used to store the observers. The idea is to keep a pointer to the next observer to notify at file-scope. Attaching or detaching an observer now involves the possible adjustment of that pointer. By exclusively using this pointer in the notification-loop, the problem with unregistrations is solved. By choosing a strategy for new registrations, defining if the new observers are to be notified during the loop where they attach or not, the solution is complete. The extra complexity is rewarded with the flexibility of allowing observers to be added or removed during notification.

One pattern, two models

In the example above, the part of the subject that changed (in our example the system time) was given to the observers as an argument to the notification-function. This technique is known as the push-model. The advantage of this model is its simplicity and efficiency; the data that changed is immediately available to the observers in the notification. A potential problem is the logical coupling between a subject and its observers; in order to deliver the correct data, the subject has to know about the needs of its observers.

This potential problem is eliminated by the other model used in OBSERVER implementations. This model, known as the pull-model, does not send any detailed information about what changed at the subject; the observers have to fetch that data themselves from the subject. In case the subject contains several large data structures, the efficiency of the pull-model may be improved by introducing an update protocol. Such a protocol specifies what changed while still putting the responsibility on the observers to fetch the actual data. A simple **`enum`** may serve well as an update protocol.

Consequences

The main consequences of applying the OBSERVER pattern are:

1. *Introduces loose dependencies*. As the subject only knows its observers through the Observer interface, the code conforms to the open-closed principle; by avoiding hard-coded notifications, any number and any types of observers may be introduced as long as they support the Observer interface. New behaviour, in the form of new types of observers, is added without modifying existing code.
   Further, the loose dependencies provide a way to communicate between layers in a sub-system. *Design Patterns* [3] recognizes this potential: "Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact." This property may serve as a tool for minimizing the impact of modifications by following the stable dependencies principle and yet enable a bidirectional communication between layers.

2. *Potentially complex management of object lifetimes*. As illustrated above, the FIRST-CLASS ADT pattern [1] simplifies the management by encapsulating the interaction with the subject. However, in case the subject is also implemented as a first-class object, the dependencies have to be resolved on a higher level and the client has to ensure that the subject exists as long as there are observers attached to it.

3. *May complicate a design with cascades of notifications*. In case an observer plays a second role as subject for other observers, a notification may result in further updates of these observers leading to overly complex interactions. Another problem may arise if an observer, during its update, modifies the subject resulting in a new cascade of notifications.

4. *Lowers the cohesion of the subject*. Besides serving its central purpose (in our example being a time-source) a subject also takes on the responsibility of managing and notifying observers. By merging two responsibilities in one module, the complexity of the subject is increased. This extra complexity is acceptable in case the loose dependencies, gained by introducing the OBSERVER pattern, provide significant benefits. Further, the subject may be simplified in cases where, during the life of the subject, there isn't any need to detach observers. In such a case, the **`detach`** function is simply omitted.

5. *Trades type-safety for flexibility.* The gist of the OBSERVER pattern is that the subject should be able to notify its dependents without making any assumptions about who they are. I.e. it must be possible to have observers of different types. The solution in this article uses **`void*`** as the common abstraction of an observer. The potential problem arises as the subject notifies its observers and passes them as **`void*`** to the notification functions. When converting a void-pointer back to a pointer of a concrete observer type, the compiler doesn't have any way to detect an erroneous conversion. This problem may be prevented by defining a unique notification function for each different type of observer in combination with using a binding such as the **`TimeObserver`** structure introduced above.

Summary

This article illustrated how the loose coupling introduced by the OBSERVER pattern may serve as a way to implement modules following the open-closed principle. We can add new observers without modifying the subject. In fact, observers may even be replaced at runtime.

Further, as OBSERVER reverses the dependencies it allows lower-level modules to notify modules at a higher abstraction level without compromising layering. This property makes it possible to implement modules following the stable dependencies principle in cases where the lower layers in a system are the more stable and yet need to communicate with the higher layers.

However, when hard-coded notifications are enough, by all means stick to them; in case the loose coupling and extra level of indirection isn't needed, the OBSERVER pattern may just overcomplicate a design.

Next time

We'll climb one step in the pattern categories and investigate the architectural pattern REACTOR. A REACTOR is useful in event-driven applications to demultiplex and dispatch events from potentially many clients.

References

1. Adam Petersen, "Patterns in C, part 1", C Vu 17.1
2. Adam Petersen, "Patterns in C, part 2: STATE", C Vu 17.2
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, "Design Patterns", Addison-Wesley
4. Robert C. Martin, "Agile Software Development", Prentice Hall
5. Franz Kafka, "The Metamorphosis"

Acknowledgements

Many thanks to Tord Andersson, Drago Krznaric and André Saitzkoff for their feedback.