Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 14: Observer Design Pattern

Contents

14.1 Introduction

This lecture note discuss the observer design pattern. The pattern defines a one-to-many relationship between subjects and their observers. A single subject may have many observers. When the subject changes state all the observers are notified and updated automatically. This pattern would work well in a multithreaded environment which is beyond the scope of this lecture note.

14.2 Observer Pattern

14.2.1 Identification

Name	Classification	Strategy
Observer	Behavioural	Delegation (Object)
Intent		
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. ([1]:293)		

14.2.2 Structure

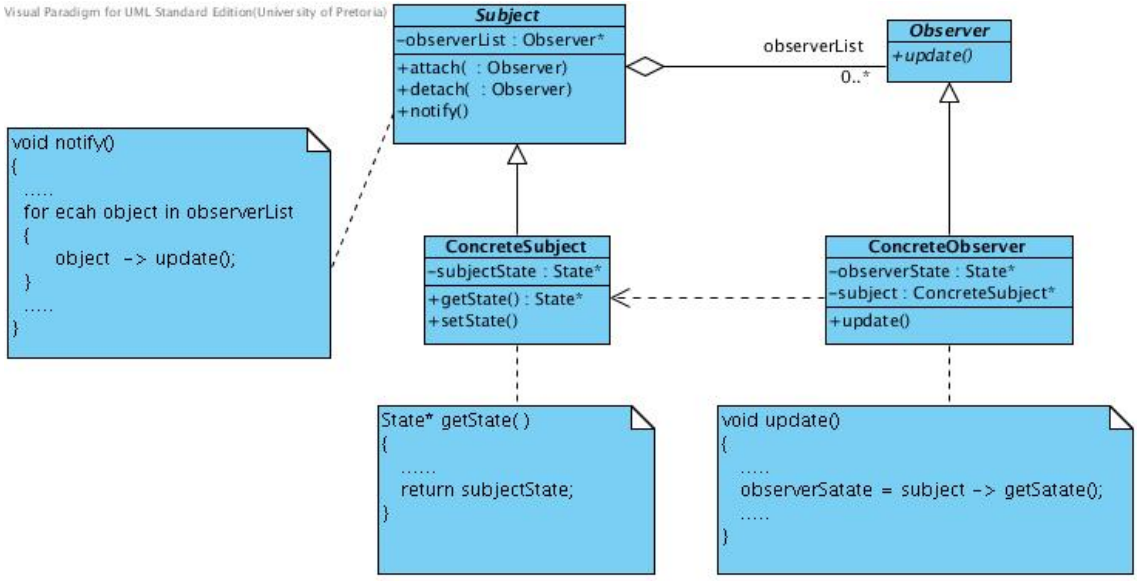

Figure 1: The structure of the Observer Pattern

14.2.3 Problem

The observer design pattern solves the problem of needing to change the code for every observer that is related to the subject during maintainance when observers are changed, added or removed from the system. The pattern facilitates the attaching and detaching of the observers from the subject leaving the subjects code intact.

14.2.4 Participants

Subject

- Provides an interface for observers to attach and detach to the concrete subject.

ConcreteSubject

- Implementation of the subject being observed.
- Implements the functionality to store objects that are observing it and sends update notifications to these objects.

Observer

- Defines the interface of objects that may observe the subject.
- Provides the means by which the observers are notified regarding change to the subject.

ConcreteObserver

- Maintains a reference to the subject it observes.
- Updates and stores relevant state information of the subject in order to keep consistent with the state of the subject.

14.3 Observer Explained

The observer design pattern offers a mechanism by which observers of a subject register with the subject and will be notified when a change occurs in the subject. Figure 2 shows how the communication between the observers and the subject takes place.

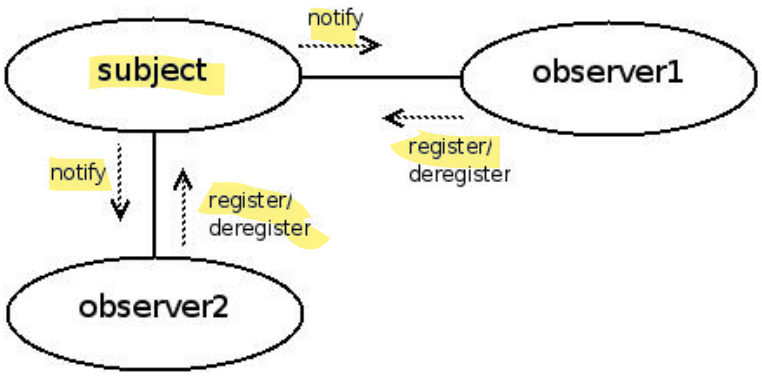

Figure 2: Overview of the interaction between the subject and its observers

An observer will register with the subject. When the subject changes it will notify all the observers registered with it. When an object no longer is an observer of the subject it will deregister from the subject.

14.3.1 Clarification

The pattern comprises of two hierarchies, the objects that are to be observed are represented by the `Subject` hierarchy and the objects that are to do the observation in the `Observer` hierarchy. It is the way in which these two hierarchies interact that brings about the power of the observer pattern. The sequence diagram in Figure 3 shows this interaction.

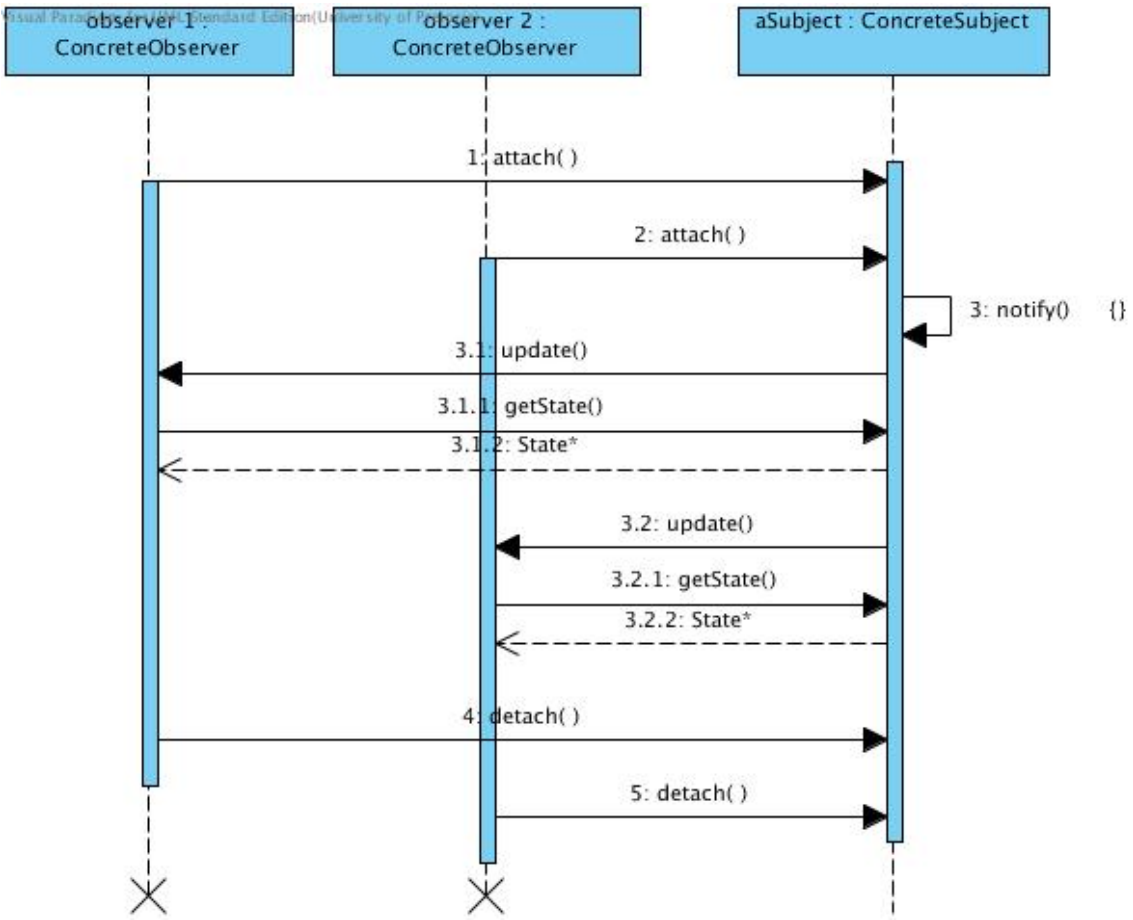

Figure 3: Sequence diagram of the interaction between the participants of the observer

14.3.2 Code improvements achieved

The most significant achievement of the Observer pattern is the separation of concerns. The observers are not embedded, but are independent of the subjects they observe and therefore may register and deregister as the need may arise.

14.3.3 Implementation Issues

The are two design issues that need to be considered when implementing the Observer Design pattern. The first concerns the detaching of the observer from the subject when it goes out of scope. The second issue concerns the how state is transferred to the observer from the subject. These two issues will be briefly addressed in the sections that follow.

Detaching from the subject

The designer of the ConcreteObserver class must ensure that when the related object goes out of scope it detaches from the subject object it is observing. This means that the destructor defined in the ConcreteObserver class needs to call the relevant detach function as defined in the Subject hierarchy. A further important design decision concerns possible extension of the Observer hierarchy, particularly in C++. In the case where the ConcreteObserver is extended to provide additional functionality, the destructor of the ConcreteObserver must be defined as virtual to ensure that the observer object calls the parent destructor and the detachment of the observer actually takes place.

Transfer of state to the observer

The state of the subject can be transferred to the observer by following either a push or a pull model [2]. With the push model, the subject takes responsibility by packaging its public state and providing the observers with this when issuing an update function. The pull model on the other hand requires the subject to provide a public interface by which the observer can request specific state information from the subject. The model depicted in the structure of the design pattern in Figure 1 shows the pull model.

Each of the models have their respective disadvantages. The efficiency of the pull model is lower because it may require a string of function calls to the subject to acquire all the required information. On the other hand, the flexibility of the push model is lower due to the subject needing to keep an additional register of the requirements of the observers and thereby increasing the coupling between the hierarchies.

14.3.4 Common Misconceptions

The observer is used to broadcast events. These events that take place in the subject are only broadcast to the observers that have registered with the subject.

14.3.5 Related Patterns

Mediator

The mediator defines how objects interact and thereby promotes loose coupling between the objects. The objects can therefore interact independently. Mediators are often used to ensure the independent transfer of state between the subject and its observers.

Singleton

By making the subject a singleton, it ensures that the subject has only one access point to it.

14.4 Example

Consider the F1 Grand Prix, or any other motor racing scenario where a pit crew needs to make decisions regarding refueling and changing the tyres of the car that is racing throughout the race. The car in this case is the subject and the pit crew are the observers. Figure 4 provides an incomplete class diagram showing the relationship between the classes. The PitStop hierarchy represents the subject and the PitCrew hierarchy the observers with two concrete observers, the Refueller and the TyreChanger.

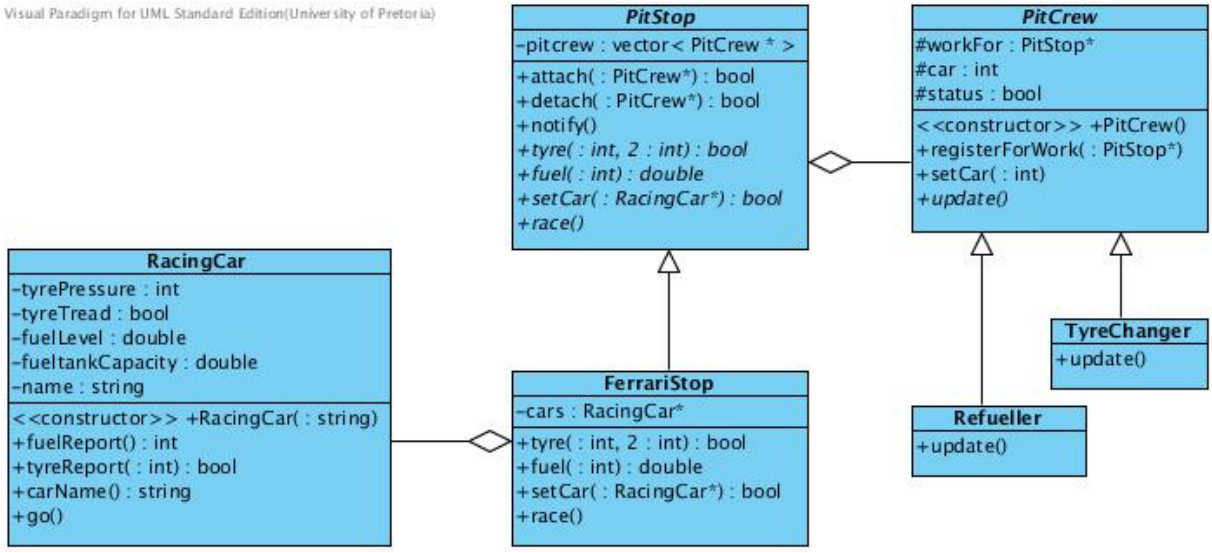

Figure 4: Racing Example

Comparison of the class diagram given in Figure 4 to that of the structure in Figure 1 will show that the dependency between the concrete observer and the concrete subject has been abstracted and included in the Observer participant of the implementation. It is the update function implementations of each of the concrete observer participants that differentiate them from each other not the dependency with the concrete subjects. This being said, a dependency relationship between the `PitCrew` class and the `PitStop` class needs to be included in the figure.

14.5 Exercises

1. Consider the following main program and draw the corresponding sequence diagram for the class diagram as given in Figure 4.

```
int main() {

  RacingCar* car[2];
  car[0] = new RacingCar("Ferrari One");
  car[1] = new RacingCar("Ferrari Two");

  PitStop* ferrariWorkshop = new FerrariStop();

  ferrariWorkshop->setCar(car[0]);
  ferrariWorkshop->setCar(car[1]);

  printWorkshopStatus(ferrariWorkshop);

  PitCrew* refueller = new Refueller();
  ferrariWorkshop->attach(refueller);

  PitCrew* tyreMech = new TyreChanger();
  ferrariWorkshop->attach(tyreMech);

  ferrariWorkshop->race();

  return 0;
}
```

2. Mediator works well with Observer. Show how you would incorporate mediator into the observer structure given in Figure 4.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Colin Moock. Introduction to design patterns, 2003. URL `http://www.moock.org/lectures/introToPatterns/`. [Online; accessed 1 September 2012].