Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 19: Adapter Design Pattern

Contents

19.1 Introduction

This Lecture Note introduces the Adapter design pattern. The adapter pattern is also referred to as a wrapper pattern which describes it intent very well. This wrapper is presented in two guises, the first adapts using delegation and the other using inheritance. Both these implementation strategies will be considered. Due to the nature of the inheritance, inheritance access specification other then public will be briefly discussed.

19.2 Adapter Design Pattern

19.2.1 Identification

Name	Classification	Strategy
Adapter	Structural	Inheritance (Class) and Delegation (Object)
Intent		
Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. ([2]:139)		

19.2.2 Problem

Used to modify existing interfaces to make it work after it has been designed.

19.2.3 Structure

The Adapter design pattern is the only pattern to which one of two structures can be applied. The pattern can either make use of delegation or inheritance to achieve its intent. The delegation structure is referred to as an Object Adapter, Figure 1, and the inheritance structure as shown in Figure 2 for the Class Adapter.

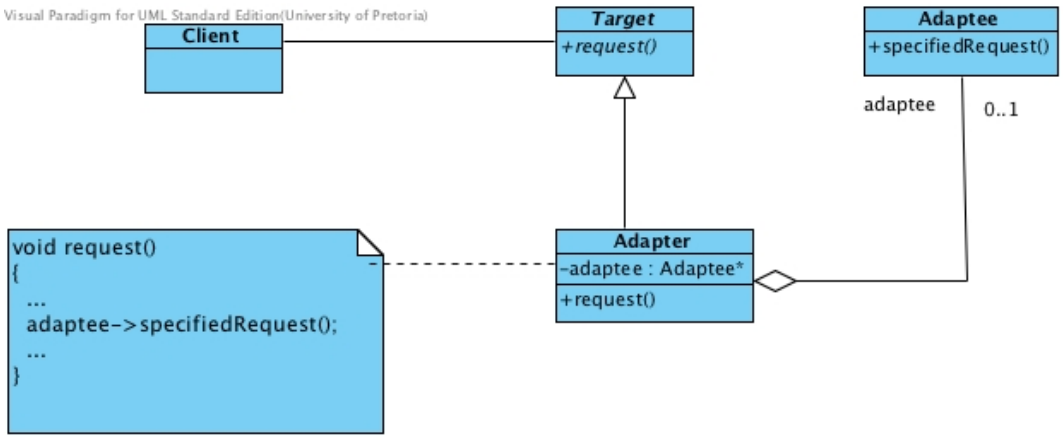

Figure 1: The structure of the Object Adapter Design Pattern

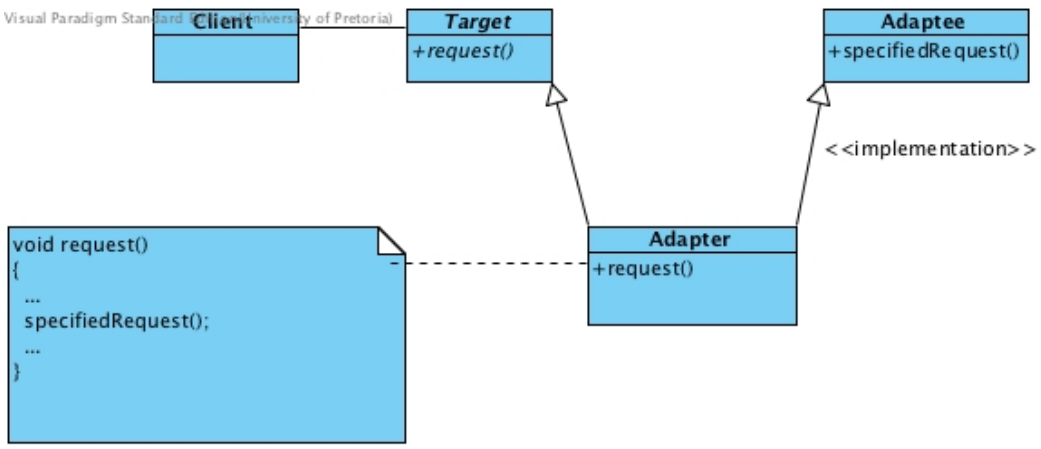

Figure 2: The structure of the Class Adapter Design Pattern

19.2.4 Participants

Adaptee

- The existing interface that needs to be adapted

Target

- Domain specific interface used by the client

Adapter

- Adapts the interface of Adaptee to the Target interface

Client

- Manipulates objects conforming to the interface specified by the abstract class Target

19.3 Protected and private inheritance in C++ explained

So far, the member access specifier (`memberAccessSpecifier` in Figure 3) for inheritance has been `public`. Two other member access specifiers for inheritance may be used, namely `protected` and `private`. Table 1 provides the visibility of the members of the base class in the derived class for each of the member access specifiers.

It can be said that a `memberAccessSpecifier` that is `private` provides the derived class with the functionality defined in the base class. Effectively, the base class is wrapped and no class that may inherit from the derived class will have access to its member functions. Private inheritance in C++ can be seen as a type of *has-a* relationship.

```
class Base {
  ...
};

class Derived : memberAccessSpecifier Base {
  ...
};
```

Figure 3: Example inheritance classes

		Inheritance access specifier of derived class		
		public	protected	private
Base member visibility	public	Derived access specifier is **public**. Derived class can access the member and so can an outside class.	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **private**. Derived class can access the member, but there is no access from an outside class.
	protected	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **private**. Derived class can access the member, but there is no access from an outside class.
	private	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.

Table 1: C++ member access specifiers and base class member visibility

19.4 Adapter Pattern Explained

19.4.1 Design

Object Adapter

Object Adapter makes use of object composition to delegate to Adaptee.

Class Adapter

Class Adapter makes use of mixin idiom [4]. A mixin is an object-orientated concept by which a class provides functionality, either to be inherited or just used, but is not explicitly instantiated. Adapter inherits and implements Target (public inheritance). Adapter inherits only the implementation, or functionality, and therefore the use of private inheritance of Adaptee resulting in a *linearisation* of the hierarchy.

19.4.2 Comparison of the approaches

When does one use delegation and when does one use private inheritance. Try to always use delegation. Use composition (inheritance) only when necessary [1].

19.4.3 Real world example

Many instances of the adapter pattern can be found in data structures where one data structure such as a list is wrapped so that it behaves as another data structure, for example a stack.

Further use of the Adapter pattern is when legacy systems are being integrated into a new system. The functionality of the legacy system is therefore encapsulated and used by the new system through an adapter.

19.4.4 Related Patterns

Bridge

Structurally they are similar. However their intent is different, the Adapter changes the interface while the Bridge separates the implementation from the interface.

Decorator

Enhances an object without changing the interface.

Proxy

Defines a surrogate of to an object without changing its interface.

19.5 Example

19.5.1 Billboard

In this example, the electronic billboard is to be adapted to in order to simplify its interface. Electronic billboards can be in one of two states, either on or off with the ability

to toggle between the states. Electronic billboards in the on state display a message, to change this message another setter method is called and then the method to display needs to be called for the message to change.

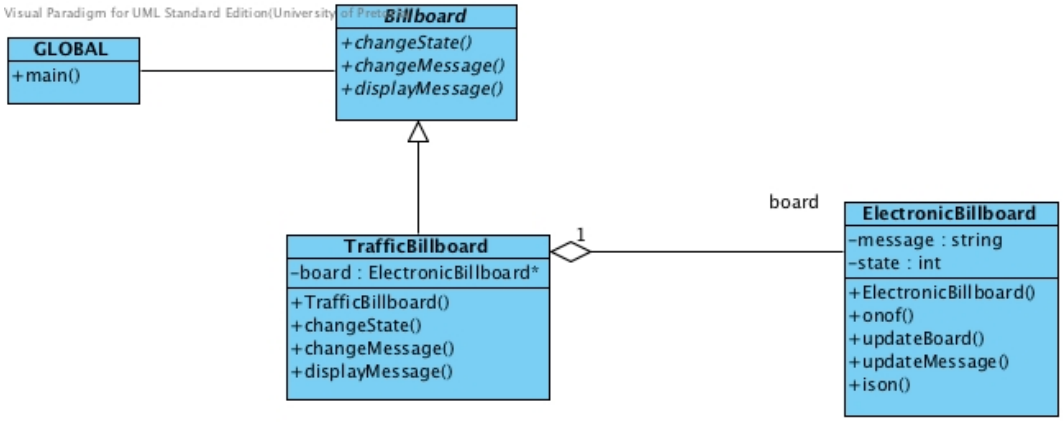

Figure 4: Billboard example - object adapter

The `Billboard` interface (Figure 4) simplifies the the electronic billboard and `TrafficBillboard` implements this simplified interface and uses the `ElectronicBillboard` functionality to do so.

Implementing the Billboard as a class adapter instead of as an object adapter is not difficult. Figure 5 shows the resulting class diagram.

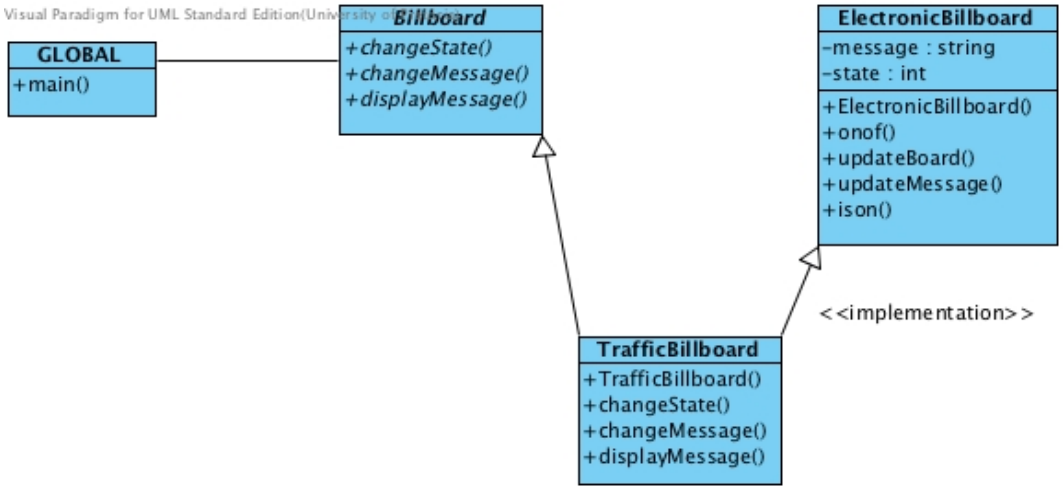

Figure 5: Billboard example - class adapter

By comparing Figures 4 and 5 it can be seen that the most significant difference between the two implementations is that the member attribute, providing the delegation functionality, is not defined in the class adapter. In order to see other subtle differences, it is necessary to look at the implementation level. Listings 1 and 2 represent the Adapter participant of the pattern for the Object Adapter and Class Adapter implementations respectively. The Object Adapter implementation instantiates an object of the Adaptee participant and access the functionality provided by the object through its public interface. The Class Adapter implementation, through private inheritance, uses and effectively

subsumes the functionality of the Adaptee participant. It is important to note, that had there been protected features in the Adaptee participant, these would have been available to the Adapter participant with the Class Adapter implementation and not with the Object Adapter implementation.

Listing 1: Object Adapter Implementation of the Billboard

```
class TrafficBillboard : public Billboard {
    public:
        TrafficBillboard() {
          board = new ElectronicBillboard("all clear");
        };
        virtual void changeState() {
          board->onof();
        };
        virtual void changeMessage(int msgId) {
          switch (msgId) {
             case 1: board->updateMessage("slow traffic ahead"); break;
             case 2: board->updateMessage("accident ahead"); break;
             default: board->updateMessage("all clear");
           }
        };
        virtual void displayMessage() {
           if (board->ison()) {
             cout << "Traffic warning: "; board->updateBoard();
             cout<<endl;
           }
           else cout<<"Board is off"<<endl;
        };
    private:
        ElectronicBillboard* board;
};
```

Listing 2: Class Adapter Implementation of the Billboard

```
class TrafficBillboard : public Billboard, private ElectronicBillboard {
    public:
        TrafficBillboard() {
          updateMessage("all clear");
        };

        virtual void changeState() {
          onof();
        };

        virtual void changeMessage(int msgId) {
          switch (msgId) {
            case 1: updateMessage("slow traffic ahead"); break;
            case 2: updateMessage("accident ahead"); break;
            default: updateMessage("all clear");
          }
```

```
        };

        virtual void displayMessage() {
          if (ison()) {
            cout << "Traffic warning: "; updateBoard();
            cout<<endl;
          }
         else cout<<"Board is off"<<endl;
        };

};
```

19.5.2 Rectangle

This example is available on the internet in different guises [3]. This is yet another adaptation along the same theme and illustrates the implementation of a class adapter. The class diagram for the rectangle is given in Figure 6 and the implementation of the Adapter participant in listing 3. `LegacyRectangle` specifies a rectangle by using 4 values, the first two values represent the x and y coordinates of the top left corner of the rectangle and the last two values the x and y coordinates of the bottom right corner of a rectangle. The adapted rectangle defines a rectangle by its top left corner coordinates and then a width and a height value towards the right and down.

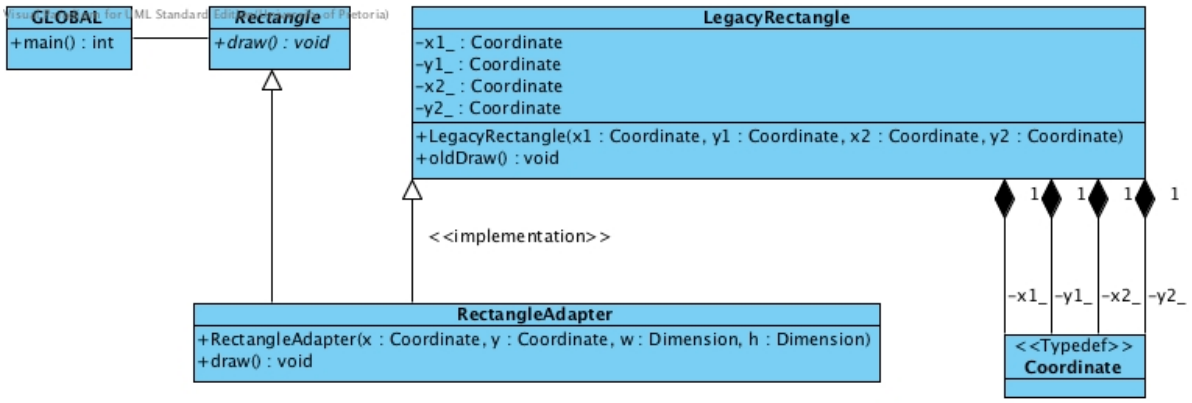

Figure 6: Rectangle example - class adapter

Listing 3: Rectangle Class Adapter Implementation

```
class RectangleAdapter : public Rectangle,
                         private LegacyRectangle {
public:
   RectangleAdapter( Coordinate x, Coordinate y,
                     Dimension w, Dimension h )
                  : LegacyRectangle( x, y, x+w, y+h ) {

    cout << "RectangleAdapter: create.  (" << x
         << "," << y
```

```
            << "), width = " << w
            << ", height = " << h << endl;
    }
    virtual void draw() {
        cout << "RectangleAdapter: draw." << endl;
        oldDraw(); }
};
```

19.6 Exercises

1. Make use of the state design pattern to encapsulate the messages displayed by the traffic billboard.

2. Rewrite the example given in Section 19.5.2 as an object adapter.

References

[1] Marshall Cline. C++ faq: Inheritance - private and protected inheritance, 1991–2011. URL `http://www.parashift.com/c++-faq-lite/private-inheritance.html`. Online; accessed 26 September 2011.

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

[3] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].

[4] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in c++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 163–177, London, UK, 2001. Springer-Verlag. ISBN 3-540-42578-0. URL `http://dl.acm.org/citation.cfm?id=645417.652070`.