Adapter

Linda Marshall

Department of Computer Science
University of Pretoria

14 and 15 October 2021

Name and Classification: Adapter (Object and Class Structural)

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. ” GoF(139)

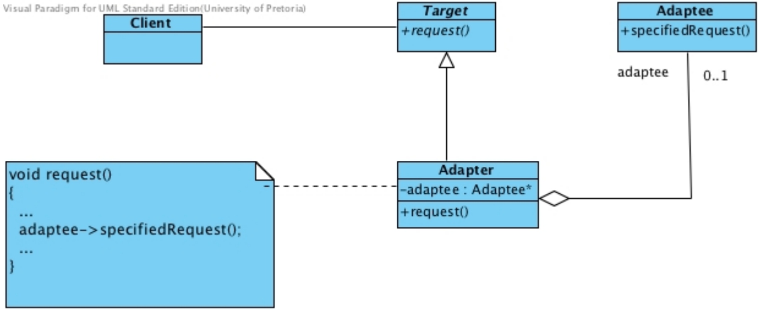
“Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces. ”

GoF(139)

There are two versions of the Adapter pattern:

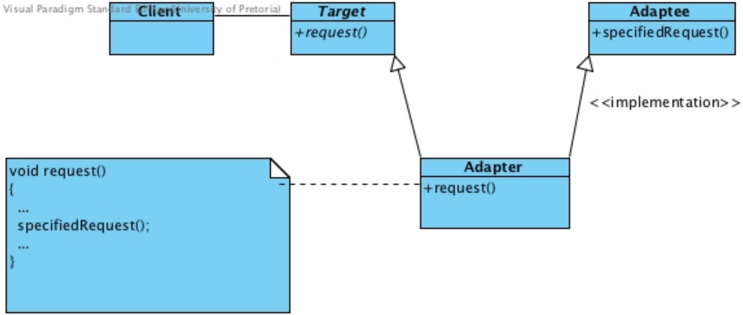
- Object Adapter - uses delegation as the mechanism to adapt an object
- Class Adapter - makes use of *private* inheritance

Object Adapter



```
void request()
{
  ...
  adaptee->specifiedRequest();
  ...
}
```

Class Adapter



```
void request()
{
  ...
  specifiedRequest();
  ...
}
```

Adaptee

- The existing interface that needs to be adapted

Client

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

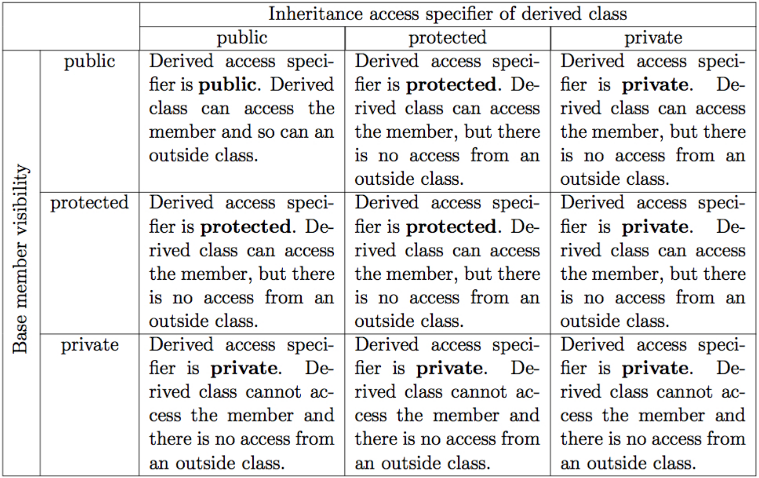
Target

- Domain specific interface used by the client

Adapter

- Adapts the interface of Adaptee to the Target interface

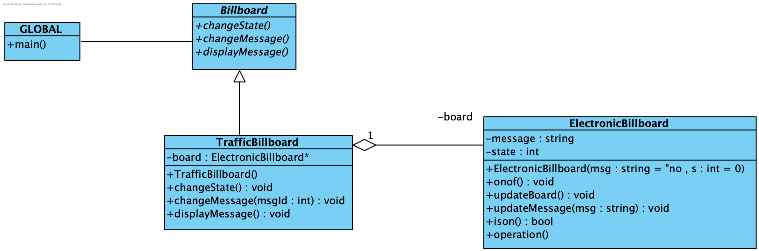
- Used to modify exsiting interfaces – make it work after it has been designed.
- **Object Adapter** makes use of object composition to delegate to the Adaptee.
- **Class Adapter** makes use of *mixins*. Adapter inherits and implements Target (public inheritance). Adapter inherits only the implementation of Adaptee (private inheritance).



		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.

- **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.

Billboard - Object Adapter



GLOBAL
+main()

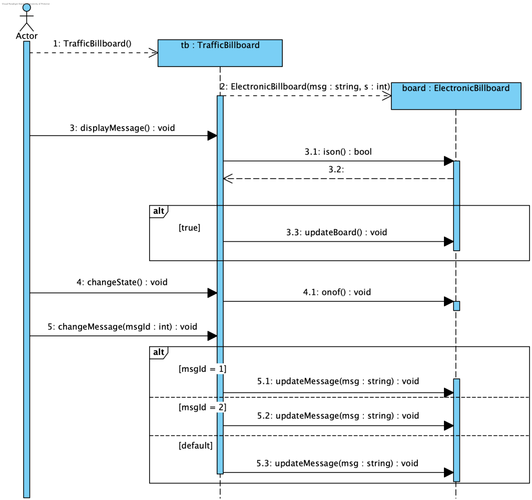
Billboard
+changeState()
+changeMessage()
+displayMessage()

TrafficBillboard
-board : ElectronicBillboard*
+TrafficBillboard()
+changeState() : void
+changeMessage(msgId : int) : void
+displayMessage() : void

ElectronicBillboard
-message : string
-state : int
+ElectronicBillboard(msg : string = "no , s : int = 0)
+onof() : void
+updateBoard() : void
+updateMessage(msg : string) : void
+ison() : bool
+operation()

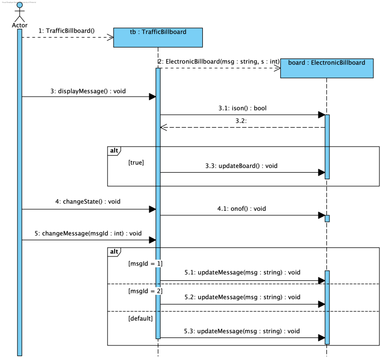
Identifying the participants

- **Adaptee** - `ElectronicBillboard`
- **Target** - `Billboard`
- **Adapter** - `TrafficBillboard`


Actor
tb : TrafficBillboard
board : ElectronicBillboard
1: TrafficBillboard()
2: ElectronicBillboard(msg : string, s : int)
3: displayMessage() : void
3.1: ison() : bool
3.2:
alt
[true]
3.3: updateBoard() : void
4: changeState() : void
4.1: onof() : void
5: changeMessage(msgId : int) : void
alt
[msgId = 1]
5.1: updateMessage(msg : string) : void
[msgId = 2]
5.2: updateMessage(msg : string) : void
[default]
5.3: updateMessage(msg : string) : void

Exercise

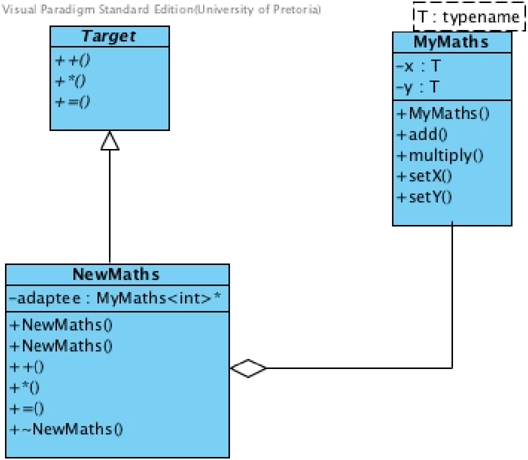
Write the main and `TrafficBillboard` class using the UML Sequence diagram.


Actor
1: TrafficBillboard()
tb : TrafficBillboard
2: ElectronicBillboard(msg : string, s : int)
board : ElectronicBillboard
3: displayMessage() : void
3.1: ison() : bool
3.2:
alt
[true]
3.3: updateBoard() : void
4: changeState() : void
4.1: onof() : void
5: changeMessage(msgId : int) : void
alt
[msgId = 1]
5.1: updateMessage(msg : string) : void
[msgId = 2]
5.2: updateMessage(msg : string) : void
[default]
5.3: updateMessage(msg : string) : void

```
int main() {
    TrafficBillboard tb;
    tb.displayMessage();
    tb.changeState();
    tb.changeMessage(1); // could be any integer value
    ...
    return 0;
}
```

```
TrafficBillboard::TrafficBillboard() {
        board = new ElectronicBillboard("all␣clear");
}

void TrafficBillboard::displayMessage() {
        if (board->ison()) {
                cout << "Traffic␣warning:␣";
                board->updateBoard();
                cout<<endl;
        } else
                cout<<"Board␣is␣off"<<endl;
}

void TrafficBillboard::changeState() {
        board->onof();
}
```

```
void TrafficBillboard::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");
  }
}
```


Visual Paradigm Standard Edition(University of Pretoria)
Target
++()
+*()
+=()
T : typename
MyMaths
-x : T
-y : T
+MyMaths()
+add()
+multiply()
+setX()
+setY()
NewMaths
-adaptee : MyMaths<int>*
+NewMaths()
+NewMaths()
++()
+*()
+=()
+~NewMaths()

```
#ifndef MYMATHS_H
#define MYMATHS_H

template <typename T>
class MyMaths {
public:
    MyMaths(T, T);
    T add ();
    T multiply();
 //protected:
    void setX(T);
    void setY(T);
  private:
    T x;
    T y;
};

#include "MyMaths.cpp"

#endif
```

```
template <typename T>
MyMaths<T>::MyMaths(T v1, T v2)
{
    x = v1;
    y = v2;
}

template <typename T>
T MyMaths<T>::add()
{
    return x + y;
}


template <typename T>
T MyMaths<T>::multiply()
{
    return x * y;
}
```

```
template <typename T>
void MyMaths<T>::setX(T object)
{
    x = object;
}

template <typename T>
void MyMaths<T>::setY(T object)
{
    y = object;
}
```

T must be:

- assignable
- copy constructible; and
- operators + and * must be defined; and
- if T allocates memory on the heap - destructible as well

```
#ifndef TARGET_H
#define TARGET_H

class Target {
public:
    virtual int operator+(int) = 0;
    virtual int operator*(int) = 0;
    virtual int operator=(int) = 0;
};

#endif
```

Exercise

- You will be randomly assigned to a breakout group.
- In your group:
  - Define the `NewMaths` class that will be specified in `NewMaths.h`
  - Implement the class functions in `NewMaths.cpp`
- Discuss for 10min and then return to main room

```
#ifndef NEWMATHS_H
#define NEWMATHS_H

#include "Target.h"
#include "MyMaths.h"

 class NewMaths : public Target
 {
 public:
     NewMaths();
     NewMaths(int);
     virtual int operator+(int);
     virtual int operator*(int);
     virtual int operator=(int);
     ~NewMaths();
 private:
     MyMaths<int>* adaptee;

 };

#endif
```

```
#include <iostream>
#include "Target.h"
#include "NewMaths.h"

using namespace std;

int main()
{
    Target* obj = new NewMaths(4);

    int temp;
    temp  = (*obj +3);
    cout << temp << endl;

    *obj = 10;
    temp  = (*obj + 3);
    cout << temp << endl;

    return 0;
}
```

Changing the Maths example from an object to a class adapter.

- `MyMaths.h` and `MyMaths.cpp` do not need to change
- `Target` remains the same
- The **client** (*main*) stays the same

```
#ifndef MYMATHS_H
#define MYMATHS_H

template <typename T>
class MyMaths {
public:
    MyMaths(T, T);
    T add ();
    T multiply();
protected:   // Access to the setters no longer needed
    void setX(T);
    void setY(T);
  private:
    T x;
    T y;
};

#include "MyMaths.cpp"

#endif
```

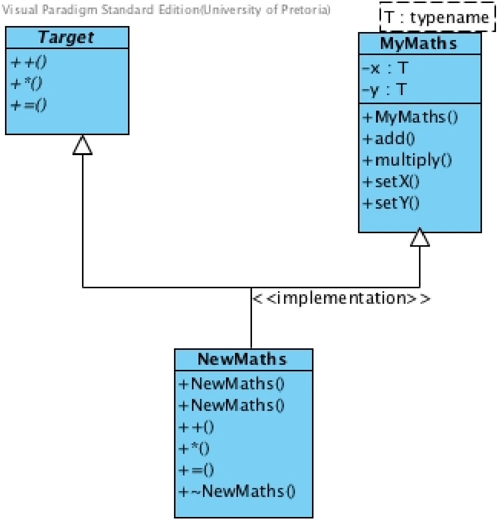
```
#ifndef TARGET_H
#define TARGET_H

class Target {
public:
    virtual int operator+(int) = 0;
    virtual int operator*(int) = 0;
    virtual int operator=(int) = 0;
};

#endif
```

- NewMaths.h changes a little
  - add private inheritance
  - remove private member
- instantiation and reference to the adaptee object removed from NewMaths.cpp
  - influences the constructor and destructor - no need to construct and destruct adaptee
  - calls to members of adaptee replaced with direct calls to functions in MyMaths

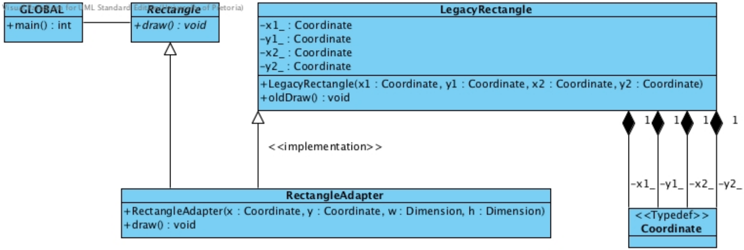
```
#ifndef NEWMATHS_H
#define NEWMATHS_H

#include "Target.h"
#include "MyMaths.h"

 class NewMaths : public Target,  private MyMaths<int>
 {
 public:
     NewMaths();
     NewMaths(int);
     virtual int operator+(int);
     virtual int operator*(int);
     virtual int operator=(int);
     ~NewMaths();
//private:
//      MyMaths<int>* adaptee;

 };

#endif
```

```
NewMaths::NewMaths() : MyMaths<int>(0,0)
{
    //adaptee = new MyMaths<int>(0,0);
}

NewMaths::NewMaths(int v) : MyMaths<int>(v,0)
{
    //adaptee = new MyMaths<int>(v,0);
}

NewMaths::~NewMaths()
{
    //delete adaptee;
}
```

```
int NewMaths::operator+(int i)
{
    //adaptee->setY(i);
    //return adaptee->add();
    setY(i);
    return add();
}

int NewMaths::operator*(int){  ...  }

int NewMaths::operator=(int v)
{
    //adaptee->setX(v);
    //return v;
    setX(v);
    return v;
}
```


Visual Paradigm Standard Edition(University of Pretoria)
Target
++()
+*()
+=()
T : typename
MyMaths
-x : T
-y : T
+MyMaths()
+add()
+multiply()
+setX()
+setY()
<<implementation>>
NewMaths
+NewMaths()
+NewMaths()
++()
+*()
+=()
+~NewMaths()

Class Adapter - Rectangle



(This example has been taken from: http://www.vincehuston.org/dp/adapter.html)

- `LegacyRectangle` defines a rectangle using the top left and bottom right coordinates of the corners
- `Rectangle` defines a rectangle with the top left coordinate and then the width on the x-axis and height in the y-axis

```
class RectangleAdapter : public Rectangle,
                         private LegacyRectangle
{
        public:
                RectangleAdapter( Coordinate x, Coordinate y,
                      Dimension w, Dimension h )
                : LegacyRectangle( x, y, x+w, y+h )
           {
               ...
           }
           virtual void draw()
           {
              oldDraw();
           }
};
```