Class Adapter

Linda Marshall and Vreda Pieterse

Department of Computer Science
University of Pretoria

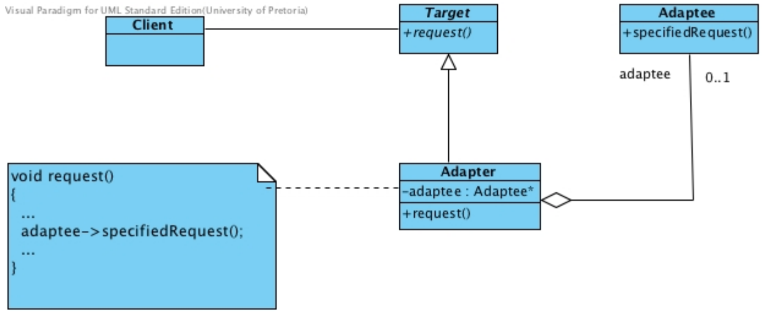
26 September 2014

Overview

Name and Classification: Adapter (Object and Class Structural)

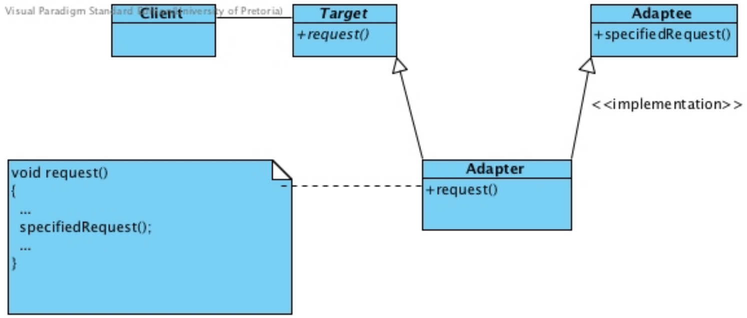
Intent: "Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. " GoF(139)

Object Adapter

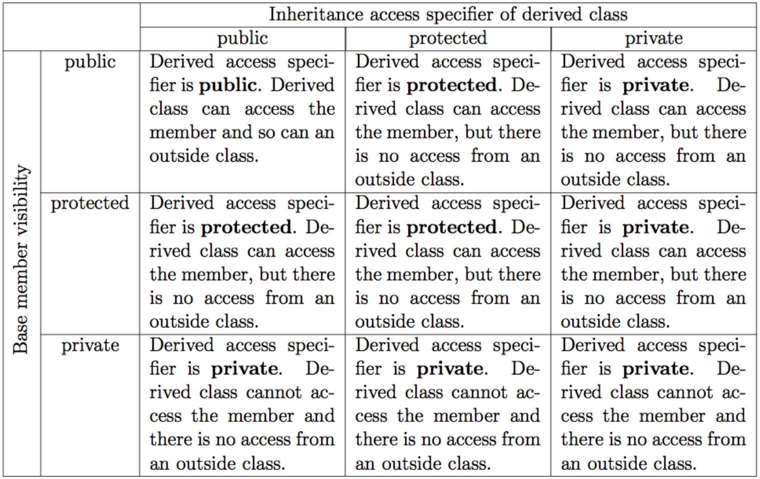


```
void request()
{
  ...
  adaptee->specifiedRequest();
  ...
}
```

Class


Visual Paradigm Standard Edition(University of Pretoria)
Client
Target
+request()
Adaptee
+specifiedRequest()
<<implementation>>
Adapter
+request()
void request()
{
...
specifiedRequest();
...
}

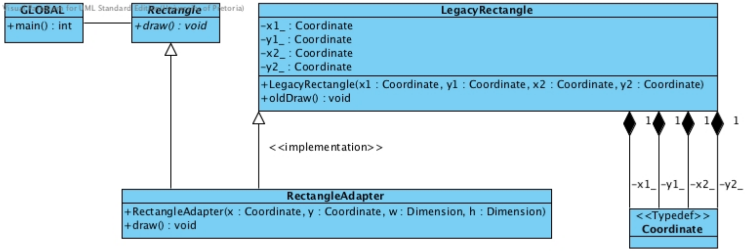
- Used to modify existing interfaces make it work after it has been designed
- Class Adapter makes use of mixins. Adapter inherits and implements Target (public inheritance). Adapter inherits only the implementation of Adaptee (private inheritance).



		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.

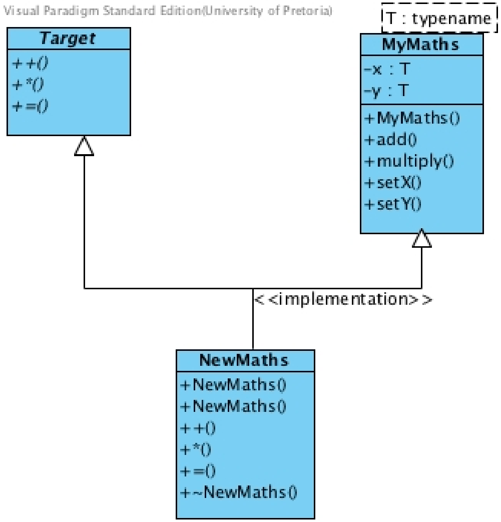
These are the same as the Object Adapter

- Target
- Adapter
- Adpatee
- Client


Visual Paradigm for UML Standard Edition(University of Pretoria)
GLOBAL
+main() : int
Rectangle
+draw() : void
LegacyRectangle
-x1_ : Coordinate
-y1_ : Coordinate
-x2_ : Coordinate
-y2_ : Coordinate
+LegacyRectangle(x1 : Coordinate, y1 : Coordinate, x2 : Coordinate, y2 : Coordinate)
+oldDraw() : void
<<implementation>>
1
1
1
1
-x1_
-y1_
-x2_
-y2_
RectangleAdapter
+RectangleAdapter(x : Coordinate, y : Coordinate, w : Dimension, h : Dimension)
+draw() : void
<<Typedef>>
Coordinate

- `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();
   }
};
```


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()

- `MyMaths.h` and `MyMaths.cpp` do not need to change
- `Target` remains 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
```

Remember, `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
```

- `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;
}
```

```
#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;
}
```