Design Patterns: C#

- Why use patterns?
- Short review: UML

- Patterns
  - Interface patterns
    - Adaptor classes in C#
    - Facade
    - Composite
  - Responsibility patterns
    - Singleton
    - Observer
  - Construction patterns
    - Simple Factory Pattern

Why use patterns?

- A pattern is a (well-known and tested) technique!
- Reflects **same** solution to the **same** type of problem
- A pattern has a **NAME**
  - Make design work in groups more easy (we can refer to a well-known solution)
  - The name reflects what the pattern can do for us in designing systems

Design patterns are to achieve functionality using the smallest number of classes

GoF

- Gang of Four
  - Book: Design Patterns
    - 23 patterns (the most used)

Review: UML

UML – is more than diagrams

UML describes:

- Model elements
- Relations
- Stereotypes
- Diagrams

Model elements

Model elements (structural elements):classes, interfaces

Relations

Relations: association, dependencies, generalization, aggregation and composites.

Stereotypes

S*tereotypes*: Additional values and limitations.

Diagrams

UML describes the following diagrams:

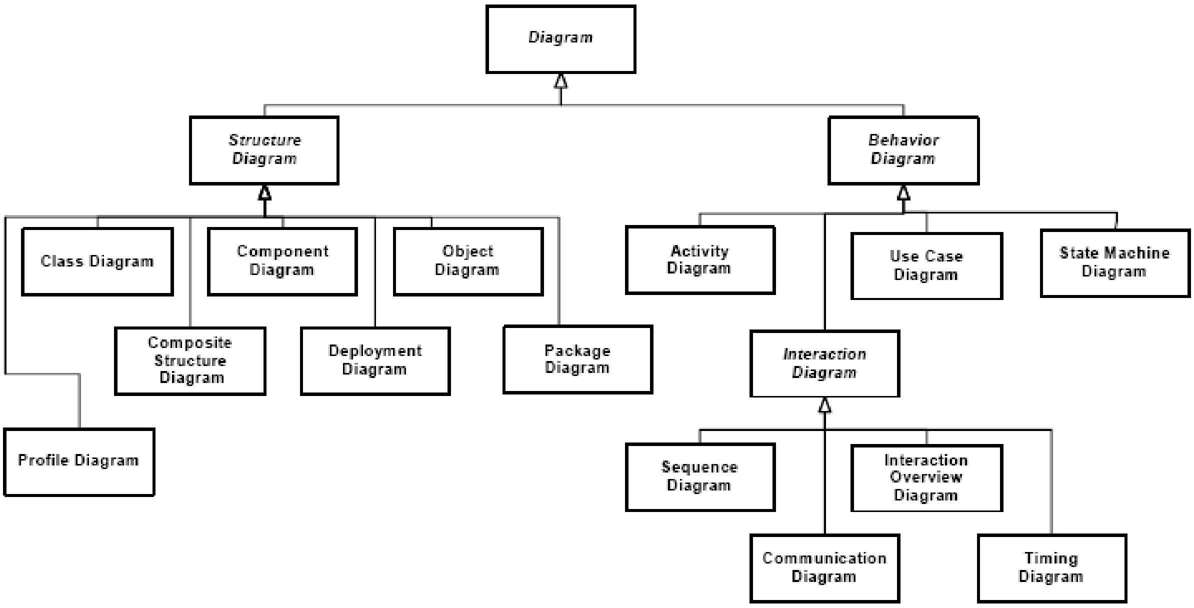


Figure A.5 - The taxonomy of structure and behavior diagram

source: uml.org

- Gives different views on the model:
- A static and dynamic view on the model.

Static diagram types:

- Use case diagrams
- Class diagrams
- Objekt diagrams
- Component diagrams
- Deployment diagrams

Dynamic diagram types:

- Sequence diagrams
- Communication diagrams
- Statechart diagrams
- Activity diagrams

More about Stereotypes

Stereotypes are defining adhancements to UML components. Stereotyper uses always *guillemet* (<< >>) .Creates new version of something well-known: <<*interface*>> is a special class type. Also <<create>> equals the call to the constructor.

Using BEGIN and END paranthesis more information can be added:

{documentation = }

{location =}

{persistence =}

GoF Patterns

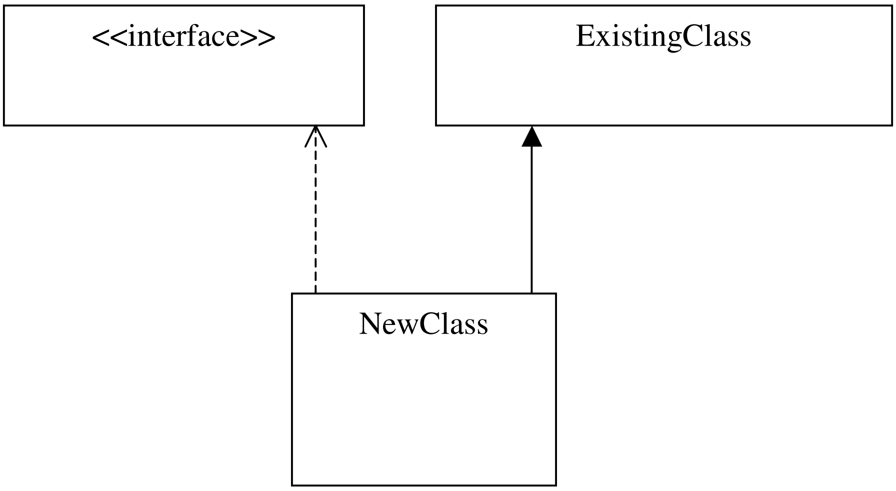
Interface Patterns

- Interfaces defines methods that must be implemented
- C# has **interfaces**
    - A C# interface
        - Allowss different functionality (according to the implementation)
            - using same call

Remember: An interface ia a contract

Adaptor classes in C#

- Adaptors and interfaces



- NewCalss is an adaptor class
- Inherits from ExistingClass
- MUST implement the content of the interface

(Example) Adaptors in .NET – database handling in .NET

- We will look at ADO.NET later!

- ADO.NET has a n-tier structure
  - Data is represented different on different levels

Example.

```
…

public class DataServices
{
   public static OleDbDataAdaptor
            CreateAdaptor(string select)
   {
        return new OleDbDataAdaptor( select,
                 CreateConnection());

   }

}
…
```

- CreateAdaptor returns an ”adaptor”

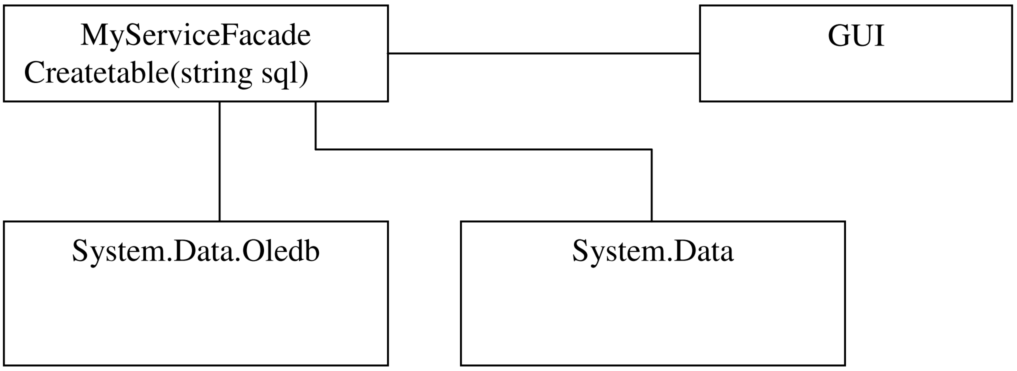
- OleDbDataAdaptor has a Fill() method
  - Used to fill data in a DataSet object
  - DataSet is an ”in-memory” table
- Several graphical Control classes can fetch data from a DataSet object
  - For instance: DataGrid

Facade patterns

- For instance used in connection with GUI design
  - Creates a facade between GUI and logic

- Reuse is then possible
  - Example: Windows GUI or Linux xWindows

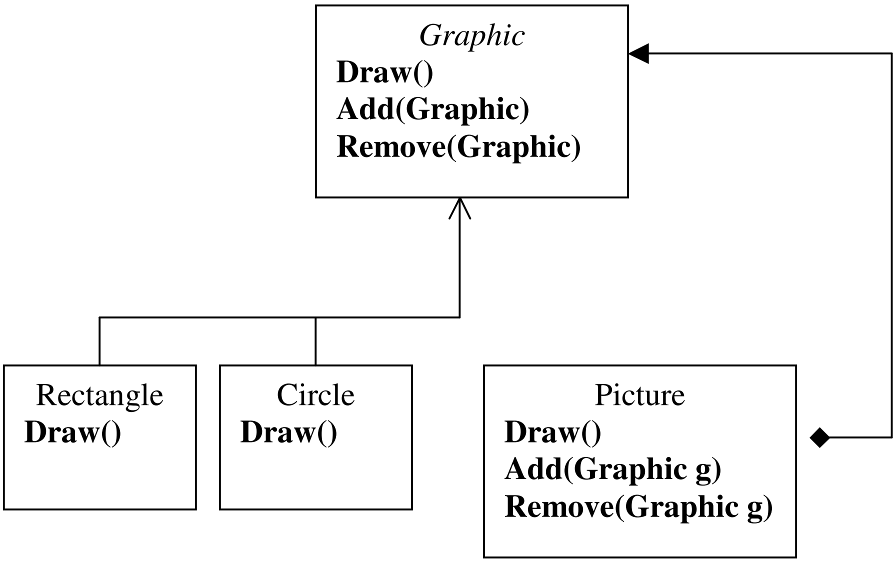
- A facade creates a simplified interface to a complex system!



Composite pattern

- Common behavior for different objects

Example:



- Basic idea:
  - How to build komplex graphic objects (example. Electronic comps.) by separated graphical building elements
  - How are objects treated the same way?
    - Use Composite patterns

Example C#:

```
using System;
using System.Collections;



abstract class Component
{
abstract public void AddChild(Component c);
abstract public void Traverse();
}




//Primitiv type
class Leaf : Component
{
private int value = 0;
public Leaf(int val)
{
    value = val;
}
public override void AddChild(Component c)
{
    //Not used in Leaf
}
public override void Traverse()
{
Console.WriteLine("Leaf:"+value);
}
}
//A composite type.
class Composite : Component
{
private int value = 0;
private ArrayList ComponentList = new ArrayList();
public Composite(int val)
{
value = val;
}
public override void AddChild(Component c)
{
ComponentList.Add(c);
}
public override void Traverse()
{
Console.WriteLine("Composite:"+value);
foreach (Component c in ComponentList)
{
c.Traverse();
}
}
}
class MyMain
{
public static void Main()
{
//creating a TREE structure.
Composite root = new Composite(100);// Root
Composite com1 = new Composite(200); //Composite 1
Leaf l1 = new Leaf(10);//Leaf1
Leaf l2 = new Leaf(20);//Leaf2
//Add two leafs to composite1
com1.AddChild(l1);
com1.AddChild(l2);
Leaf l3 = new Leaf(30);//Leaf3
root.AddChild(com1);//Add composite1 to root
root.AddChild(l3);//Add Leaf3 directlt to root
root.Traverse();//Single method for both types.
}
}
```

Responsibility patterns

Singleton

- Singleton asures one and only one instance of a class

Example C#

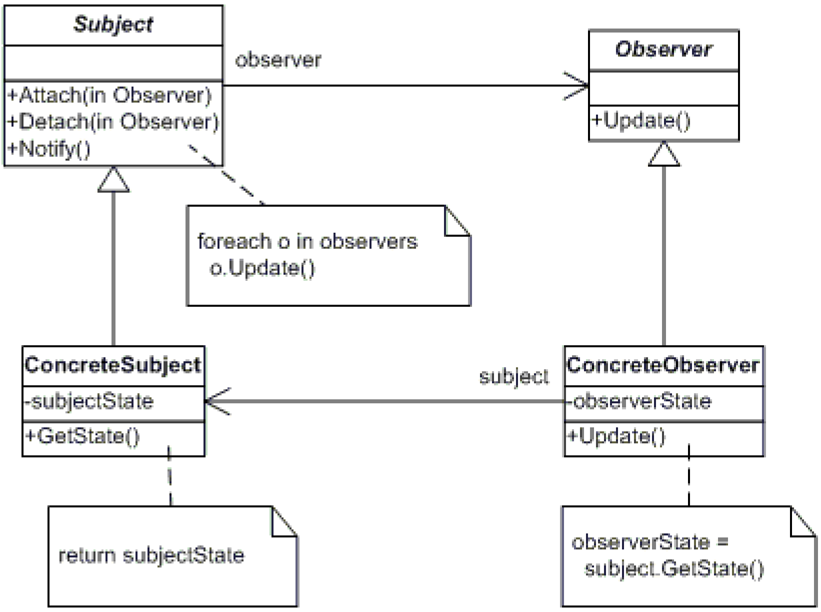
```
using System;
class SingleInstanceClass
{
private static SingleInstanceClass sic= null;
private static bool instanceFlag = false;


private SingleInstanceClass()
{
}

public static SingleInstanceClass Create()
{
if(! instanceFlag)
{
sic = new SingleInstanceClass();
instanceFlag = true;
return sic;
}
else
{
return null;
}
}
protected void Finalize()
{
instanceFlag = false;
}
}





class MyClient
{
public static void Main()
{
SingleInstanceClass sic1,sic2;
sic1 = SingleInstanceClass.Create();
if(sic1 != null)
Console.WriteLine("OK");
sic2 = SingleInstanceClass.Create();
if(sic2 == null)
Console.WriteLine("NO MORE OBJECTS");
}
}
```

Observer patterns

- GUI uses objects for business logic
- How knows GUI that information has changed (state is changed)?
    - The purpose of the the observer is to inform when an object change it's state
    - And then inform all attached objects

- C# support for Observer
    - Delegates
    - Events

Example: Plain Oberserver Pattern:



```
// Observer pattern


using System;
using System.Collections;

namespace DoFactory.GangOfFour.Observer.Structural
{

  // MainApp
```

```
class MainApp
{
  static void Main()
  {
    // Konfigurering af Observer pattern
    ConcreteSubject s = new ConcreteSubject();

    s.Attach(new ConcreteObserver(s,"X"));
    s.Attach(new ConcreteObserver(s,"Y"));
    s.Attach(new ConcreteObserver(s,"Z"));

    // Change subject and notify observers
    s.SubjectState = "ABC";
    s.Notify();

    // Wait for user
    Console.Read();
  }
}

// "Subject"

abstract class Subject
{
  private ArrayList observers = new ArrayList();

  public void Attach(Observer observer)
  {
    observers.Add(observer);
  }

  public void Detach(Observer observer)
  {
    observers.Remove(observer);
  }

  public void Notify()
  {
    foreach (Observer o in observers)
    {
      o.Update();
    }
  }
}

// "ConcreteSubject"

class ConcreteSubject : Subject
{
  private string subjectState;

  // Property
  public string SubjectState
  {
    get{ return subjectState; }
```

```
      set{ subjectState = value; }
    }
  }

  // "Observer"

  abstract class Observer
  {
    public abstract void Update();
  }

  // "ConcreteObserver"

  class ConcreteObserver : Observer
  {
    private string name;
    private string observerState;
    private ConcreteSubject subject;

    // Constructor
    public ConcreteObserver(
      ConcreteSubject subject, string name)
    {
      this.subject = subject;
      this.name = name;
    }

    public override void Update()
    {
      observerState = subject.SubjectState;
      Console.WriteLine("Observer {0}'s new state is {1}",
        name, observerState);
    }

    // Property
    public ConcreteSubject Subject
    {
      get { return subject; }
      set { subject = value; }
    }
  }
}
```

Simple Factory Pattern

- If a client uses a class a new object is createdby calling the constructor
    - If a client should not be able to choose between several classes
    - Uses a ”factory class”

```
Using System;

class Factory
{
public Base GetObject(int type)
{
Base base1 = null;
switch(type)
{
case 1:
  base1 = new Derived1();
  break;
case 2:
  base1 = new Derived2();
  break;
}
return base1;
}
}
interface Base
{
  void DoIt();
}
class Derived1 : Base
{

   public void DoIt()
  {
     Console.WriteLine("Derived 1 method");
   }
}
class Derived2 : Base
{
    public void DoIt()
    {
            Console.WriteLine("Derived 2 method");
    }
}

//Client class
//Client class needn’t know about instance creation. The creation of Product is //deferred to
the Factory class
```

```
class MyClient
{
  public static void Main()
  {
             Factory factory = new Factory();       //Decides the object type
             Base obj = factory.GetObject(2);
             obj.DoIt();
            }
```