Design Patterns in C#

Mathias Bartoll
Nori Ahari
Oliver C. Moldez

Department of Computer Science
Mälardalen University
Västerås, Sweden

mathias@mathiasbartoll.com
nori@ahari.info
oca99001@student.mdh.se

4 Juni, 2004

Abstract

The idea behind design patterns is to save good object oriented design solutions and reuse them to solve similar problems. In The early 1990:s Erich Gamma et al [1] described 23 design patterns, six of which will be described here.

The new object oriented language C#, presented by Microsoft is strongly influenced by Java and C++. Still some new and interesting features are introduced that simplify object oriented design. As it is shown in this work *interfaces* can be used to implement patterns such as *Adapter* and *Strategy*, and *Events* come in handy when the *Observer* pattern is to be used. However C# is intended to work together with the .NET platform and is therefore tightly coupled with some .NET specific issues.

1 Introduction

Object oriented programming has been the dominating style in software development for quite a few years. The intuitive way in which object oriented languages allow us to divide our code into objects and classes is what makes this style of programming attractive. Another aim of object oriented program design is to make code more reusable. After all an object, say for example a chair, will always be a chair and if we write a chair class for a program, it is likely that we can reuse that class in another context. It has however been shown that designing reusable object oriented software is not always that easy. A good software design should, not only solve existing problems, but also concern future problems. It should make the program flexible, easy to maintain and to update. Design patterns help us address these issues. The idea is quite simple; we want to document and save design solutions that have been used and worked for reoccurring problems, in order to use them again in similar situations. Erich Gamma et. al describe 23 different design patterns in their book [1]. This book is one of the first publications in the area.

The aim of this work is to take a closer look at how some of the known design patterns can be implemented in C#, and to investigate whether the new features of the language in fact do make it easier to design object oriented software. This paper should not be seen as an introduction to C#. We will briefly go through some of the features of the language that are relevant to implementing most design patterns, whereas some other details are omitted. For a more extensive overview of C# the reader is referred to some of the references at the end of this work.

Since C# is closely related to the .NET platform we will start by a short introduction to the .NET framework in the next chapter. Chapter 3 covers some features in C#, in chapter 4 we describe six of the design patterns presented by Gamma et. al [1] and look at how they can be implemented in C#. And finally the work will be wrapped up by some discussions and conclusions.

2 .NET Framework

2.1 Introduction

The .NET platform is a new development framework providing a new Application Programming Interface (API), new functionality and new tools for writing Windows and Web applications. But it is not only a development environment, it is also an entire suite of servers and services, as shown in figure 1 [7], that work together to deliver solutions to solve today's business problems. One of the main ideas with .NET is to make the connectivity and interoperability between businesses easier. In this section we will briefly focus on the main aspects in .NET Framework that relates to the C# language. These two technologies are very closely intertwined. The basic idea of .NET Framework is to have several languages use the same underlying architecture, which should have a natural relationship with the various forms of the Windows operating system. Most of Microsoft's next generation of

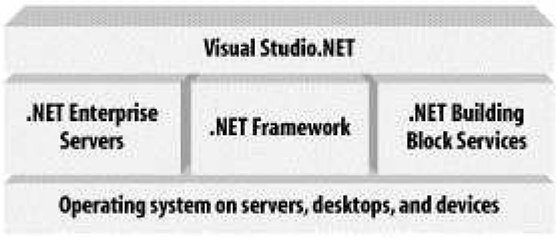
Visual Studio.NET
.NET Enterprise Servers
.NET Framework
.NET Building Block Services
Operating system on servers, desktops, and devices

Figure 1: The Microsoft .NET platform.

programming languages, including the latest editions of C++ and Visual Basic, use the .NET environment. However, C# is the first major language designed from the beginning with .NET in mind.

The .NET Framework is a new development and runtime infrastructure that changes the development of business applications on the Windows platform. It includes the Common Language Runtime (CLR) and a comprehensive class library for building web and Windows applications.

2.2 Common Language Runtime

The Common Language Runtime (CLR) is the mechanism through which .NET code is executed. It provides a lot of added value to the programs it supports. Because it controls how a .NET program executes and sits between the program and the operating system, see figure 2 [7]. It can implement security, versioning support, automatic memory management through garbage collection, and provide transparent access to system services. When a program is compiled for the CLR,

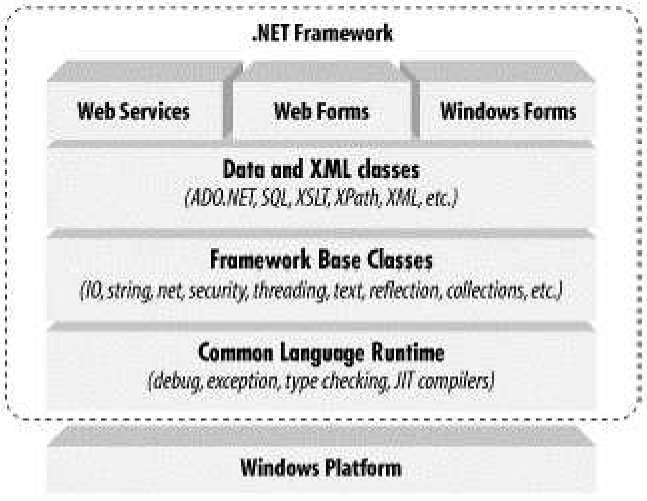
.NET Framework
Web Services
Web Forms
Windows Forms
Data and XML classes
(ADO.NET, SQL, XSLT, XPath, XML, etc.)
Framework Base Classes
(IO, string, net, security, threading, text, reflection, collections, etc.)
Common Language Runtime
(debug, exception, type checking, JIT compilers)
Windows Platform

Figure 2: The .NET Framework.

it is converted to a portable executable (PE) file that consists of different sections.

These sections are shown in figure 3 illustrated by Thai T. et. al [7]. The CLR header stores information to indicate that the PE file is a .NET executable and the CLR data section contains metadata and Microsoft Intermediate Language (MSIL, or IL for short) code. Every common language runtime–compliant development

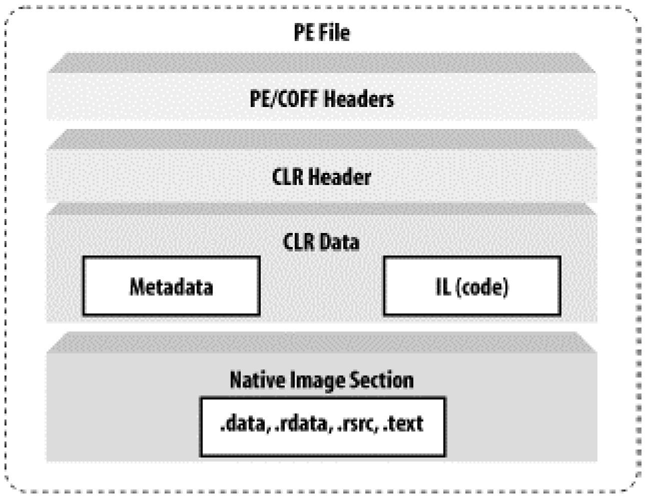

Figure 3: PE file format.

tool compiles its own source code into IL code. Because all development tools produce the same IL, regardless of the language in which their source code is written, differences in implementation are gone by the time they reach the common language runtime.

Metadata is data that is used to describe classes and what they can do, separate from the code of the class itself. It is important to understand that metadata is not part of the class in the same way that variables and methods are, but instead it is used to describe classes. The CLR uses metadata for many purposes including; locating and loading classes, laying out objects in memory, finding out what methods and properties a class has, enforcing security and discovering the class's transactional behaviour. You can ask an object at runtime for this type of information such as its type, methods, properties, events and so on.

Most of the metadata associated with a class is provided by the compilation process, but it is possible to create your own metadata items, called attributes, and attach them to your own classes. This topic is covered in chapter 3.4.

3 The Programming Language C#

3.1 Introduction

C# was designed for the .NET platform and is the first modern component–oriented language in the C and C++ family. At the heart of any object oriented language lays its support for defining and working with classes. Classes define new types, allowing you to extend the language to better model the problem you are trying to solve. C# contains keywords for declaring new classes with methods and properties, and also for implementing encapsulation, inheritance and polymorphism, the three pillars of object-oriented programming. Some other key features of this language include interfaces, delegates, namespaces, indexers, events and attributes. No header or Interface Definition Language (IDL) files are needed.

The .NET software development kit defines a "Common Language Subset" (CLS), which ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic and Visual C++. C# itself does not include a class library.

There are several advantages of sharing libraries across languages. It reduces learning time when moving from one .NET-supported language to another. It provides a larger body of samples and documentation, because these do not need to be language specific. It also allows better communication between programmers who work in different .NET-supported languages.

This part covers some of the key features that the C# language provides for the developer. You will see that some of these features will be very useful when implementing design patterns.

3.2 Classes and Methods

C# is a strictly object oriented language which means that (almost) everything is an object. The syntax of defining classes is quite similar to C++ and Java. In general C# has borrowed ideas from both Java and C++ and added some new features to solve some of the most common problems in object oriented languages [6].

3.2.1 Defining Classes

A class is defined by using the keyword *class* followed by the name of the class. The class's members and methods are then defined inside curly brackets ({}). Each class can have one or several constructors.

There is four access modifiers for class members in C# [6]:

- **public**: means the member is accessible from outside the class's definition and derived classes
- **protected** : The member can be accessed by derived classes only
- **private** : The member cannot be accessed outside the scope of the class, not even by derived classes
- **internal** : The member is visible inside the current compilation unit.

The *student* class defined here has two constructors. A constructor does not have a return value. Notice also that each method can be overloaded with different type or number of arguments.

```
class student
{
   public student(string stName)
   {
      name = stName;
   }

   public student(string stName, string gr)
   {
      name = stName;
      grade = gr;
   }

   public void SetGrade(string gr)
   {
      grade = gr;
   }

   public string GetGrade()
   {
      return grade;
   }

   private string name;
   private string grade;
}
```

Each C# application must contain at least one class that has a *main* method. It is also required that *main* is defined as public and static. For example:

```
class studentApp
{
   public static void main()
   {
      student st = new student("Nori");
      st.SetGrade("A");
   }
}
```

The main method works as the entry point for the application, however you can have several classes containing a main method in C#. In that cases you need to specify at compile time which class's main method should be used as the application's entry point. This can be done by using the compilers *main* switch like the following:

```
csc multipleMain.cs / main:studentApp
```

Here the "csc" is the C# compiler, multipleMain is the file we are compiling and "studentApp" is the name of the class that contains the main method we want to use as the entry point.

3.2.2 Inheritance

Inheritance in C# works in the same fashion as with other object oriented languages. The inherited class can be viewed and treated as the base class but not the other way around. The syntax for inheriting a class from another is as follows:

```
class derivedClass : baseClass
```

Where "derivedClass" is the name of the new class, and "baseClass" is the name of the base class. C# does not allow multiple inheritance, however the concept of interfaces is introduced to be used instead. Interfaces are described later in chapter 3.5.

3.2.3 Ref and Out Method Arguments

Recognising the fact that pointers are one of the biggest source of bugs when developing software, the concept of pointers, as known in C and C++, does not exist in C#. It is however still possible to pass reference variables as method arguments by using the keywords *ref* and *out*.

Consider the example of a *colour* class that contains three member variables of type integer, namely red, green, and blue. These three variables determine the value of the colour class based on the RGB standard [6]. Using reference variables we can retrieve all three values through a single method call. The code would look something like this:

```
class colour
{
   private int red;
   private int green;
   private int blue;
   public void GetColors(ref int red, ref int green, ref int blue)
   {
      red = this.red;
      green = this.green;
      blue = this.blue;
   }
}

The code at the client side would be:
...
int red = 0;
int green =0;
int blue = 0;

colour col = new colour(...); GetColors(ref red, ref green, ref
blue);
```

The *out* keyword can be used in the same way, the method signature of the *Get-Colors* method would then look like this:

```
public void GetColors(out int red, out int green, out int blue)
```

Using the *ref* and *out* keywords will inform the compiler that the argument being passed points to the same place as the variables in the calling code. The difference between the two is that with the *ref* keyword the arguments must be initialised before being passed, otherwise the compiler will generate an error. This means that *ref* should preferably be used when the operation carried out in the method is dependent on the value of the passed argument, thereby forcing the client to a proper initialisation.

3.2.4 Variable Method Parameters

There are circumstances where the number of arguments to a method is not known until runtime. An example could be a method that prints a line on a graph by linking together an arbitrary number of points, passed to it as arguments [6]. This is possible by using the *params* keyword and specifying an array in the methods argument list. In the following code example we assume that the class named *Point* has been defined to contain the x and y coordinates of a point.

```
...
public void DrawLine(params Point[] p) {
   for (int i = 0; i < p.GetLength(0); i++)
   {
      // Draw line
   }
}
...
```

3.3 Properties, Arrays and Indexers

Two of the new features introduced by C# are *properties,* sometimes called smart fields, and *indexers,* sometimes called smart arrays. The two concepts are related to each other because both have been introduced to make client access to class's members more intuitive. Also the syntax of defining properties and indexers are very similar. That is why both of theses concepts will be covered together in this chapter. We will start by looking at properties, then see how arrays work in C# and finally how indexers can be used to treat objects as arrays.

3.3.1 Properties

It is common object oriented practice to declare some of the members of a class as private and allow access to these members only through public accessor methods, sometimes called "getters" and "setters". This approach can be useful if you for example want to perform some verification on the input from the client before you let the value of your member variable change. Consider a class containing a *streetAddress* member and a *zipCode* member. In this case it would be a good idea to verify that the zip code provided by the client is consistent with the classes street address [6]. This could be done in the following fashion:

```
class address
{
   protected string streetAddress;
   protected string zipCode;

   public string getZipCode()
   {
      return (zipCode);
   }

   public void setZipCode(string zc)
   {
      // Validate the zip code against a database
      zipCode = zc;
   }
}
```

Code at the client side uses the *setZipCode* method to pass a new zip code to the class. C# properties provide the same functionality but make the code at the client side more elegant by letting the private members be accessed as though they were public, but still allowing the client to perform the necessary validation. Here is the example above rewritten using properties:

```
class address
{
   protected string streetAddress;
   protected string zipCode;
   public string ZipCode
   {
      get
      {
         return (zipCode);
      }
      set
      {
         // Validate the zip code against a database
         zipCode = value;
      }
   }
}
```

The code at the client side would then look like this:

```
...
address addr = new address();
addr.Zipcode = "722 28";

string zip = addr.ZipCode;
```

This way the client doesn't need to know about the existence of the accessor methods.

3.3.2 Arrays

Arrays are, like most other things in C#, objects. All arrays inherit from the *System.Array* class [6]. The following example illustrates how a simple single dimensional array is instantiated and used.

```
using System;

int[] MyArray; MyArray = new int [10];

for (int i = 0; i < MyArray.Length; i++)
   Console.Writeline(MyArray[i]);
...
```

Notice how *MyArray* is declared to hold integer variables at the first line of code. Like any other object, *MyArray* should be instantiated using the *new* keyword before it can be used. This also means that if you have a class that holds an array as a member variable, the array should be instantiated at some point, for example in the classes constructor. In the for loop the *Length* property of the *System.Array* class is used to determine the number of iterations. More information about the members and methods of the *Array* class can be retrieved from [12].

3.3.3 Indexers

Indexers are a mechanism to let you treat any object as though it was an array. One example of where this would make sense is a windows listbox class (See MSDN for more information) that should provide the clients by some methods that enables them to insert strings into the listbox, or access and modify existing strings. The Microsoft Foundation Classes (MFC) provides a class called CListBox that lets us operate on a listbox through member functions such as *AddString* and *InsertString.* In its simplest form the listbox could be viewed as an array of strings. It would be elegant if we could operate directly onto its string members in the following way:

```
listboxClass lb = new listboxClass();

lb[0] = "some data";
lb[1] = "some other data";
```

This can be done through C# indexers. The syntax of defining indexers is quite similar to defining properties. The difference is that the indexer takes an *index* argument and the *this* keyword is used as the name of the indexer to imply that the class itself is being used as an array [6]. In the following example we use a .NET built in type called *ArrayList*, which is used to store a collection of objects, to illustrate how we can define our own listbox class. Through the use of indexers the class can be treated as an array:

```
using System;
using System.Collections;
class MyListBox
{
   protected ArrayList data = new ArrayList();

   public object this [int idx]
   {
      get
      {
         return(data[idx]);
      }
      set
      {
         // If there already exists some data at the
         // given index, replace it with the new data
         if (idx > -1 && idx < data.Count)
         {
            data[idx ] = value;
         }
         // Else add the new data to the ArrayList
         else if (idx == data.Count)
         {
            data.Add(value);
         }
         else
         {
            // Error handling code goes here
         }
      }
   }
}
```

Just like we have discussed before the code at the client side can now access members of *MyListBox* through their index number:

```
MyListBox lb = new MyListBox();

lb[0] = "some data";
lb[1] = "some other data";
```

3.4 Attributes

Attributes in C# is a technique to add metadata to your classes. Metadata is declarative information about elements of source code, such as classes, members, method etc. The information provided by the attribute is stored in the metadata of the element and can be retrieved at runtime by a technique called reflection. The use of attributes allows you to change the behaviour of an object at runtime, provide information about an object or objects in your application, and to mediate information to other designers. Attributes can even by useful when debugging.

Attributes were introduced by Microsoft to allow the developer to add declarative information to the source code and to get rid of the DLL “hell’ [8]. Instead having two files for a component, one that contains the application and another the application information, attributes add the metadata along with the application assembly. With DLL:s you need both files for the component to function and if one is lost you could not use the component.

An attribute is a class that you can add to different programming elements, such as assemblies, objects, struct, enums, constructor, methods, return values, and delegates. All attributes are derived from the System.Attribute class. There are two kinds of attributes, *intrinsic* and *custom*. Intrinsic attributes are a part of the CLR and they are the most common used attributes. Custom attributes are attributes created by developer. When applying an attribute, it must be placed directly after the using statement and before the class declaration. The formal syntax of an attribute is:

```
using System;
```

```
[attribute(positional_parameters, name_parameter = value)...]

class myClass {...}
```

This attribute example tells us that the attribute applies to a class by the parameter AttributeTargets.class.

```
using System;

[AttributeUsage(AttributeTargets.Class)]

public class myAttribute :Attribute
{...}
```

Any attribute can have one or more parameters. Parameters can either be positional or named. Positional parameters are passed to the attribute constructor and named parameters are implemented as properties.

```
using System;

[AttributeUsage(attributeTargets.class)]

public class myAttribute : Attribute
{
   public myAttribute(string url) // Positional parameter
   {
      this.url = url;
   }

   public string Subject(string subject) // Named parameter
   {
      get { return subject; }

      set { this.subject = subject; }
   }

   public string Url(string url)
   {
      get { return url; }
   }

   private url;
   private subject;
}
```

```
// Now we apply the attribute to a class
[myAttribute(("http://www.myclassinfo.com") Subject = "myClass")]

class myClass
{
   {...}
}
```

You can have multiple attributes attached to a programming element. You can either stack the attributes on top of each other or you can separate them with commas.

```
using System;

[myAttribute1(parameter...)]
[myAttribute2(parameter...)]

class myClass
{
   {...}
}

// Comma separated example below

using System;

[myAttribute1(parameter...), myAttribute2(parameter...)]

class myClass
{
   {...}
}
```

Attributes can be defined to apply to more than one element type. The “or” (|) operator is used to separate the targets.

```
[AttributeUsage(AttributeTargets.Method
AttributeTargets.Constructor)]

public class myAttribute : System.Attribute { //code here }
```

Attributes as default are not allowed to be instantiated more then once, but sometimes it can be useful to have several instances of the same attribute.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]

public class myAttribute : System.Attribute { //code here }
```

Reflection is used to access the attributes information at runtime as mentioned earlier. The MemberInfo class is used for retrieving information stored by attributes in the metadata. MemberInfo class is found in System.Reflection namespace.

```
System.Reflection.MemberInfo[ ] attributeArray;

attributeArray = typeof(myClass).GetCustomsAttributes( );
```

After getting the array you can iterate through the array to get the information stored in the attributes.

```
foreach(Attribute atr in attributeArray)
{
   if (atr is myAttribute)
   {
      myAttribute myA = (myAttribute)atr;
      Console.WriteLine("Web page = {0} , class {1} = ",
         myA.Url, myA.Subject);
   }
}
```

3.5 Interfaces

One feature not supported by the C# programming language is multiple inheritance. As an example of where multiple inheritances is useful consider the following:

Let's say that you are programming a university application in which you use an *employee* class to represent a person employed by the university. A *researcher* is a subclass of *employee* with tasks such as doing experiments and writing papers. Researchers also sometimes teach courses. There are also other people teaching courses that are not researchers. Multiple inheritance would have allowed you to create a new class called something like *Teacher* and let the *researcher* class, along with other employee subclasses who teach courses, inherit both from the *employee* and from the *Teacher* classes. In this case a researcher is an employee, which also should display the characteristic behaviours of a teacher. C# introduces interfaces as a built in part of the language to represent *behaviours* as opposed to classes that represent objects [6].

An interface is basically a contract between a class and a client that guaranties that the class implements the methods specified in the interface. In other words, interfaces contain the public signature of methods, events and properties but it is up to the class, which is said to *implement* the interface, to provide the implementation of these methods. The implementing class inherits from an interface in the same way as from a base class. An instance of this class can then be casted into the interface and access its methods. In this way interfaces can be used to reach the same objectives as with multiple inheritance.

As an example we can crate an interface called *ITeach* and let the *resarcher* class (described above), which also inherits from an *employee* base class, implement it.

```
interface ITeach
{
   string SetGrade(string studentId);
}

class employee
{
   protected int Salary;
   public int salary
   {
      get { return this.Salary; }
      set { this.Salary = value; }
   }

   protected string Emp_ID;
   public string emp_ID
   {
      get { return this.Emp_ID; }
      set { this.Emp_ID = value; }
   }
}

class reseracher: employee, ITeach
{
   public researcher(string id, int sal)
   {
      emp_ID = id;
      salary = sal;
   }
   public string SetGrade(string studentId)
   {
      if (studentId == "Martin")
         return("A")
      else
         return("D")
   }
}
```

As mentioned above an instance of the researcher class can be casted into the interface(s) it implements. This means that we can use the **is** and **as** operators to see whether or not a class implements a certain interface or not. The "*is*" operator can be used to see whether the runtime type of an object is the same as another, and returns a Boolean value. The keyword *"as"* is used to cast an object into another type and returns null if the cast fails. The following example illustrates this:

```
class MyApp
{
   static void main()
   {
      researcher R = new researcher("Martin", 20000);
      if(R is ITeach)
      {
         ITeach T = (ITeach)R;
         T.SetGrade("Nori");
      }
      // Alternatively
      ITeach T = R as ITeach;
      if(T != null)
         T.SetGrade("Nori");
   }
}
```

In this particular case however it is not necessary to cast R into ITeach. Note how the *researcher* class declares the SetGrade method:

```
public string SetGrade(string studentId)
```

This means that *SetGrade* exists in the public namespace of the *researcher* class and can be invokeddirectly on an instance of researcher:

```
R.SetGrade("Nori");
```

If *researcher* however had implemented the *SetGrade* function in the following way the cast would have been necessary:

```
string ITeach.SetGrade(string studentId)
```

3.6 Error Handling

Error handling in C# provides a flexible way to handle errors, requires less overhead and provides meaningful error messages. Error handling is managed by Exception. All exception derives from the System.Exception class.

To handle Exceptions, the code, which is to be monitored for errors are put in a try block. After the try block follows a catch block. The code that execute when an exception occurs and is caught exists within the catch block.

```
try {
   // Code to monitor for exception
}
catch(System.Exception e) {
   // Code to execute if an exception happened.
}
```

The Exception is handled in the nearest catch block, but if they're no catch block to handle the exception the CLR unwinds the stack until it find an appropriate exception handler (catch block). If CLR returns back to main() and still no exception handler is found then the application is terminated.

Those who are familiar with C++ recognise this error handling technique, but there are some new features provided by C#. You can declare a finally block to your error handling code. The finally block provides a way to guarantee that a peace of code is always executed whether an Exception is thrown or not.

```
try {
   // Code to monitor for exceptions
}
catch (System.Exception e) {
   // Code to handle the exception
}
finally {
   // Code that always is executed whether
   // an exception is thrown or not.
}
```

Some other new features are as mentioned earlier that all Exceptions derive from the System.Exception class while in C++ any type can be used as an Exception. All System-level Exceptions are well defined in Exceptions classes. In C# you can create and throw your own Exceptions as long as they derive from the System.Exception class. All Exceptions that are caught in the catch block can be thrown again.

```
catch (System.Exception cought) {
   throw caught;
}
```

3.7 Delegates and Events

In the early days of computing, a program would begin its execution and then proceed through the steps until it completed. If it had some interaction with the user this was strictly controlled and limited to filling in fields. In today's Graphical User Interface (GUI) programming model the user might take many different actions, which require a different approach, known as *event-driven programming*. Each action like clicking buttons or menu selection causes an event to be raised. In these situations you often don't know in advanced which method or even which object to call to execute the requested action. This is where C# adds support through event and delegates.

3.7.1 Delegates

In chapter 3.5 we saw how interfaces specify a contract between a caller and an implementer. Delegates are similar to interfaces but rather than specifying an entire interface, a delegate merely specifies the form of a single method. Also, interfaces are created at compile time, whereas delegates are created at runtime.

As Liberty [5] states, in the programming language C#, delegates are first-class objects, fully supported by the language. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. You can encapsulate any matching method in that delegate without having to know at compile time which method that will be invoked. A delegate in C# is similar to a function pointer in C or C++. But unlike function pointers, delegates are object-oriented, type-safe and secure managed objects. This means that the runtime guarantees that a delegate points to a valid method and that you don't get errors like invalid address or that a delegate corrupting the memory of other objects. Furthermore, delegates in C# can call more than one function; if two delegates are added together, a delegate that calls both delegates is the result. Because of their more dynamic nature, delegates are useful when the user may want to change the behaviour of the application. For example, a collection class implements sorting, it might want to support different sort orders. The sorting could be controlled based on a delegate that defines the comparison function.

There are three steps in defining and using delegates: declaration, instantiation, and invocation. Delegates are declared using the keyword *delegate*, followed by a return type, the signature and the parameters. In the following example we define a class with a callback function using a delegate.

```
using System;

class TestDelegate
{
   // 1. Define a callback prototype - Declaration
   delegate void MsgHandler(string strMsg);

   // 2. Define callback method
   void OnMsg(string strMsg)
   {
      Console.WriteLine(strMsg);
   }

   public static void Main( )
   {
      TestDelegate t = new TestDelegate( );

      // 3. Wire up our callback method - Instantiation
      MsgHandler msgDelegate = new MsgHandler (t.OnMsg);
```

```
        // 4. Invoke the callback method indirectly - Invocation
        msgDelegate("Hello, Delegate.");
    }
}
```

When the delegate is defined it can be instantiated with any method that matches the delegate's signature and return type. Once instantiated, the delegate reference can be used directly as a method as in the example above.

When designing your delegates, it is important to consider when to create them. C# allows you to define the appropriate delegates as static members. In that case the client doesn't have to instantiate the delegate each time it will be used. However the drawback of a static member is that the delegate is always created, even if it is never used. It would be better if the delegate were created on the fly, as needed. This can be done by replacing the static functions with properties [6].

3.7.2 Events

A class can use an event to notify another class (or other classes) that something has happened. In GUIs an event might be a menu selection, but events are well suited for any asynchronous operation, such as a file being changed. Events are something that happens in an unpredictable order and the system must respond to it.

In C#, events follow the publish-subscribe design pattern in which a class can publish a set of events and other classes can subscribe to the specific events. When an event is raised by the publishing class, the runtime takes care of notifying all the subscribed classes [6].

The implementation of events in C# is done by delegates. Liberty [5] describes it as "The publishing class defines a delegate that the subscribing classes must implement. When the event is raised, the subscribing class's methods are invoked through the delegate". However, keep in mind that event handlers in .NET Framework has some grammatical rules. The event handlers always return void and it must take two arguments that represent objects. The first argument is the object that raised the event (the publisher) and the second object contains the information about the event. The second object with information must derive from the .NET Framework's EventArgs class. This class is the base class for all event data. It contains a static readonly property called *Empty*, which represents an event with no state.

Suppose you want to create a class named Clock, which will use events to notify each subscriber when the time changes value by one second. The keyword *event* controls how the subscribing classes access the event property. The declaration of the event and the event-handler delegate are as follow [5]:

```
public event SecondChangeEventHandler TriggerSecondChange;
```

The declaration above consists of an event called TriggerSecondChange and is implemented by a delegate of type SecondChangeEventHandler.

The SecondChangeEventHandler implementation is:

```
public delegate void SecondChangeEventHandler (
   object clock,
   TimeInfoEventArgs timeInformation );
```

This delegate returns, as mentioned earlier, void and takes two objects as arguments. The second object derives from the EventArgs class and in this case it is the TimeInfoEventArgs.

The benefits that you gain by using events and delegates are several. For instance, there can be any number of subscribing classes that are notified when an event is raised. A button can publish an OnClick event and any number of unrelated objects can subscribe to that event, receiving notification when the button is clicked. Another big advantage is the decoupling of the publisher and the subscribers. They run independently of one another and can be changed without any consequences for the other part. This is highly desirable because it makes the code more flexible, robust and easier to maintain.

4 Design Patterns

4.1 Introduction

When working on a particular problem, it is unusual to tackle it by inventing a new solution that is completely dissimilar from the existing ones. One often recalls a similar problem and reuses the essence of its solution to solve the new problem. This kind of thinking in problem solving is common to many different domains, such as software engineering.

Design patterns are important building blocks for designing and modelling applications on all platforms. Design patterns help us understand, discuss and reuse applications on a specific platform. The most commonly stated reasons for studying patterns are; reuse of solutions and establishment of common terminology. By reusing already established designs, the developer gets a head start on the problem and avoids common mistakes. The benefit of learning from the experience of others results in that he does not have to reinvent solutions for commonly recurring problems. The other reason for using patterns is that common terminology brings a common base of vocabulary and viewpoint of the problem for the developers. It provides a common point of reference during the analysis and design phase of a project.

The design patterns are divided into three types: creational, structural, and behavioural. We'll be looking at two patterns from each category and present these in a C# point of view. To fully understand design patterns, knowledge of the object oriented paradigm and some familiarity with the Unified Modelling Language (UML) is required.

4.2 Creational Pattern

Creational pattern as the name implies are concerned with the creation of object. The patterns help you build a part of an application that hides how an object is created and composed from the representation of the object. The only information known of the object is its interface. Creational patterns can be divided in class creational patterns and object creational patterns. The difference lies in that the class creational patterns use inheritance to instantiate a class, while object creational patterns delegate the instantiation to another object.

All creational patterns hide which concrete class is used by the system and, further more they hide how an object is created and put together.

4.2.1 Singleton Pattern

This pattern ensures that there exists only one instance of a class with a global access point to it. There are different situations where it is desired to have a single instance of a class for example a printer spooler; it would be most unfortunate to have several printer spoolers. Another example is when you need a single point of access to a database. C# has the mechanisms to implement the singleton de-

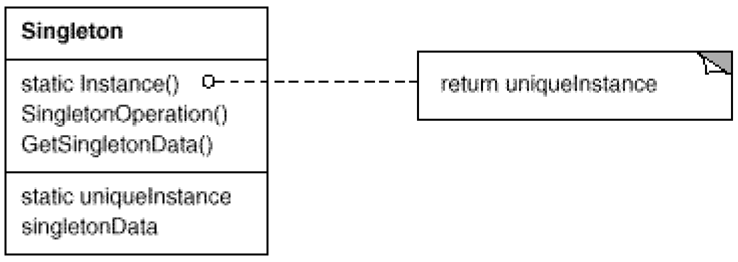

Figure 4: The structure of the Singleton pattern.

sign pattern. The example illustrates the implementation of a printer spooler. To instantiate a single instance of the class spooler we use a static variable.

```
public class Spooler
{
   private static Spooler spoolerInstance = null;
   private Spooler()
   {
   }

   public static Spooler getSpooler()
   {
      if(spoolerInstance != null)
         return spoolerInstance;
      else
      {
         spoolerInstance = new Spooler();
         return spoolerInstance;
      }
   }
}
```

When the spooler is created the value of the variable instance_flag is changed. Since the instance_flag is a static class variable there can only be one instance_flag, thus we have ensured that a single instance of a printer spooler exists. The spooler is created in the getSpooler method instead of the constructor. Note that the constructor is private, so any attempt to access the Spooler constructor will fail. How does C# facilitate a global access point for the printer spooler class? By defining a static class method. Static methods can only be called from the classes and not the class instances.

For example if we add a static instance, queueThis(string filName), to the Spooler class.

```
public class Spooler
{
   private static Spooler spoolerInstance = null;
   private string [] queueList;
   private spooler()
   {
   }

   public static Spooler getSpooler()
   {
      if(spoolerInstance != null)
         return spoolerInstance;
      else
      {
         spoolerInstance = new Spooler();
         return spoolerInstance;
      }
   }

   public static Spooler queueThis(string filename)
   {
      // Code to add filename to the queue
   }
}
```

And we have, by defining the queueThis method, created a global access point to the printer Spooler. The singleton example was from [2].

4.2.2 Abstract Factory

Another creational pattern is abstract factory. Abstract factory pattern is useful when you want to create families of related or dependent objects and not reveal their concrete classes according to [1].

The structural composition of abstract factory patter is shown in the figure 5. As you can see from the figure you cannot only add and remove objects (Abstract-

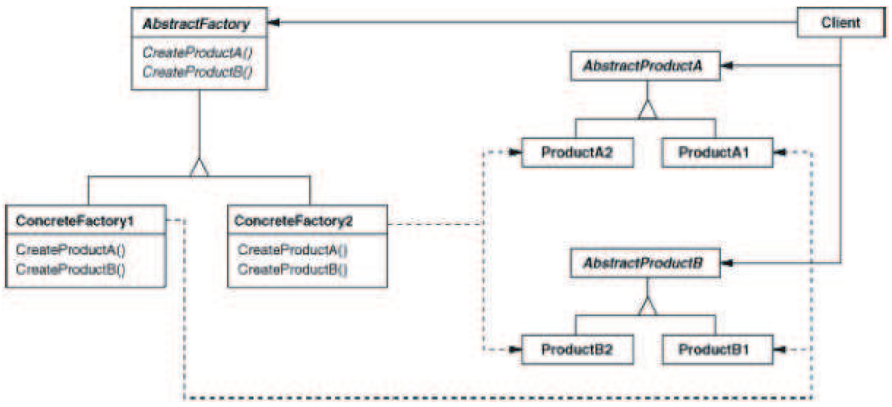

Figure 5: The structure of the Abstract factory pattern.

Product) to an existing factory (ConcreteFactory), you can also add and remove the factories (ConcreteFactory). Abstract factory pattern reveal only the interfaces of the products and hide how they are created. This design pattern has a higher level of abstraction then the Factory method, which is another creational pattern.

With C# you have the means to implement a design made with the abstract factory pattern, and the best way to show this is by an example.

Imagine you are designing software that builds aeroplanes. First you need to define the AbstractFactory and the AbstractProducts classes, we call then aeroplaneFactory and aeroplane.

```
abstract class aeroplane
{
   public abstract string type { get; }
}

abstract class aeroplaneFactory
{
   public abstract aeroplane getAeroplane();
}
```

Now we define the concreteFactory for the aeroplaneFactory class and the product class for the aeroplane class.

```
class aeroplaneProductMustangP51 : aeroplane
```

```
{
   string _type = "mustang";

   public override string type
   {
      get { return _type; }
   }
}

class concreteNorthAmericaFactory : aeroplanFactory
{
   public override aeroplane getAeroplane ()
   {
      return new aeroplaneProductMustangP51();
   }
}
```

Now that we have the factory and the product we are ready to implement the classes and build our aeroplane. But first we need a client and a main class. For our client class we implement the testpilot class.

```
class testpilot
{
   public void askformodel(aeroplaneFactory factory)
   {
      aeroplane aircraft = factory.getAeroplane();
      Console.WriteLine("Planemodel {0}",aircraft.type);
   }
}

class MainClass
{
   static void Main (string[] args)
   {
      aeroplaneFactory factory = new concreteNorthAmericaFactory();
      new testpilot().askformodel(factory);
   }
}
```

Now we are going to add another factory and another type of aeroplane.

```
class aeroplaneProductFW190 : aeroplane
{
   string _type = "Foker-Wulf 190";
   public override string type
   {
      get { return _type; }
   }
}

class concreteFokerWulfFactory :aeroplanesFactory
{
   public override aeroplane getAeroplane ()
   {
      return new aeroplaneProductFW190();
   }
}
```

We just need to modify the main class a little to be able to use the new factory. Note that we do not change the client class (testpilot) the interface remains the same.

```
class MainClass
{
   static void Main (string [] args)
   {
      aeroplaneFactory factory = null;
      if (args.length > 0 && args[0] == "P51")
         factory = new aeroplaneNorthAmericaFactory();
      else if ( args.length > 0 && args[0] == "FW190")
         factory = new aeroplaneFokerWulfFactory();

      new testpilot().askformodel(factory);
   }
}
```

This simple example shows how you can add a new factory to you system but you can see it is easy to add new products as well. You just add new aeroplanes and change in the code how to choose the new products (aeroplanes).

4.3 Structural Pattern

Structural patterns suggest ways to put together existing objects into complex structures, in order to achieve new functionality. *Class* structural patterns use inheritance to compose interfaces and implementation, whereas in *object* structural patterns an object can reside inside another object. In this chapter we will look more closely at the patterns Adapter and Composite. In the chapter about Adapter both *class* and *object* approaches of implementing the design pattern will be described.

4.3.1 Adapter

The Adapter pattern sometimes also known as wrapper is used whenever we want to change the interface of an object into another, desired interface. This is useful in the following scenario: An application accepts a set of functions or a certain interface to be implemented. The same application (or class) needs to communicate with an object that provides the functionality we are looking for but does not support the exact interface required by the application.

Consider for example the following [1]: A graphics application uses an abstract *Shape class.* Each graphical shape such as circle, triangle and polygon is a subclass of *Shape* and implements it's own drawing function. The drawing function for a *TextShape* object however might be more difficult to implement. Let's say that we find an off the shelf object called *TextView* that provides us with the functionality we are looking for. *TextView* however is not compatible with the *Shape* class.

There are mainly two way of applying the adapter pattern to make the *TextView* class work with our application: class and object [1]. Class means that we inherit *Shapes* interface and *TextViews* implementation. Object means that we hold an instance of *TextView* inside *TextShape* and implement *Shapes* interface in terms of *TextView.* figure 6 and figure 7 illustrate a diagram for the two methods respectively. Both approaches are fairly easy to implement in C#. Let's look at an example on each approach. First looking at the Object approach, we assume that the

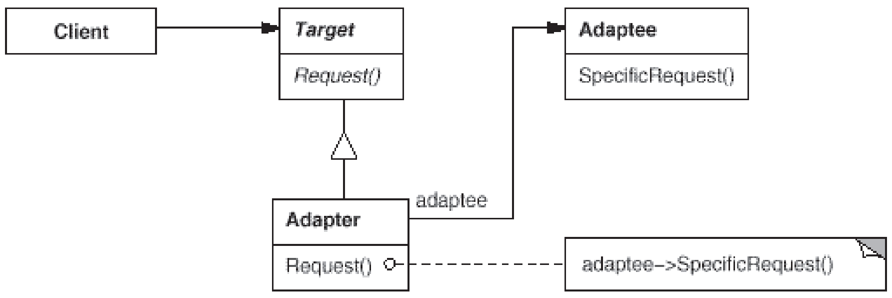

Figure 6: The structure of the Adapter pattern, the object approach.

DisplayText of the *TextView* class in the following example corresponds to the draw operation in *Shape.* Notice how the *TextShape* class takes an instance of a *TextView* object as an argument in its constructor and later overrides the *Draw* method of the *Shape* class by calling the *DisplayText* method of *TextView* and thereby acts as an adapter for *TextView.*

```
class Shape
{
   ...
   public virtual void Draw()
   {
```

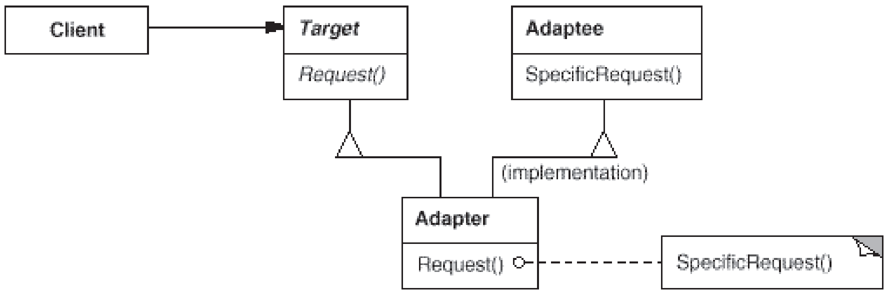

Figure 7: The structure of the Adapter pattern, the class approach.

```
   }
   ...
}

class TextView
{
   ...
   public DisplayText()
   {
      ...
   }
   ...
}

class TextShape: Shape
{
   private TextView Text;
   public TextShape(TextView T)
   {
      this.Text = T;
   }

   public override void Draw()
   {
      this.Text.DisplayText();
   }
   ...
}
```

An alternative design would be to view the drawing capabilities of each object as a behaviour that we want all of our graphic objects to display. In other words our application demands that each shape implements a *Draw* method (along with any other methods we might need). Based on our previous discussion on interfaces we can present an interface, perhaps called *IDisplay* containing the methods necessary to reflect this particular behaviour. This way the functionality in the *TextView* object could be used if we let *TextShape* inherit from *TextView*. We can then add the methods we need to *TextShape* by letting it implement the *IDisplay* interface. This is the class approach of implementing the adapter pattern and would look something like the following in code:

```
interface IDisplay
{
   void Draw();
}

class TextView
{
   ...
   public DisplayText()
   {
      ...
   }
   ...
}

class TextShape: TextView, IDisplay
{
   ...
   public Draw()
   {
      this.DisplayText();
   }
   ...
}
```

Notice that in the class approach, introducing an interface is necessary since C# does not support multiple inheritance. Which approach is more suitable to use depends on the situation. In this case if we already have built our application around a class hierarchy with *Shape* as an abstract base class, the first approach is probably more suitable. In other situations it might be a better idea to introduce an interface. Ultimately it is up to the developer to decide where and when to use each approach.

4.3.2 Composite

The composite pattern is designed to deal with situations where a certain object can either be viewed individually, or as a placeholder for a collection of objects of the same type. Consider for example a graphics application that will let you group a collection of simple graphic objects, such as lines and texts, into a picture. A picture which itself is a graphic object can in turn contain other pictures. The composite patterns allows us to treat all graphic objects, independent of whether they are complex (pictures) or simple (lines), in an uniform fashion. This can be achieved by introducing an abstract base class that represent both the simple and complex type. This abstract type contains methods that are shared between the

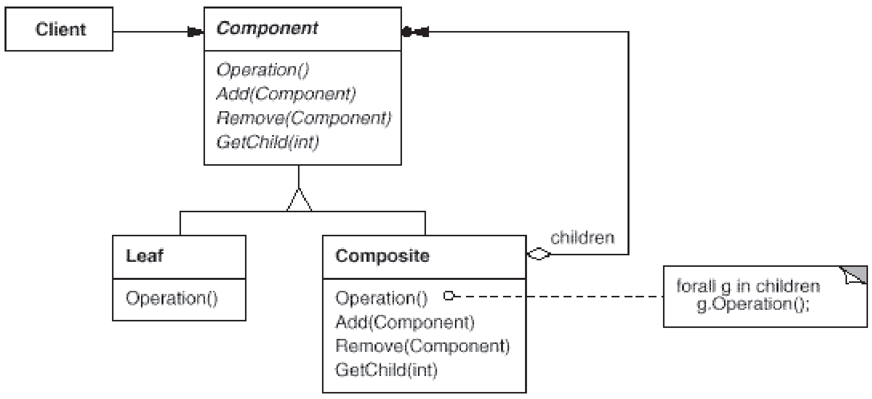

Figure 8: The structure of the Composite pattern.

simple and complex types, and also methods that will let the client access and perform operations on the complex types children (the objects it contains) [1]. The diagram in figure 8 illustrates this.

The simplest analogy for describing a composite object is a tree. Each node object should have the same set of methods such as GetValue, AddChild, RemoveChild and GetChild. Some nodes are of a simple type and cannot have any children. They are therefore always leaves. In that case we might want to raise an error upon calling the AddChild method. This can be easily done in C# by creating an exception and throwing it. The GetChild method could return an ArrayList object (see 3.3.2) as a return value, which will be empty for a leaf object. This can easily be verified by checking the count property that will be 0 [2].

In the graphics example a picture is a typical node whereas a line or a text object is a leaf and cannot have any children. This is illustrated in figure 9. Let us now look at how we can implement a *picture* and a *line* class. We will assume for simplicity that each class contains a draw method, and that the draw method of the picture class simply calls the draw methods of all of its children. We will start by defining an abstract *graphics* class that will contain a draw method, and the necessary functions for accessing and manipulating children objects.

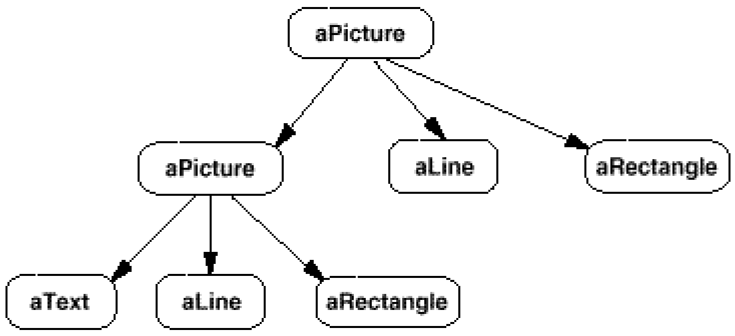

Figure 9: The graphics example illustrated as a tree.

```
class graphic
{
   public virtual void draw()
   {
   }

   // A list containing the children
   protected ArrayList children;
   public bool isLeaf()
   {
      // Simply check the count property
      return children.Count == 0;
   }

   public virtual void AddChild(graphic)
   {
   }

   public virtual graphic GetChild(int i)
   {
   }
}
```

We now derive the *picture* and the *line* classes from the *graphic* class. First the simple line class:

```
class line : graphic {
   public override void draw()
   {
      // Implements its own draw function
   }
   public override void AddChild(graphic g)
   {
      // This is a simple type object
      // Raise an exception
      throw new Exception("No children in this class");
   }

   public override graphic GetChild(int i)
   {
      // Simple type, no children to return
      return null;
   }
}
```

And now the complex picture class:

```
class picture: graphic
{
   public override void draw()
   {
      // Call the draw function for all children
      foreach (graphic g in this.children)
         g.draw();
   }

   public override void AddChild(graphic g)
   {
      this.children.add(g);
   }

   public override graphic GetChild(int i)
   {
      return this.children[i];
   }
}
```

Other simple graphic objects such as a text object could be built in the same way as the line class defined above. It is also worth mentioning that an indexer could be used instead of implementing the AddChild and GetChild methods.

4.4 Behavioural Pattern

Behavioural patterns are most specifically concerned with communication between objects. These patterns describe communication between objects in your system and how the flow is controlled in a complex program. They move the focus away from the flow of control and let the developer concentrate on the way objects are interconnected.

4.4.1 Observer

The Observer pattern lets one part of a system know when an event takes place in another. According to Gamma et. al [1] it is a one-to-many dependency between objects; if one-object changes state the other dependent objects will automatically be notified and updated.

The most obvious use of the Observer pattern is when a change of one object requires changing of an unknown number of objects. The notifying object should not have knowledge about who these objects are. The objects aren't supposed to be tightly coupled; the Observer pattern is an example of a decoupling pattern. It could also be applicable if the abstraction has two aspects and you would like to encapsulate these aspects in separate objects. This will make the objects more reusable and allow you to change them independently.

The following class diagram from Gamma et. al [1], figure 10, shows the relationship between the Subject and the Observer. The Subject may have one or more Observers, and it provides an interface to attaching and detaching the observer object at run time. The observer provides an update interface to receive signals from the subject. The ConcreteSubject stores the subject state interested by the observers, and it sends notification to its observers. The ConcreteObserver maintains reference to a ConcreteSubject, and it implements an update operation. When designing and implementing the Observer pattern in C# we take advantage

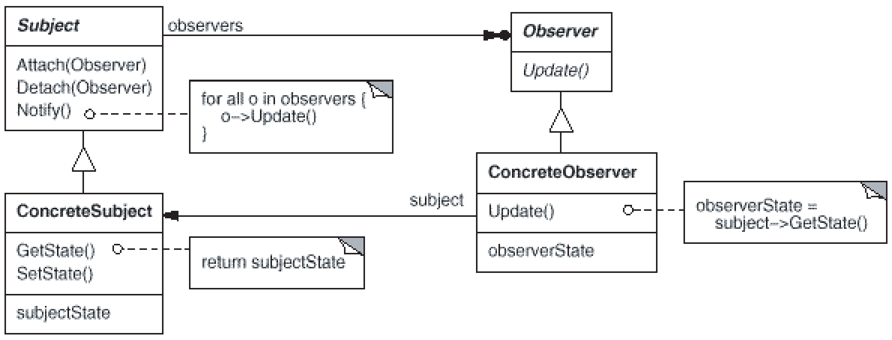

Figure 10: The structure of the Observer pattern.

of the delegates and events that where introduced in section 3.7. These provide a powerful way of implementing this pattern without developing specific types. The delegates permitting anonymous invocation of the bound method and the events help to expose state changes to interested objects at run time. The delegates and events are in fact first class members of the Common Language Runtime (CLR);

the foundation of this pattern is incorporated into the core of the .NET Framework. The Framework Class Library (FCL) makes extensive use of the Observer pattern throughout its structure.

With figure 10 in mind, the subject is the class declaring the event. The subject class doesn't have to implement a given interface or to extend a base class. It just needs to expose an event. The observer must create a specific delegate instance and register this delegate with the subject's event. It must use a delegate instance of the type specified by the event declaration otherwise the registration will fail. During the creation of the delegate instance, the name of the method (instance or static) is passed by the observer that will be notified by the subject to the delegate. Once the delegate is bound to the method it may be registered with the subject's event as well as it can be unregistered from the event. Subjects provide notification to observers by invocation of the event.

The following example is from Purdy D. et. al [4] and shows how to use delegates and events in the Observer pattern. The class, Stock, declare a delegate and an event. The instance variable askPrice fires an event when its value is set.

```
public class Stock
{
   public delegate void AskPriceDelegate(object aPrice);
   public event AskPriceDelegate AskPriceChanged;

   object askPrice;
   public object AskPrice {
      set
      {
         askPrice = value;
         AskPriceChanged(askPrice);
      }
   }
}
```

The StockDisplay class represents the user interface in the application.

```
public class StockDisplay
{
   public void AskPriceChanged(object aPrice) {
      Console.Write("The new ask price is:" + aPrice + "\r\n");
   }
}
```

The main class first creates a new display and a stock instance. Then it creates a new delegate and binds it to the observer's AskPriceChanged method. After that, the delegate is added to the event. Now when we use the set property of the stock class an event is fired every time. The delegate captures the event and executes the AskPriceChanged method, which in this case outputs the price in the console window. Before we quit the application, we remove the delegate from the event.

```
public class MainClass
{
   public static void Main()
   {
      StockDisplay stockDisplay = new StockDisplay();
      Stock stock = new Stock();
      Stock.AskPriceDelegate aDelegate = new
         Stock.AskPriceDelegate(stockDisplay.AskPriceChanged);

      stock.AskPriceChanged += aDelegate;

      for(int looper = 0; looper < 100; looper++)
         stock.AskPrice = looper;

      stock.AskPriceChanged -= aDelegate;
   }
}
```

4.4.2 Strategy

The Strategy pattern defines a family of algorithms, encapsulates the related algorithms and makes them interchangeable. This allows the selection of algorithm to vary independently from clients that use it and allows it to vary over time [4].

An application that requires a specific service or function and that has several ways of executing that function is a candidate for the Strategy pattern. The choice of proper algorithms is based upon user selection or computational efficiency. The client program could tell a driver module (context) which of these strategies to use and then tell it to carry out the operation. There are a number of cases in applications where we would like to do the same thing in several different ways like; compress files using different algorithms or save files in different formats.

The construction behind the Strategy pattern is to encapsulate the number of strategies in a single module and provide an uncomplicated interface to allow the clients to choose between these strategies. If you have several different behaviours that you want an object to perform, it is much simpler to keep track of them if each behaviour is a separate class, instead of the most common approach of putting them in one method. This is illustrated in figure 11 from Gamma et. al [1]. By doing this you can easily add, remove, or change the different behaviours, since each one is its own class. Each such behaviour or algorithm encapsulated into its own class is called a Strategy. The strategies do not need to be members of the same class hierarchy but they do have to implement the same interface [2]. The new language support for interfaces in C# comes in handy when implementing the

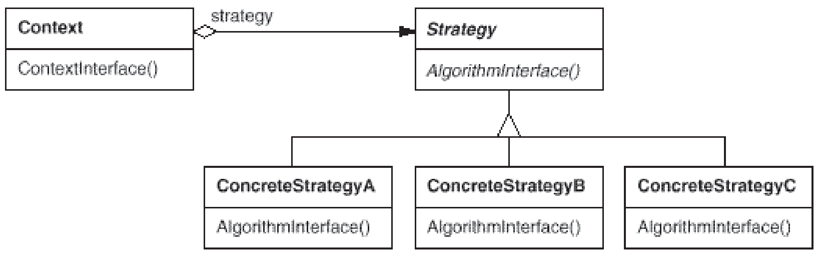

Figure 11: The structure of the Strategy pattern.

Strategy pattern. C++ programmers typically create interfaces by defining abstract classes with pure virtual methods. In C#, all interface members are public, and classes adhering to an interface must implement all methods in the interface.

The following code demonstrates the Strategy pattern, which encapsulates functionality in the form of an object. This basic example is based on the structure in figure 11.

```
interface Strategy
{
   void AlgorithmInterface();
}

public class ConcreteStrategyA : Strategy
{
   public void AlgorithmInterface()
   {
      Console.WriteLine(
         "Called ConcreteStrategyA.AlgorithmInterface()");
   }
}

public class Context
{
   private Strategy strategy;
   public Context(Strategy strategy)
   {
      this.strategy = strategy;
   }

   public void ContextInterface()
   {
      strategy.AlgorithmInterface();
   }
}
```

The encapsulation of the functionality allows clients to dynamically change algorithmic strategies by specifying the desired strategy.

```
Context c = new Context(new ConcreteStrategyA());
c.ContextInterface();
```

5 Summary

A design pattern is a description of a set of interacting classes that provide a framework for a solution to a generalized problem in a specific context or environment. In other words, a pattern suggests a solution to a particular problem or issue in object-oriented software development.

In today's software development, applications and systems are complex. These products require a great deal of flexibility in design and architecture to accommodate the ever-changing needs of clients and users during the product development and also after the product has been released. Design patterns assist in laying the foundation for a flexible architecture, which is the characteristic of every good object-oriented design.

C# together with .NET brings about many benefits, including the easy-to-use object model, the garbage collection mechanism for automatically cleaning up resources, and far improved libraries covering areas ranging from Windows GUI support to data access and generating web pages. The .NET framework insures that enough information is included in the compiled library files (the assemblies) and that your classes can be inherited from and used by other .NET-aware code without requiring access to your source files.

One of the primary goals of C# is safety; many of the problems that programmers can cause in C and C++ are avoided in C#. For example, a C# array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run-time, but the assumption is that the safety and increased productivity is worth the expense.

A problem with C# is its close interconnection with the .NET platform. Unfortunately using this program language makes your application equally platform dependent.

Generally the abstraction level of programming is higher in C# compared to C++. As usual the advantages gained by raising the abstraction level, come at the cost of lowered performance and less control.

Introducing C# in design patterns provides the programmer with a modern object-oriented programming language offering syntactic constructs and semantic support for concepts that map directly to notions in object-oriented design.

Design patterns suggest that you always program to an interface and not to an implementation. Then in all of your derived classes you have more freedom to implement methods that most suits your purposes. Since C# supports interface, this feature is useful when implementing patterns like adapter or strategy. Delegates and events are other well-suited features that make the design patterns cleaner.

6 References

1. Gamma E., et. al, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

2. Cooper J., *Introduction to Design Patterns in C#*, IBM T J Watson Research Center, 2002

3. Shalloway A., Trott J., *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison Wesley Professional, 2001

4. Purdy D., Richter J., *Exploring the Observer Design Pattern*,http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/obser verpattern.asp, 2002.

5. Liberty J., *Programming C#, 2nd Edition*, O'Reilly, ISBN: 0-596-00309-9, 2002

6. Archer T., *Inside C#*, Microsoft Press, ISBN: 0-735-61288-9, Washington, 2001

7. Thai T., Lam H., *.NET Framework Essentials, 2nd Edition*, O'Reilly, ISBN:0-596-00302-1, Sebastopol, 2002

8. Paul Kimmel, *Advanced C# Programming 1st Edition*, McGraw-Hill Osborne Media, ISBN 0-072-22417, September 4, 2002,

9. MSDN, *C# Language specification 17. Attributes*, Microsoft Corporation 2004 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfcsharpspec_17.asp

10. Doug Purdy, *Exploring the Factory design pattern*, Microsoft Corporation, February 2002, http://msdn.microsoft.com/library/default.asp?url=/libr ary/en-us/dnbda/html/factopattern.asp

11. MSDN Training, *Introduction to C# Programming for the Microsoft .Net Platform (Prerelease) Workbook*, Course nr 2124A, march 2001

12. MSDN Library, *Array Class*, http://msdn.microsoft.com/library/default.a sp?url=/library/en-us/cpref/html/frlrfSystemArrayClassTopic.asp.