

# Experiences from a C# project

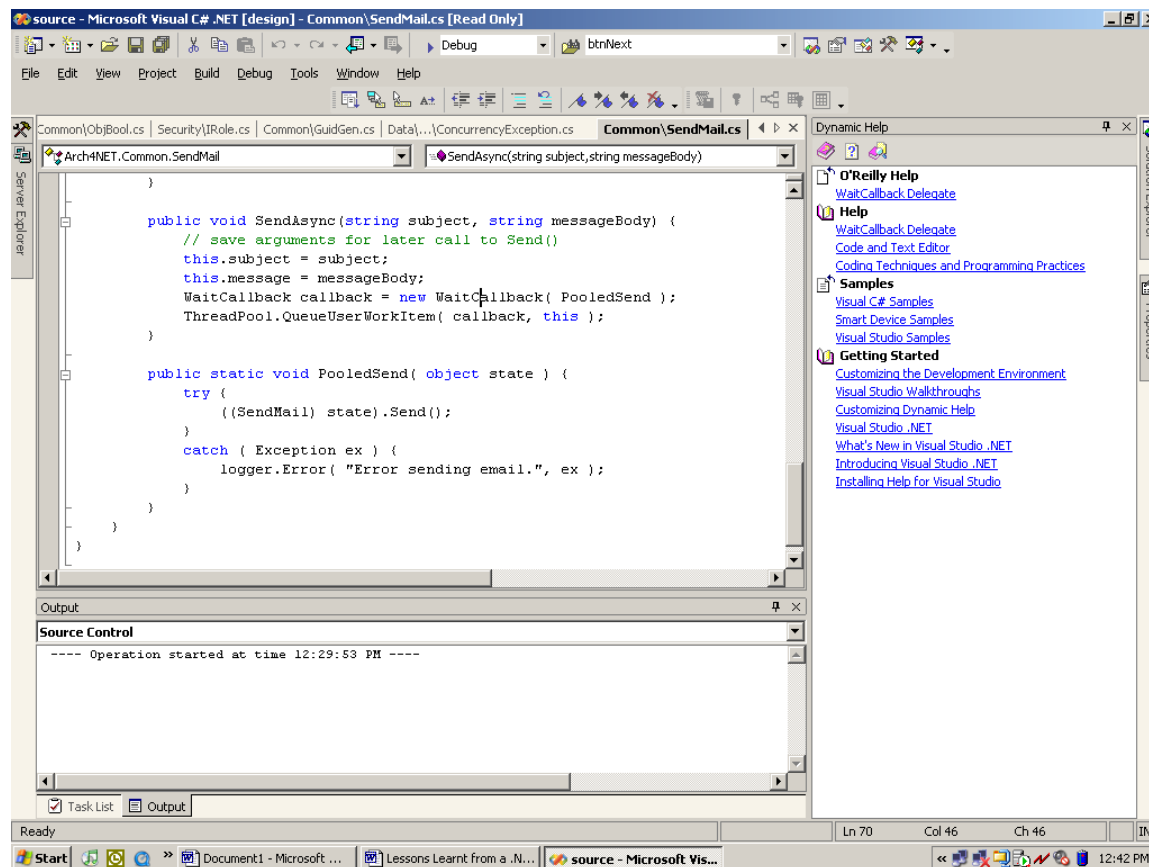
- A *java* developers perspective
- By Vibhu Srinivasan -

Experiences from a C# project.....	1
.....	1
.....	1
It isn't pretty but it rocks – Visual Studio.NET. ....	2
Look up and know the API before writing anything of own.....	4
Use Data grids with care.....	4
Use foreach loops as much as possible.....	5
Use one place to store session related information.....	5
Index Server – A hidden treasure .....	6
Realize the need for a layer based programming.....	6
Use internal classes whenever there is a need and be aware of its implications.....	6
Use Factory Methods to create domain objects.....	7
Use GUID and version number in database tables .....	8

This article is based on experiences on a year long C# .NET based web project that came to a successful completion recently. A number of lessons were learnt along the way and this article is an attempt to capture some of these ideas, before jumping into something else totally different. None of the ideas are new and nothing is alarming. Many problems would have been the same even if the language was C++, VB.NET or Java.



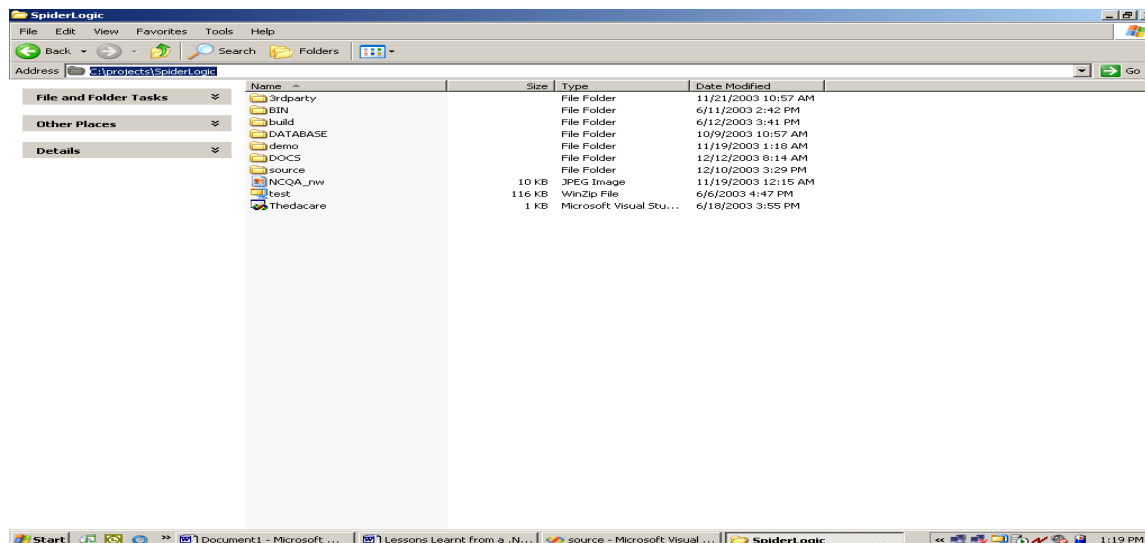




There is excellent use of available screen space. The auto-hide feature (the little pin on every window top right) enables screen's to hide and become visible as you hover the mouse over)

The initial release of Visual Studio did not ship with refactoring support appreciated by most OO programmers. Tools like IntelliJ IDEA for Java provide refactoring support out of the box. In the latest PDC it was announced that the next version of Visual Studio – Whidbey will fulfill this wish of many. There also is no integration with NUnit or other similar unit testing .NET frameworks at this time. However it is possible to write NUnit tests as an external program.

It takes a while to figure out the best way to work with the concepts of solution, project and then using that to create a tree in Visual Source Safe. The best way we found was to have independent projects and not put them into one solution. Create the structure first on the machine and then use Visual Studio itself to check the code into source code control the first time. The structure is shown below



Under the source directory various projects can be arranged . [Here is a good article](#) on MSDN that details ways to use Visual Studio with Source Safe.

### Look up and know the API before writing anything of own

There is a class for every possible reason in the .NET API. Anytime there is an urge to write some kind of a utility class like file upload over HTTP , make sure you look up the API before coding it all.. If you haven't done it already get the .NET API posters from Microsoft and paste them somewhere you can easily see them. Look up the poster before coding a solution. Apart from a good exercise to the tired eyes, the chances are high that you will find something for what you are trying to achieve. **The time taken is just figuring out where the magic bullet is!**

It is a good idea to install the MSDN CD that comes along Visual Studio installation. There are numerous examples given in VB.NET, C# and C ++. They may not all be the best way to implement a problem, but they do give enough leads to get going.

### Use Data grids with care.

Datagrid provide features like limiting the number of records in a page, providing the next previous page option etc. They can take datasets or any other collection as inputs. Paginating becomes very easy when working with datasets. However Viewstate has to be enabled to work with Datasets. If the size of collection is large then there are performance issues.

However not all information need to be loaded into the grid. Lazy loading can be done on a page by page basis. But this feature does not come out of the box.

As shown in the code below the page index changed method has to then know to initialize the grid with the correct items from the collection.

```
public void PageIndexChanged(object sender,
    DataGridPageChangedEventArgs e) {
    this.itemsDataGrid.CurrentPageIndex = e.NewPageIndex;

    IList results = (IList) this.Session["List Items"];
}
```

```

        // Add logic here to sort through this list and get only what is relevant to this page
        // or call the backend to retrieve records from 1-10, 11-20 and so on
        this.Initialise (results);
    }

```

Datasets can be bound to a datagrid. Instead of binding the entire dataset, we retrieved the data from the backend using regular collections and then created a Dataset in the UI layer to be used only by the UI.

- Unlike datasets the entire data for the query need not be loaded for a call.
- Domain objects can contain more information than the information that might be required by each row in the datagrid. Instead of binding the entire domain object to the grid and make the viewstate bigger, the UI datagrid can selectively have only those fields from the domain / value object relevant to the UI screen. The example code is shown below

```

DataSet dataset = new DataSet("ItemsDataSet");
DataTable table = new DataTable("Items");
table.Columns.Add("MonkeyName");
table.Columns.Add("MonkeyId");
dataset.Tables.Add(table);

foreach ( Monkey monkey in monkeyList ) {
    DataRow dataRow = table.NewRow();
    dataRow["Code Name"] = monkey.name;
    and so on..
}

```

The monkey object can have many attributes (Like all monkeys do !!), but the grid contains only what it needs

### Use foreach loops as much as possible

Wonder why Java never had this? Java 1.5 will have this foreach loop. This is another reason why competition makes products better.

Instead of using IEnumerator to iterate through a list, it is much more readable to use a foreach loop

<pre> IEnumerator iEnum= monkeyList.GetEnumerator(); while ( iEnum.MoveNext()) { Monkey monkey = (Monkey) iEnum.Current; } </pre>	<pre> foreach ( Monkey monkey in monkeyList ) { </pre>
---	--

### Use one place to store session related information

UI pages can have state information – e.g. shopping cart. Depending on the screen and state the screen is in, different types of information is often stored in the session object. Over a period of time it becomes a nightmare to manage what goes in session. Many times the same items are stored in the session under different keys.

---

In the simplest of scenario it is best to use some kind of a user context object that holds on all the objects that need to be put in the session object. Then there is a way to track the objects in the session. Instead of doing Session ["Attribute Name"] it is then easier to do context.AttributeName. The benefits to this typed approach are many.

- It is easy to write unit tests to verify what can go in a session. This helps to make sure that the session is not holding on to some object it is not supposed to be holding when the user is doing something in the system. Example – The session may be holding on to a list whose state might have changed in the database and the developer might have forgotten to refresh the session. A unit test in this scenario will avoid such mistakes.
- It is easy to log what is in the session by overloading the ToString method

### Index Server – A hidden treasure

Web systems have some kind of searching requirements. In most cases developers end up rolling out their own searching algorithms. The issue with these home grown algorithms is that they don't scale up when the demands of searching increase and over a period of time if the database grows in size the searching gets slower and slower. Issues like wild card searches, indexing etc need to be addressed. Searching is best done by search engines. However the best search engines turn out to be quite expensive and may not still be .NET compliant.

What came to our rescue was Microsoft Index server that comes installed with most of the Windows installations. Index server can be used to search both files in folders and also search databases. The only issue is these are two separate calls and if there is a match for a search in both these places, then there is no way to assign weights to the one that has a better match.

The SQL Server indexing can be triggered only on a SQL Server – Server installation and hence cannot be done on a typical developer machine. For searching PDF documents a [filter from adobe](#) has to be installed as index server by default does not provide this feature.

A search service gateway can delegate calls to index server and sql server and retrieve the results. [This article from Microsoft](#) points out how to search using index server.

### Realize the need for a layer based programming.

Although most of the business logic can be embedded into the code behind in an aspx page, it makes sense to separate the projects into logical layers. Web, domain / business and common tools at the least. These can also be physically separate into different namespaces and different Visual Studio projects.

Although it is tough to decide at what point in a project would you need to worry about these various layers, the benefits of layer based program far outweigh the reasons to not do so. It is much easier to refactor, reuse and maintain. Tests can be written at these different layers.

This is probably the biggest shift for VB programmers migrating to .NET. It is very easy to repeat the same style of programming.

Also it helps to define some kind of coding standard and the project team can follow the similar style ( Code will smell same ) . It should not matter who in the team wrote the code. It should all look alike.

### Use internal classes whenever there is a need and be aware of its implications

.NET introduced the concept of internal classes. When a class is declared as internal than all other classes in that distribution ( DLL ) can see it. It is different from the package level declaration in Java.

---

When a class is declared as package level only classes in that package can access that class. If another class was needing that class, the only resort was to make it public ( issue- The class is being exposed unnecessarily ) or the class had to be sub classed ( Issue - Inheriting unwanted behavior without a need ) .NET solved this with the notion of an internal class.

```

Namespace Animal
{
    internal class CodeMonkey{
        ..
    }
}

```

This makes CodeMonkey class available to namespaces other than Animal as long as they are in the same DLL/ Application.

In our case all classes that retrieve data are internal. Classes that have any kind of a security implementation like validating a user credentials are internal. It is best to start with a class as internal unless there is a need to make it protected or public.

#### Use Factory Methods to create domain objects

#### **Avoid exposing constructors of domain objects**

Consider a Account class for a bank that offers Savings and Checking accounts. The attributes being account number, Account Holder, Money balance, Minimum Required Balance. The checking account has no minimum balance but the savings has.

```

public class Account
{
    private object accountId;
    private AccountHolder accountHolder;
    private Money balance;
    private Money minBalance;
    private string type;

    public static string TYPE_CHECKING = "Checking";
    public static string TYPE_Savings = "Savings";

    public Account ( AccountHolder holder, Money balance, Money minBal, string type )
    {
        accountId = //create a account id here ( Encapsulate this logic )
    }
}

```

Client code for a checking account will be

```

new Account( account, holder, balance, null, Account.Checking );

```

In the above example the intent is not very clear reading the constructor. The rule that minimum balance is not required is not visible. So the programmer will end up Passing null and might result in a null pointer exception somewhere. Providing setter method does not make sure that clients of this object actually set the required attributes.

A better intent is conveyed by the code below

```

public class Account
{
    public static Account CreateSavingsAccount (AccountHolder holder, Money balance, Money
minBalance ){
        return new Account (holder,balance, minBalance, Account.TYPE_SAVINGS);
    }

    public static Account CreateCheckingAccount ( AccountHolder holder, Money balance){
        //add code here
    }

    private object AccountId;
    private AccountHolder accountHolder;
    private Money balance;
    private Money minBalance;
    private string type;

    private static string TYPE_CHECKING = "Checking";
    private static string TYPE_SAVINGS = "Savings";

    private Account(AccountHolder holder, Money balance, double minBal, string type)
    {
        accountId = //create a account id here ( Encapsulate this logic )

    }
}

```

Client code for a savings account will be

```

Account.Create( account, holder, balance, null, Account.Checking );
}

```

Notice now the method CreateSavingsAccount clearly conveys the intent . The client does not have to pass in any type and the way the object is constructed can be changed any time. Also since the constructor is private clients can never instantiate this object in an invalid state.

Another approach would have been to add two sub classes to account and call them Checking and Saving. But there is not much different between the two classes and adds up to the class library.

It is also possible to provide a Load method , internal to the distribution that loads data from the database, and includes others parameters like the primary key ( object id )

```

internal static Account LoadSavingsAccount ( object accountId, AccountHolder holder, Money balance,
Money minBalance ){
}

```

### Use GUID and version number in database tables

This might sound controversial. Many DBA's don't necessarily like guid ( Globally Unique Identifiers ) as primary keys to a table. Here are a few reasons why I feel GUID makes sense.

---

.NET provides a simple way to create a guid, `Guid guid = Guid.NewGuid();` That simple.  
In many systems as was the case in our project, there was a need to do a live propagation of data.

The options we had for propagation were

- Homegrown
- SQLServer synchronization.

We chose not to use SQL Server synchronization for the following reasons

- Someone will have to baby-sit the process and restart the propagation and there was no easy way to tell in what state the propagation failed
- It was cool to write one of our own.

Every domain object would automatically create a GUID, anytime the clients ( aspx pages ) create one. It was easy to do updates to the databases as a guid was unique and hence allowed us to quickly update a row without looking at other composite keys in the table.

Every table has a column called a version number. By comparing version numbers, it was easy to prevent concurrency issues with clients updating table after someone else had changed the row.

Note of Caution – DBA's are usually right too. Not every occasion might warrant guid and version number.

## **Contact**

Vibhu Srinivasan

[vibhu.srinivasan@gmail.com](mailto:vibhu.srinivasan@gmail.com)

---