



# .Net Code Conventions

## Standards Document

Prepared By

Eric Wroolie

For

Overpass Ltd

26 June 2006

## Table of Contents

### Table of Contents

Table of Contents.....	2
Document Information.....	3
Purpose .....	3
Version History .....	3
Distribution.....	3
Overview .....	4
Tools.....	5
Development .....	5
Testing .....	5
Source Control.....	5
Virtualisation .....	5
Naming Conventions and Standards .....	6
Indentation and Spacing .....	9
Good Programming practices .....	12
Architecture .....	17
ASP.NET .....	18
Comments .....	19
Exception Handling .....	20
Source Code Control.....	22
Deployment .....	23
Data Access .....	24
Testing.....	25
Code Reviews .....	26

## Document Information

### *Purpose*

All code produced by or on behalf of Overpass Ltd will adhere to certain code conventions tailored for each client. In the absence of a client's documented coding conventions, the conventions of the following document will be adhered to.

The code conventions used in this document adhere to the practices and documentation of several revered and established developers in the industry.

### *Version History*

Version	Author	Date	Comment
1.0	Eric Wroolie	26 June 2006	Initial Draft
1.1	Eric Wroolie	29 Sept 2006	Added Data Access Section to document
1.2	Eric Wroolie	15 August 2007	Added Tools section
1.3	Eric Wroolie	29 August 2007	Minor Changes for Corporate Client
1.4	Eric Wroolie	8 Sept 2009	Tools section brought up to date

### *Distribution*

Name	Title

## Overview

A coding standards document outlines how code must be written for a certain organisations so it can be easily maintained, or worked on together with, other members of a software development team. When standards are not adhered to or even defined, each developer will write code in different ways—making it difficult for other developers to read it.

Coding standards generally acknowledged in the IT industry as being a positive step in the progression of software development. However, in our experience, most organisations have little or no documented coding standards. Most legacy code in companies consists of a hodgepodge of development styles created by developers who have long since left the company. Among companies where we have seen an adoption of documented standards, it is often considered that the true mark of a professional developer is the ability to work in a team environment accroding to the standards that have been layed out.

This document is meant as a starting point for Overpass clients to create a coding standards plan. After this document is signed off or modified by all existing developers, it will be given to all new developers to join a software development team.

The standards and “best practices” listed in this document are not unique and were not invented by Overpass Ltd, they are are followed by several developers and organisations around the world. Much of this document was taken from other coding standards of successful IT companies and from the MSDN website.

This document covers development using C#, SQL Server, and the .Net platform.

## Tools

The following tools should be used during the development process:

### *Development*

1. **Visual Studio.Net 2008**  
Visual Studio.Net is the standard development environment from Asp.Net and C#.
2. **ASP.Net Ajax ([ajax.asp.net](http://ajax.asp.net))**  
ASP.Net Ajax is a freely-available download to add Microsoft's AJAX components to Asp.net.
3. **Notepad2 ([www.notepad2.com](http://www.notepad2.com))**  
Notepad2 is an opensource notepad substitute which makes debugging HTML output easier by providing colour-encoding and line numbers.
4. **ASP.Net MVC Framework ([mvc.asp.net](http://mvc.asp.net))**  
ASP.Net MVC Framework to be used for rapid technology development using OR/M and REST technologies.

### *Testing*

1. **Nunit ([www.nunit.org](http://www.nunit.org))**  
Nunit is the open-source unit test application that is used by the majority of developers. Visual Studio.Net Team Editions now incorporate unit testing, but this tool is the cheapest and still considered the best available.
2. **Selenium Recorder ([seleniumhq.org](http://seleniumhq.org))**  
Selenium is used with client-side code to create automated tests which can be run repeatedly.
3. **Fiddler 2 ([www.fiddler2.com](http://www.fiddler2.com))**  
Fiddler is a free tool which allows a developer to examine and modify http packets send from and to a web browser. This allows for very low level testing of what goes in and comes out of an Ajax HTTP request.

### *Source Control*

1. **TortoiseSVN ([tortoisesvn.tigris.org](http://tortoisesvn.tigris.org))**  
TortoiseSVN is widely considered to be one of the best clients for Subversion source control. It is open-source and there is plenty of documentation for it.

### *Virtualisation*

1. **Microsoft Virtual Machine 2007**  
Microsoft Virtual Machine 2007 is a free tool to create virtual machines on a development pc. This will allow a developer to test all applications on a clean machine without any referenced development tools.

## Naming Conventions and Standards

1. Use Pascal casing (first character of all words are upper case and other characters are lower case) for Class names

```
public class HelloWorld
{
    ...
}
```

2. Use Pascal casing for Method names

```
void SayHello(string name)
{
    ...
}
```

3. Use Camel casing (first character of all words except the first word is upper case) for variables and method parameters

```
int totalCount = 0;
void SayHello(string name)
{
    string fullMessage = "Hello " + name;
    ...
}
```

4. Use the prefix "I" with Camel Casing for interfaces ( Example: **IEntity** )
5. Do not use Hungarian notation to name variables.

In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m\_ as prefix for member variables. Eg:

```
string m_sName;
int nAge;
```

However, in .NET coding standards, this is not recommended. Usage of data type and m\_ to represent member variables should not be used. All variables should use camel casing.

6. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

Good:

```
string address
int salary
```

Not Good:

```
string nam
string addr
int sal
```

7. Do not use single character variable names like *i*, *n*, *s* etc. Use names like *index*, *temp*

One exception in this case would be variables used for iterations in loops:

```

for ( int i = 0; i < count; i++ )
{
    ...
}

```

8. Do not use underscores (\_) for local variable names.
9. All member variables must be prefixed with underscore (\_) so that they can be identified from other local variables.
10. Do not use variable names that resemble keywords.
11. Prefix `boolean` variables, properties and methods with "is" or similar prefixes.

Ex: `private bool _isFinished`

12. Namespace names should follow the standard pattern

```
<company name>.<product name>.<top level module>.<bottom level module>
```

13. Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables.

Some examples are:

Control	Prefix	Example
Label	lbl	lblSurname
TextBox	txt	txtSurname
DataGrid	dg	dgResults
Button	btn	btnSave
ImageButton	ibtn	ibtnSave
Hyperlink	lnk	lnkHomePage
DropDownList	ddl	ddlCompany
ListBox	lst	lstCompany
DataList	dlist	dlistAddress
Repeater	rep	repSection
Checkbox	chk	chkMailList
CheckBoxList	chk	chkAddress
RadioButton	rdo	rdoSex
RadioButtonList	rdo	rdoAgeGroup
Image	img	imgLogo
Panel	pan	panSection
PlaceHolder	plh	plhHeader
Calender	cal	calMyDate
Adrotator	adr	adrBanner
Table	tbl	tblResults
[All] Validators	val	valCreditCardNumber
ValidationSummary	vals	valsErrors

14. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs

15. Use Pascal Case for file names.

## Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.
2. Comments should be in the same level as the code (use the same level of indentation).

Good:

```
// Format a message and display

string fullMessage = "Hello " + name;
DateTime currentTime = DateTime.Now;
string message = fullMessage + ", the time is : " +
currentTime.ToShortTimeString();
MessageBox.Show ( message );
```

Not Good:

```
// Format a message and display
    string fullMessage = "Hello " + name;
    DateTime currentTime = DateTime.Now;
    string message = fullMessage + ", the time is : " +
    currentTime.ToShortTimeString();
    MessageBox.Show ( message );
```

3. Curly braces ( {} ) should be in the same level as the code outside the braces.

```
if ( ... )
{
    // Do something
    // ...
    return false;
}
```

4. Use one blank line to separate logical groups of code.

Good:

```
bool SayHello ( string name )
{
    string fullMessage = "Hello " + name;
    DateTime currentTime = DateTime.Now;

    string message = fullMessage + ", the time is : " +
currentTime.ToShortTimeString();

    MessageBox.Show ( message );

    if ( ... )
    {
        // Do something
        // ...

        return false;
    }

    return true;
}
```

Not Good:

```

bool SayHello (string name)
{
    string fullMessage = "Hello " + name;
    DateTime currentTime = DateTime.Now;
    string message = fullMessage + ", the time is : " +
currentTime.ToShortTimeString();
    MessageBox.Show ( message );
    if ( ... )
    {
        // Do something
        // ...
        return false;
    }
    return true;
}

```

5. There should be one and only one single blank line between each method inside the class.

6. The curly braces should be on a separate line and not in the same line as `if`, `for` etc.

Good:

```

if ( ... )
{
    // Do something
}

```

Not Good:

```

if ( ... ) {
    // Do something
}

```

7. Use a single space before and after each operator and brackets.

Good:

```

if ( showResult == true )
{
    for ( int i = 0; i < 10; i++ )
    {
        //
    }
}

```

Not Good:

```

if(showResult==true)
{
    for(int      i= 0;i<10;i++)
    {
        //
    }
}

```

8. Use `#region` to group related pieces of code together. If you use proper grouping using `#region`, the page should like this when all definitions are collapsed.

```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

9. Keep private member variables, properties and methods in the top of the file and public members in the bottom.

## Good Programming practices

1. Avoid writing very long methods. A method should typically have 1-25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.
2. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

Good:

```
void SavePhoneNumber ( string phoneNumber )
{
    // Save the phone number.
}
```

Not Good:

```
// This method will save the phone number.
void SaveDetails ( string phoneNumber )
{
    // Save the phone number.
}
```

3. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good:

```
// Save the address.
SaveAddress ( address );

// Send an email to the supervisor to inform that the address is updated.
SendEmail ( address, email );

void SaveAddress ( string address )
{
    // Save the address.
    // ...
}

void SendEmail ( string address, string email )
{
    // Send an email to inform the supervisor that the address is changed.
    // ...
}
```

Not Good:

```
// Save address and send an email to the supervisor to inform that
// the address is updated.
SaveAddress ( address, email );

void SaveAddress ( string address, string email )
{
    // Job 1.
    // Save the address.
    // ...

    // Job 2.
    // Send an email to inform the supervisor that the address is changed.
    // ...
}
```

4. Use the `c#` specific types (aliases), rather than the types defined in System namespace.

```
int age;    (not Int16)
string name; (not String)
object contactInfo; (not Object)
```

5. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

```
If ( memberType == eMemberTypes.Registered )
{
    // Registered user... do something..
}
else if ( memberType == eMemberTypes.Guest )
{
    // Guest user... do something..
}
else
{
    // Un expected user type. Throw an exception
    throw new Exception ("Un expected value " + memberType.ToString() +
        ".")

    // If we introduce a new user type in future, we can easily find
    // the problem here.
}
```

Not Good:

```
If ( memberType == eMemberTypes.Registered )
{
    // Registered user... do something..
}
else
{
    // Guest user... do something..

    // If we introduce another user type in future, this code will
    // fail and will not be noticed.
}
```

6. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

7. Do not hardcode strings. Use resource files.
8. Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.

```
if ( name.ToLower() == "john" )
{
    //...
}
```

9. Use `String.Empty` instead of `""`

Good:

```
If ( name == String.Empty )
{
```

```
// do something
}
```

Not Good:

```
If ( name == "" )
{
    // do something
}
```

10. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

11. Use `enum` wherever required. Do not use numbers or strings to indicate discrete values.

Good:

```
enum MailType
{
    Html,
    PlainText,
    Attachment
}

void SendMail (string message, MailType mailType)
{
    switch ( mailType )
    {
        case MailType.Html:
            // Do something
            break;
        case MailType.PlainText:
            // Do something
            break;
        case MailType.Attachment:
            // Do something
            break;
        default:
            // Do something
            break;
    }
}
```

Not Good:

```
void SendMail (string message, string mailType)
{
    switch ( mailType )
    {
        case "Html":
            // Do something
            break;
        case "PlainText":
            // Do something
            break;
        case "Attachment":
            // Do something
            break;
        default:
            // Do something
            break;
    }
}
```

12. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.
13. The event handler should not contain the code to perform the required action. Rather call another method from the event handler.
14. Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.
15. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
16. Never assume that your code will run from drive "C:". You may never know, some users may run it from network or from a "Z:".
17. In the application start up, do some kind of "self check" and ensure all required files and dependancies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.
18. If the required configuration file is not found, application should be able to create one with default values.
19. If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.
20. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
21. When displaying error messages, in addition to telling what is wrong, the message should also tell what should the user do to solve the problem. Instead of message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."
22. Show short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.
23. Do not have more than one class in a single file.
24. Have your own templates for each of the file types in Visual Studio. You can include your company name, copy right information etc in the template. You can view or edit the Visual Studio file templates in the folder `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ItemTemplatesCache\CSharp\1033`.
25. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.
26. Avoid public methods and properties, unless they really need to be accessed from outside the class. Use "internal" if they are accessed only within the same assembly.
27. Avoid passing too many parameters to a method. If you have more than 4-5 parameters, it is a good candidate to define a class or structure.
28. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the "count" rather than doing an additional check for "null".
29. Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.

30. Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.
16. Make sure you have a good logging class which can be configured to log errors, warning or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you trouble shoot a problem.
17. If you are opening database connections, sockets, file stream etc, always close them in the `finally` block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the `finally` block.
18. Declare variables as close as possible to where it is first used. Use one variable declaration per line.
19. Use `StringBuilder` class instead of `String` when you have to manipulate string objects in a loop. The `String` object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

Consider the following example:

```
public string ComposeMessage (string[] lines)
{
    string message = String.Empty;

    for (int i = 0; i < lines.Length; i++)
    {
        message += lines [i];
    }

    return message;
}
```

In the above example, it may look like we are just appending to the string object 'message'. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use `StringBuilder` class instead of `String` object.

See the example where the `String` object is replaced with `StringBuilder`.

```
public string ComposeMessage (string[] lines)
{
    StringBuilder message = new StringBuilder();

    for (int i = 0; i < lines.Length; i++)
    {
        message.Append( lines[i] );
    }

    return message.ToString();
}
```

## Architecture

1. Always use multi layer (N-Tier) architecture.
2. Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.
3. Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.
4. Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.
5. A middle-tier class library should never return UI objects or formatted HTML. All business objection and data access tiers must never be dependent on a particular calling object type. For example, data returned from a business object class should be usable in WPF Application, a Winforms application, or a web page with the client handling all display.

## ASP.NET

1. Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using `System.Web.HttpContext.Current.Session`
2. Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.
3. Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them
4. Session data should be stored in SQL Server (using the web.config file) and never held in memory. This way, it will be consistent across server farms.
5. As a general rule, all HTML elements should conform to XHTML 1.1 standards. This means closing `<br />` and `<img />` tags, etc. In Visual Studio, the XHTML Strict setting should be set. By using XHTML as a standard (instead of HTML 3.4 or HTML 4), applications will be better prepared for future browser changes.

## Comments

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.
2. Use `//` or `///` for comments. Avoid using `/* ... */`
3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
4. Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.
8. The bottom line is, write clean, readable code such a way that it doesn't need any comments to understand.
9. Perform spelling check on comments and also make sure proper grammar and punctuation is used.

## Exception Handling

1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.
2. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.
3. Always catch only the specific exception, not generic exception.

Good:

```
void ReadFromFile ( string fileName )
{
    try
    {
        // read from file.
    }
    catch (FileIOException ex)
    {
        // log error.
        // re-throw exception depending on your case.
        throw;
    }
}
```

Not Good:

```
void ReadFromFile ( string fileName )
{
    try
    {
        // read from file.
    }
    catch (Exception ex)
    {
        // Catching general exception is bad... we will never know whether
        // it was a file error or some other error.
        // Here you are hiding an exception.
        // In this case no one will ever know that an exception happened.

        return "";
    }
}
```

4. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.
5. When you re throw an exception, use the `throw` statement without specifying the original exception. This way, the original call stack is preserved.

Good:

```
catch
{
    // do whatever you want to handle the exception

    throw;
}
```

Not Good:

```
catch (Exception ex)
{
    // do whatever you want to handle the exception

    throw ex;
}
```

6. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exist in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.
7. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.
8. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class `SystemException`. Instead, inherit from `ApplicationException`.
9. If an application is to be hosted on a server that is not easily accessible, log all unexpected errors in a database table or on a web-accessible file directory. Never rely on Windows Event Viewer if you have not likely to have access to that server.

## Source Code Control

1. Unless another source control platform already exists at an organisation, we prefer to use Subversion to maintain code history. If Visual SourceSafe must be used, we prefer to use version 2005 with the merge option instead of document locking.
2. In the Subversion directories, the structure should be organised in the tradition trunk and branch layouts. Main development should be carried out in the directory "trunk" and a separate version directory should be created for each release.
3. All database objects should be scripted out into sql and stored in Subversion. Ideally, these objects should be created using SQL and not Enterprise Manager.

## Deployment

1. All applications should be clearly versioned for deployment. In desktop applications, this version number should be included in an “About <application>” menu option. In a web application, this version number should be available on the page or included in the meta tags of the page headers.

## Data Access

1. Connection strings should only listed in the ConnectionStrings node of the web.config file. Where possible, this data should be encrypted.
2. Stored Procedures should be used in all data access. In-line SQL should never be used within a business object or data access layer. A database administrator should be able to view all SQL by viewing stored procedures.
3. Stored Procedures should never include business logic. Stored Procs should only ever be used for generic CRUD activity (Create, Read, Update, and Delete) and nothing else. The only exception to this may be to check if a record already exists and deliver an error back to the calling application.
4. Business objects must not interact directly with the database and should instead use a Data Access object that connects with the database. This will aid in writing unit tests which test business logic and will allow use to change database platforms if necessary at a future date without re-writing an entire application.
5. Database engine-specific namespaces should not be references within UI or Business object layers but can be used in Data Access layers. For example, the business objects layer can use System.Data to interact with generic DataSet objects, but it must not interact with a SqlDataReader object from the System.Data.SqlClient namespace.
6. All database connections should be opened using the **Using** block. This will ensure that all connections are closed if an unexpected error occurs.

## Testing

1. Unit Tests should be created for each class while that class is being developed. The more tests that can be created, the better.
2. For usability testing, a test script should be created of everything that needs to be tested. A UAT tester can perform the random button clicking, but developers need a set script to check off for each release.
3. Where ever possible, use a GUI Interaction recorder like Selenium (see tools section below) for re-runnable tests.

## Code Reviews

1. Peer code reviews are encouraged. Whenever it is possible, another developer should look over the code that you have written. They may be able to point out some alternative ways of performing the actions you are performing or they may learn something from your code.
2. Code reviews should be performed by the lead developer at least once a week for about twenty minutes. Each line of code does not need to be scrutinised, but a general feel of the code quality should be looked at and advice given. This may need to be more performed more frequently for new developers on the project.