Ignyte Software
Microsoft .NET Platform

Coding Standards and Code Walkthrough Guidelines For Microsoft .NET Platform

Developer Role

Version 1.7
June 28, 2009

Revision History

Date	Version	Request #	Description	Author
02/03/2008	1.0		Initial Draft	Rob Wells
5/23/2008	1.1		Updated Content	Rip Dallas
7/1/2008	1.2		Updated Content	Burt Summar
8/28/2008	1.3		Minor updates	Mark Overstreet
10/30/2008	1.4		Minor Updates	Mark Overstreet
5/18/2009	1.5		Added database specific information regarding password encryption	Mark Overstreet
6/9/2009	1.6		Added clarifications for several items including business object expectations, database expectations and requirements for AJAX and CSLA.NET versions	Mark Overstreet
6/28/2009	1.7		Added more clarification.	Mark Overstreet

Sign-Off

Sign-off Level	Date	Name	Signature
Level 1			
Level 2			
Level 3			

Table of Contents

Principles of Construction

There are four principal concerns in construction. They are, in order of precedence; security, quality, reuse and compliance with standards. This document will ensure that developers will produce a quality product consistently.

General Guidelines

Application Architecture

- All projects (unless other otherwise approved) must implement a design phase prior to any coding being done. This design phase must include a prototype of the user interface (in Visual Studio), a UML diagram of the business layer and all business objects, and an ER diagram of the database. Each phase must be completed and signed off on by Ignyte before any other phase is started as this will eliminate a lot of re-work. Also, the screen prototyping phase must implement each and every screen the system will have and include enough hard coded data to demonstrate the screens functionality as well as enough code to move between screens. The screens must look identical to how they will look once deployed to a production server.
- The project must conform to a logical 3-tier design at a minimum and include a project for the user interface, the business layer, and the data layer. If the project includes webservice functionality, there should be a separate layer for that project as well. In the case of projects that do not contain a user interface, a test client (or test clients) should be added that allows us to test the all functionality. Also, if the solution contains references to third party DLLs these should be contained in a folder under the solution folder called 'OtherDLLs'. The projects and folders should be named as follows: Ignyte.*ProjectName*.UI(WebUI), Ignyte.*ProjectName*.BusinessLayer, Ignyte.*ProjectName*.DataLayer, Ignyte.*ProjectName*.Webservice, Ignyte.*ProjectName*.TestClient1.
- Each project should have downstream referencing only and the application absolutely should not contain any circular references. In most cases the User Interface will reference the Business Layer which will then reference the Data Layer. The Data Layer should never reference the Business Layer and the Business Layer should never reference the User Interface.
- The data layer must be built with Ignyte's Data Layer Generator unless some other alternative is approved. This generator will provide a basis for your data layer but WILL NOT produce all the code you will need. You will still be responsible for supporting transactions, rollbacks, commits etc as well as writing custom queries. All custom code should be placed in the inherited Table class, NOT in the base class as this is where the generator will always overwrite its code should you need to execute the Data Layer Generator multiple times.

- All projects must be done in Visual Studio 2008 Professional with Service Pack 1 while targeting the .NET 3.5 Framework using C# unless an alternative is approved ahead of time.
- All applications requiring a database should use SQL Server 2008 Standard Edition (unless an alternative is approved ahead of time)
- The business tier must utilize the CSLA.NET framework and it must implement all business logic appropriately using the constructs of this framework. Also, Business objects should **NEVER** expose concepts of the actual physical data layer. For example, if we are storing data in a relational database, you should never expose foreign keys as part of the public interface as this is a relational concept and should the backend change, it would affect any program consuming the business object. We should be able to swap out any persistence mechanism (e.g. XML file, SQL Server database, IMS database, etc) without affecting our user interface or any other consumer of our Business objects.

  The current version of the CSLA.NET framework is 3.6.2 and can be downloaded here (http://www.lhotka.net/files/csla36/cslacs-3.6.2-090322.zip)

- The application must utilize the latest version Infragistic's NetAdvantage UI components (NetAdvantage 2009 Vol. 1) when it will clearly provide value to the application. For example, if you need enhanced functionality the basic GridView component does not provide, use the Infragistic WebGrid.
- For reporting, you must use the latest version of Active Reports from Data Dynamics (ActiveReports for .Net Standard Edition 3.0). This is our reporting standard and no other components are acceptable unless an extreme circumstance dictates it in which case we must approve it.
- The application must **NOT** use the Microsoft Enterprise Blocks unless explicitly authorized.
- The application must **NOT** use any third party DLLs other than the ones specified by this document unless explicitly authorized.
- Each application must be self-contained and cannot rely on any outside databases or components or services unless explicitly approved ahead of time. For example, the ASP.NET Membership database tables cannot be used unless approved. Instead this information must be maintained in the application being built.
- Each application must provide custom logging and tracing functionality in order to assist in troubleshooting. Logging errors should include, at a minimum, writing a text file called '*Application Name Error Log'* to a '*Logs*' sub-directory of the application and the ability to email each error to one or more email addresses.
- All webservices must be built with WCF (.svc). Older ASMX services cannot be used. Webservices should contain NO business logic. Any and all logic should be contained in a middle-tier business object and called by the service.

- Web applications must support and be fully compatible with all current versions of the major browsers which include but are not limited to Internet Explorer, Safari, Chrome, Firefox and Opera.
- All web applications must fully support IIS Compression.
- All web applications must provide support for Viewstate Compression at the application level. This must be controlled via a web.config setting called 'EnableViewStateCompression'
- All web applications must provide support for Website Compression at the application. This must be controlled via a web.config setting called 'EnableWebsiteCompression'.
- All web applications should utilize AJAX completely and should use the AJAX libraries included in Visual Studio 2008 SP1. The AJAX Control Toolkit should be utilized when appropriate and you must use **version 3.0.30512** (http://ajaxcontroltoolkit.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=27326#DownloadId=68191) or a later version.
- All ASP.NET coding must be done in code-behind files and should not be directly in-line with the aspx page unless approval is given ahead of time.

Reusability

- Minimize code duplication.
- Be aware of and reuse existing controls and components wherever possible.
- Look for opportunity to create new reusable controls and components.
- Reusable code should be self-contained and should not have any external dependencies.
- Inherited object should be easy to extend.

Application Security

- Data fields should always be declared as private so that they are not directly accessible outside the class.
- Declare get/set accessors to allow access to the data fields.
- Passwords should always be encrypted or utilize a one-way hash
- Use proper industry standard programming techniques to minimize security risk including programming against SQL injection, buffer overflows, etc.
- Always use stored procedures. Dynamic SQL should not be used unless authorized on a case-by-case basis and if it is used, must protect against SQL injection.
- All applications must store passwords in the database using encryption. Also, the application must be able to read the password field and determine if the password is encrypted or un-encrypted and function properly. If the password read from the database table is un-encrypted, the password must then be

encrypted and stored back to the database. This will allow an administrator the ability to reset any password by merely typing a new password directly into the table in the database.

- All applications must allow users to retrieve their actual password (not reset the password) via email unless another method is approved ahead of time.

Class / Method Functionality

- Each class should be responsible for carrying out only a small set of related tasks, making it easy to inherit from and easy to test.
- Each method should be responsible for carrying out only a single task.
- Use Overloading functions for backward compatibility wherever feasible.
- Avoid using Optional Parameters in VB.Net.
- Avoid writing very long methods. A method should typically have 1 to 25 lines of code. If a method has more than 25 lines of code, you must consider re-factoring into separate methods.
- Method name should tell what it does. Do not use misleading names. Do not use abbreviations.

Business Rule Validation

All business rules must be implemented in their appropriate business object utilizing the CSLA.NET BrokenRules collection technique. In addition, some of these rules will need to be enforced again in the user interface. For example, a text box might limit the length of the last name field in the user interface in addition to checking this length in the business object because you want instant user feedback. However, the business rule must not be solely implemented in the user interface for several reasons. First, the business object might be directly used without ever utilizing the user interface. Second, in a web application the web-browser could have JavaScript disabled and any implemented business logic would be ineffective. In both cases, if the business rules were not implemented in the business object itself, the data could be invalid.

Data Validation

Data validation should be used to make sure data conforms to table schema before attempting to update the table. As mentioned above, you should be enforcing data validation business logic in the user interface as well as the business tier.

Presentation

- An application should have a consistent look and feel throughout.
- These guidelines may be decided on a case-by-case basis always keeping the customer's needs in mind.
- Design consideration should include: MDI/SDI interface, Label size and font, Label and control orientation, etc
- All buttons/fields should be symmetrical, aligned consistently, and consistent in size.
- When writing ASP.NET applications, we do not want to use pop-up windows. Instead, utilize the modal popup AJAX control (AJAX Control Toolkit) that utilizes a DIV tag instead or use your own DIV-based window.
- All aspx and html pages must be fully XHTML 1.1 compatible.
- ASP.NET pages should never implement Frames.

Other Guidelines

- Trim all leading and trailing spaces from each string field before persisting to the database, unless there is a valid reason not to.
- Do not hardcode numbers. Use constants instead.
- Do not hardcode strings. Use resource files or constants.
- Avoid using many member variables. Declare local variables and pass it to methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
- Use enumerations wherever required. Do not use numbers or strings to indicate discrete values.
- Do not make the member variables public or protected. Keep them private and expose public/protected *Properties*.
- Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
- In the application start up, do some kind of "self check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.
- If the required configuration file is not found, application should be able to create one with default values.
- If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.

- Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
- Release Candidate source code must always build without generating any errors or warnings.
- Communication is very important to us. Our office is officially open Monday – Friday 8:00AM – 4:00PM Eastern Standard Time and we expect all inquiries and questions from us to be answered within 1 business day unless you notify us ahead of time that this will be a problem (e.g. Holidays, etc). Must provide project updates every one or two days during the development process to make sure the requirements are understood.
- When coding is completed the project is not over. We will begin user acceptance testing and the project will not be considered completed until all bugs are reported and fixed. Please realize that this may take more than one iteration based on how many things get broken in each testing/fix phase. Also, you can expect that we will take between 5 and 21 business days to organize test cases and produce a feedback document for each release candidate you provide. **Therefore, we highly recommend that you TEST and RE-TEST the application before submitting it to us for User Acceptance Testing. Otherwise, do not be surprised if the application takes longer to complete. Also, if we find immediate errors that you should have found during testing, it will affect your rating because we cannot waste time and money organizing test teams only to find errors that happen immediately when we execute the first test case so make sure you TEST every scenario completely!!!!!**

Naming Conventions

A consistent naming pattern is one of the most important elements of predictability and discoverability in a managed class library. Widespread use and understanding of these naming guidelines should eliminate unclear code and make it easier for developers to understand shared code.

We will use mostly the following two approaches for our development purposes.

Pascal Case Notation

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized.

Examples: **BackColor, DataSet**

Camel Case Notation

The first word is all lower case. For all other words in the name, the first letter is capitalized.

Examples: **lastName, firstName**

Naming Guidelines

Type	Case Notation	Example
Private Class Level Variable	Camel case prefixed with "_".	_firstName, _lastName
Protected Variable	Camel case	firstName, lastName
Public Variable	Pascal case	FirstName, LastName
Local Variable	Camel case	socialSecurityNumber, numberOfItems
Namespace	Pascal case or uppercase for very small word or abbreviations.	System.Windows.Forms, System.Web.UI
Class	Pascal case	Customer, Order, InvoiceDetail
Interface	Pascal case prefixed with the letter "I".	Examples: IServiceProvider, IFormatable
Parameter	Camel case	employeeName, numberOfItems
Method/Function	Pascal case	RemoveAll (), GetFormattedPhone()
Property/Enumerations	Pascal case	BackColor, NumberOfItems
Event	Pascal case notation suffixed with "EventHandler".	MouseEventHandler

Constant	Uppercase with words separated by underscores	AP_WIN_MAX_WIDTH, AP_WIN_MIN_WIDTH
Exception	"ex"	Catch (Exception ex)
Custom Exception	*Classname*Exception	PageNotFoundException
Project	*Projectname.*Solution *Projectname*.BusinessLayer *Projectname*.Website *Projectname*.DataLayer	Ignyte.SecretGift.Solution Ignyte.SecretGift.BusinessLayer Ignyte.SecretGift.Website Ignyte.SecretGift.DataLayer
Physical Files (.cs, .vb, etc)	*ClassName*.cs	User.cs, Person.cs, Invoices.vb
Common DLLs	Strong name and place it in a common location or GAC	

***User Interface Control*:** Pascal case prefixed with the appropriate abbreviation as listed below.

Control	**Prefix**	**Control**	**Prefix**
Button	btn	NumericUpDown	nud
CheckBox	chk	Panel	pnl
ComboBox	cbx	PictureBox	pbx
Container	ctr	ProgressBar	prg
ContextMenu	ctm	RadioButton	rdb
DataColumn	dcol	RichTextBox	rtf
DataGrid	dgrid	SDI_Form	frm
DataGridDateTimePickerColumn	dgdtpc	Splitter	spl
DataGridTableStyle	dgts	SqlCommand	sqlcom
DataGridTextBoxColumn	dgtbc	SqlConnection	sqlcon
DataReader	dr	SqlDataAdapter	sqlda
DataRow	drow	StatusBar	stb

DataSet	dset	TabControl	tabctrl
DataTable	dtable	TabPage	tabpg
DateTimePicker	dtp	TextBox	tbx
Dialog	dlg	ToolBar	tbr
DialogResults	dlgr	ToolBarButton	tbb
ErrorProvider	erp	ToolTip	ttp
GroupBox	gbx	TrackBar	tkb
ImageList	iml	TreeView	tvw
Label	lbl	Timer	tmr
LinkLabel	lkl	UserControl	uc
ListBox	lbx	XmlDocument	xd
ListView	lvw		
Mainmenu	mnu		
MenuItem	mnui		
MDI-Frame	frame		
MDI-Sheet	sheet		

In addition to providing naming standards,

- Method names should be a Verb or Verb Phrase
- Class/Property/Variable names should be Noun or noun phrase. In business objects, boolean properties should have a Verb as the first part of the name such as IsActive, IsAvailable, CanTransfer etc.
- Parameter names should have a descriptive name
- Attributes should always suffix with "Attribute"
- Exceptions should always suffix with "Exception"
- EventHandler should specify two parameters sender and e. Sender should always be object type, e should have the event class state

Naming Guidelines for SQL

Item	Case Notation	Example

Database Files	Pascal case. ProjectName.mdf ProjectName_log.ldf	ASPXFileManager.mdf ASPXFileManager_log.ldf
Tables	Pascal case notation prefixed with a “T” and should be singular.	TAddress
User Generated Stored Procedures	Pascal case prefixed with “usp_”	usp_GetAgencyKey
Application Generated Stored Procedures	Pascal case prefixed with “sp_”	sp_GetAgencyKey
Report Stored Procedures	Pascal case prefixed with "rsp_"	rsp_Report1
Views	Pascal case notation prefixed with a “V”.	VPerson
Functions	Pascal case prefixed with “fn_”.	fn_GetUserNameByKey
Triggers	Pascal case prefixed with "tr_"	tr_UpdateCounty
Input Parameters	Pascal case prefixed with “@i_”.	@i_Key
Output Parameters	Pascal case prefixed with “@o_”.	@o_Key
Variables	Pascal case prefixed with “@”.	@Key
Jobs	Pascal case prefixed with “(system abbreviation)_”.	ECH_SendEmailNotifications
DTS Packages	Pascal case prefixed with “(system abbreviation)_”.	CPMS_Conversion
Primary Keys	Pascal case	Key
Foreign Keys	Pascal case	*TableName*Key
Unique Constraints	Pascal case (Unique*TableNameFieldName*)	*UniqueTUserEmail* *UniqueTCountryName* *UniqueTUserToRoleUserKeyRoleKey*

Namespaces

A namespace is a way to categorize your classes and types so as to avoid naming conflicts between your code and other developer's code. The recommended structure of a namespace is as follows:

CompanyName.ProjectName[.Feature][.Design].

Example: Ignyte.BracketManager

Feature and design are optional but useful in cases where you need to explicitly organize and separate functionality from the main project, particularly if the main project comprises multiple structures.

Examples: Ignyte.BracketManager.***Accounting***
Ignyte.BracketManager.Accounting.***GeneralLedger***

Assembly Naming

An assembly must be named exactly as the namespace of the project. If there are multiple namespaces in a project (which is strongly discouraged), then the primary namespace should be used.

Examples: Namespace → Ignyte.BracketManager.**BusinessLayer**
Assembly → Ignyte.BracketManager.**BusinessLayer.dll**

Code Formatting

Indentation

- Comments should be in the same level as the code.
- Use TAB for indentation. Do not use SPACES.
- Use one blank line to separate logical groups of code.
- There should be one and only one single blank line between each method inside the class.
- Use a single space before and after each operator and brackets.
- When an expression will not fit on a single line break it up according to these guidelines.

  1) Break after comma.
  2) Break after operator.
  3) Align the new line with the beginning of the expression in the previous line.

Example:

```
MethodCall(expr1, expr2,
           expr3, expr4, expr5);
```

Code Structure

- The source directory and sub-directories would directly correlate to the namespaces. However, the starting point is irrelevant but unless otherwise stated should be *drive_letter\projects\projectname\source\cs\company.projectname.solution*

Examples:
Namespace → Ignyte.BracketManager**.BusinessLayer**
Directory → *drive_letter\projects\BracketManager\source\cs\Ignyte.BracketManager.Solution\Ignyte.BracketManager.BusinessLayer*

Line Length

- Use common sense with line lengths.
- There are cases when it makes sense that a line should go past the visible IDE and force the use of scrollbars.
- However, these should be kept to a minimum and lines should be broken according to the guidelines above.

Code Regions

Use the following code region names in the following order whenever possible. Region names should have a space prior to the first character and a space after the last character to increase readability. Also, when all regions are collapsed you should have one blank line between each region.

- **Class Level Variables**
  Declare your private variables here. Remember to prefix your private class scope variables with an underscore ("_").
- **Constants**
  Declare any constants and follow the naming standards (e.g. DEFAULT_FILE_NAME)
- **Constructors / Destructors**
  Declare your constructors here along with destructors. Remember to only utilize destructors in special scenarios as they can cause performance issues.

- **Properties**
  Place your public property getters/setters here.
- **Methods (Public)**
  Place your public methods in this region.
- **Methods (Private)**
  Place your private methods in this region.
- **Events**
  Declare the event arguments classes and the events in this region.
- **Embedded Classes**
  Declare any internal classes you are going to use. However, this typical should not be done and in general, all classes should be placed in their own code file (class.cs)

Examples:

```
#region Class Level Variables
#endregion

#region Constants
#endregion

#region Constructors / Destructors
 #endregion

#region Properties
#endregion

 #region Methods (Public)
 #endregion

#region Methods (Private)
 #endregion

#region Events
 #endregion

#region Embedded Classes
#endregion
```

Code Commenting

Inline Comments

- 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.
- Comment each type, each non-public type member, and each region declaration.
- Use end-line comments only on variable declaration lines. End-line comments are comments that follow code on a single line.
- Do not create formatted blocks of asterisks that surround comments.
- Separate comments from comment delimiters (apostrophe) with one space.
- Begin the comment text with an uppercase letter.
- End the comment with a period.
- Include task-list keyword flags to enable comment filtering.
- Always apply C# comment blocks (///) to public, protected, and internal declarations.
- Use VB Comment Blocks('") to all class, method, property, event declarations
- Make sure all methods, properties and classes include our standard header and that all parameters are defined completely and accurately based on their purpose. **NOTE**: Parameters definitions such as (<param name="nodeToUpdate" type=”TreeNode”> Node to update</param>) must include the ‘type=’ attribute and which will not be included automatically by Visual Studio.
- All comments must be documented in English without spelling or grammatical errors.

Class Header Description

The following comment blocks should appear at the top of each class module developed (before the class or interface keyword). Using this header will convey pertinent information to other developers.

```
/// <summary>
/// Base class used when needing a screen that is divided into 4 sections and
/// includes a top, a bottom, a left and a right panel. The left and right panels are
/// divided by a splitter bar control.
/// </summary>
///
/// <remarks>
///
/// <CodeWalkthroughHistory>
/// Reviewed By          Date Reviewed          Checklist Status
/// Rob Wells            10/10/2004             Passed
/// </CodeWalkthroughHistory>
///
```

```
/// <RevisionHistory>
/// Author              Date          Description
/// Jeff Howard         10/1/2004     Created Class
/// Lisa Goins          10/5/2004     Added NodeSelected function
/// </RevisionHistory>
///
/// </remarks>
```

In the above example, replace /// with ‘’’(ticks) if you are coding in VB.Net

Function Header Description

The following comment block should appear at the beginning of each method definition. Using this header will convey pertinent information to other developers.

```
/// <summary>
/// Function to update the text
/// displayed on the active node
/// </summary>
///
/// <param name="nodeToUpdate" type=”TreeNode”> Node to update</param>
///
/// <returns>true if node was updated</returns>
///
/// <exception cref="System.Exception">Thrown when…</exception>
///
/// <remarks>
/// <CodeWalkthroughHistory>
/// Reviewed By         Date Reviewed       Checklist Status
/// Rob Wells           10/10/2004          Passed
/// </CodeWalkthroughHistory>
///
/// <RevisionHistory>
/// Author              Date          Description
/// Jeff Howard         10/1/2004     Created Class
/// Lisa Goins          10/5/2004     Added NodeSelected function
/// </RevisionHistory>
/// </remarks>
```

In the above example, replace /// with ‘’’(ticks) if you are coding in VB.Net

User Message Guidelines

The purpose of this section is to setup guidelines for displaying effective and consistent message to the users throughout the system. Displaying information using message box requires setting title, message, icon, buttons, and default button setting.

- MessageBox title should be descriptive of the action being performed, for example “Confirm Delete”, “File not found”. Never use “Information”, “Critical” and “Warning”
- MessageBox text should give the user a clear, non-technical explanation of the problem.
- MessageBox text should include steps or a clear explanation of how to prevent the problem from occurring again.
- MessageBox text should state the problem or goal first, then the solution.

Message Type	Icons	Buttons	Default Button
Information	Information	OK	OK
Question	Information	Yes & No	No
System Error	Error	OK	OK
Other	Warning	OK	OK

Variables

Declaration

- Declaring variable with proper scope. For example, do not declare variable globally if they are going to be used in one function.
- Declare variables at the top of the function, class, or sub.
- Use the simplest data type, list, or object required. For example, use Int over Long unless a 64 bit value will be stored.
- Declare member variables as private. Use properties to provide access to them with
- Public, Protected, Internal, or Friend 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.
- Only declare constants for simple types such as int and string.
- Always explicitly initialize arrays of classes, interfaces, delegates, objects, and strings using a for loop.

  Example:

```
int count =1;
Object refCount = count;          // implicitly boxed.
int newCount = (int)refCount;     // explicitly boxed.
```

- Floating point values should include at least one digit before the decimal place and one after. Example: totalPercent = 0.05;
- In C#, use the "@" prefix for string literals instead of escaped strings.

  Example:

```
//Bad
ConfigFile = "\\VIP.exe.config";

//Good
ConfigFile = @"\VIP.exe.config";
```

- Prefer string.Format() or stringBuilder over string concatenation.
- If doing string concatenation inside of a loop, always use stringBuilder.
- Do not compare strings to string.empty or "" to check for empty strings. Instead, compare by using string.length == 0.
- Avoid hidden string allocations within a loop. Use string.Compare() instead.

  Example:

```
// Bad
If (Name.ToLower() == "John")

//Good
If(String.Compare(Name,"John",true) == 0)
```

- Avoid boxing and unboxing of value-types. Use Generics if necessary.

Cleanup

- Destroy objects before exiting a function.
- Use Dispose method to destroy objects.
- Use the finally block to release resources.
- Avoid using GC.Collect

C# Error Handling

- Never declare an empty Catch block.
- Avoid nesting a try/catch within a catch block.
- Unhandled exceptions must be written to a log.
- Display the error message only once to the user. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.
- Catch the most specific exception possible. Order exception filters from most to least derived exception type.
- Use Constants for all application related error messages instead of hard coding.
- 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)
{
        throw(ex);
}

// Good
Catch (Exception ex)
{
        throw;
}
```

- Only use the finally block to release resources from a try statement.
- Always use validation to avoid exceptions. Avoid using Try..catch instead of If..Else logic

  Example:

```
// Bad
try
{
        Conn.Close()
}
Catch(Exception ex)
{
        //Do what ever
}

// Good
```

```
if (Conn.State != ConnectionState.Closed)
{
        Conn.Close();
}
```

- In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.

  For example, it makes sense to define a **FileNotFoundException** because the developer might decide to create the missing file. However, a **FileIOException** is not something that would typically be handled specifically in code.

- Do not expose privileged information in exception messages. Information such as paths on the local file system is considered privileged information. Malicious code could use this information to gather private user information from the computer.
- When throwing a new Exception always pass the Inner Exception in order to maintain the exception tree & inner call stack.
- Write your own custom exception classes, if required in your application. Do not derive your custom exceptions from the base class “SystemException”. Instead, inherit from “ApplicationException”.
- Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.

SQL Server Specific (SQL Server 2005 and Above)

Each field in every table of the database should be defined as not nullable unless there is specific reason to not use a default value. This is especially true for string fields because there is no need to have to check for both null and empty string values when coding. Obviously a table containing a field such as ‘DateTerminated’ would be a candidate for using a null value but something like a ‘OptionalDescription’ should be using a default empty string value instead of a null value.

Also, each table in the database should implement the template definition below to handle auditing information and non-business meaning primary keys (unless specifically indicated otherwise). In addition, remember that foreign keys should always be named consistently using this format: *TableName*Key.

Standard Auditing SQL

```
CREATE TABLE [dbo].[TTableName](
	[Key] [uniqueidentifier] ROWGUIDCOL  NOT NULL DEFAULT
(newsequentialid()),
	[TStamp] [datetime] NOT NULL DEFAULT (getdate()),
	[DateCreated] [datetime] NOT NULL DEFAULT (getdate()),
	[CreatedBy] [varchar](50) NOT NULL DEFAULT (suser_sname()),
	[LastUpdatedBy] [varchar](50) NULL,
	[Source] [varchar](50) NOT NULL DEFAULT (app_name()),
 CONSTRAINT [TTableName_PK] PRIMARY KEY CLUSTERED
(
	[Key] ASC
)

CREATE TRIGGER [dbo].[TTableName_RecordUserNameAndTimeStampOnUpdate]
ON [dbo].[TTableName]
   AFTER UPDATE
 AS
 BEGIN
   UPDATE [TTableName]
   SET LastUpdatedBy = SUSER_SNAME ()
	WHERE [TTableName].[key] in (select [key] from inserted)

   UPDATE [TTableName]
   SET TStamp= getdate()
	WHERE [TTableName].[key] in (select [key] from inserted)
 END
```

Standard Error Handling Template

```
-------------------------------------------------------------------------
-- Stored procedure that will select an existing row from the table 'TApplication'
-- based on the Primary Key along with all its related Users from 'TUsers'.
-- Gets: @Key uniqueidentifier
-- Returns: @ErrorCode int
-------------------------------------------------------------------------

ALTER PROCEDURE [dbo].[usp_TTableName_SelectOneByPrimaryKey]
	@Key uniqueidentifier
AS

BEGIN
	BEGIN TRY
```

```
		-- SELECT an existing row from the table.
		SELECT
			[Key],
			[Name],
			[ExecutionLink],
			[TStamp],
			[DateCreated],
			[InsertedBy],
			[LastUpdatedBy],
			[Source]
		FROM [dbo].[TApplication]
		WHERE
			[Key] = @Key
	END TRY
	BEGIN CATCH
		DECLARE @T INT
		SELECT @T = 9
		-- An error has occurred so format it and
		-- return it to the caller.
		--EXEC usp_ReturnFormattedError (define this for more control)
		RAISERROR('THIS IS SHOULD BE REPORTED BACK', 16, 34) WITH
SETERROR
	END CATCH;

END
```

Code Walkthrough Guidelines

Procedure

- A code walkthrough will be performed after the completion of each unit of work.
- The code walkthrough is a short process and should take no more than an hour or two for unit of work.
- The technical lead may review the code themselves or delegate this task to another team member.
- The code should be reviewed using the checklist
- Any related database items should be reviewed by the DBA/Technical Lead.
- Add name of reviewer, date reviewed, and current checklist status (passed, failed) to the top of the source code of the class/form/module.
- If the code failed for any reason, the reviewer should include enough information in the comments section of the checklist for the developer to make corrections. For example, line number, function name, etc…

- In the event of a dispute, the technical lead will make the final decision based on coding standard.
- The developer should make any correction noted on the checklist.
- If the code is modified significantly, the class/form/module should be reviewed again.

Source Code Documentation

- A compact html (.chm) file should be created for the code documentation. Every class along with its public methods, public properties and public events should have a brief documentation with examples. Use NDoc (http://ndoc.sourceforge.net) for generating code documentation.

Source Code Checklist

Unit of work	
Developer	
Reviewer	
Date of review	
Result	

Item	Complies	Comments
The source code complies with naming conventions		
The source code is formatted correctly		
Variables are declared using proper scope, type, and initialization		
Considering security, all attributes are declared with the proper scope identifier		
Resources are released properly		

The source code includes appropriate commenting		
Reusable classes and components are used where possible		
Error handling is used where appropriate		
User messages comply with standards		
GUI complies with the presentation standard of the project		
All relevant database items comply with standards.		

Change Log

This document is originally prepared by Rob Wells for Ignyte Software Inc.