SYNERGEX

Synergy Style Guide

Table of Contents

1. Naming Conventions

1.1 Use US-English for naming identifiers.

1.2 Use Pascal and camel casing for naming identifiers.

In Pascal casing the first letter of each word is capitalized. For example, **IsValid**.

In camel casing only the first letter of the second and subsequent words are capitalized. For example, **thisIsLocal**.

Two-letter abbreviations in Pascal casing have both letters capitalized. The same is true in camel casing, except at the start of an identifier where both letters are lowercase. Abbreviations with more than two letters are capitalized like ordinary words in both Pascal and camel casing. Some examples:

PASCAL CASING	CAMEL CASING
UIEntry	uiEntry
NewImage	newImage
SaveUISettings	saveUISettings
GetApiParams	getApiParams

1.3 Do not use casing to differentiate identifiers.

Some languages, like C#, support distinguishing identifiers by case. Synergy/DE does not. Do not attempt define a type called A and a in the same context.

1.4 Use abbreviations with case.

Do not contract words in an identifier but do use well-known abbreviations. For example, use **`GetWin`** instead of **`GetWindow`**. Use well-known abbreviations such as **`UI`** instead of **`UserInterface`**.

1.5 Do not use an underscore in identifiers.

1.6 Name an identifier according to its meaning and not its type.

Avoid using language-specific terminology in the names of identifiers.

As an example, do not use definitions like

```
public method Write      ,void
in req stringValue       ,string
public method Write      ,void
in req intValue          ,int
```

Instead use

```
public method Write      ,void
in req value             ,string
public method Write      ,void
in req value             ,int
```

1.7 Do not add a suffix to a class or struct name.

Do not add suffixes like **`Struct`** or **`Class`** to the name of a struct or class.

1.8 Use a noun or noun phrase to name a class or struct.

Also, if the class involved is a derived class, it's a good practice to use a compound name. For example, if you have a class named **`File`**, deriving from this class may result in a class named **`CustomerFile`**.

1.9 Do not add an Enum suffix to an enumeration.

1.10 Use singular names for enumeration types.

For example, do not name an enumeration type **`Protocols`**; name it **`Protocol`** instead.

1.11 Use a plural name for enumerations representing bit fields.

Use a plural name for such enumeration types. The following code snippet is a good example of an enumeration that allows combining multiple options:

```
public enum SearchOptions
CaseInsensitive ,^x(0001)
WholeWordOnly ,^x(0002)
AllDocuments ,^x(0004)
Backwards ,^x(0008)
AllowWildcards ,^x(0010)
endenum
```

1.12 Do not use numeric digits that can be mistaken for letters, and vice versa.

For example:

```
begin
data b001          ,boolean
data lO            ,int
data I1            ,int
end
```

1.13 Built-in aliases to reference types should be all lowercase.

All built-in types should be lowercased. For example, string, boolean, a, i, int, etc.

1.14 All built-in reference types should be mixed case, beginning with an uppercase letter.

Built-in reference types like **`System.Object`** and **`System.String`** should be mixed case, beginning with an uppercase letter.

1.15 All coding should be lowercase.

1.16 All .includes should be double quoted.

1.17 In a .define, the identifier being defined should be uppercase.

1.18 Naming items in Visual Studio

Button:	**btn**	**Radio Button:**	**rbtn**
Check Box:	**cbox**	**Text Box:**	**tbox**
Combo Box:	**combox**	**Panel:**	**pan**
Label:	**lab**	**Tab Control:**	**tabctrl**

Using this naming convention, items will be named according to where they are placed in the form (e.g., if form name is ISAMUtils, main tab is ISUTL, and sub tab is Reload, check box A would follow the format of cboxISAMUtilsISUTLReloadA). Also, any buttons or fields associated with an item should have the same naming standard (i.e., a button associated with that check box would be coded as btnISAMUtilsISUTLReloadA). By following this format, you will know where a specific item is in the form when coding.

2. Comments and Documentation

2.1 All source code should have a disclaimer header.

Example:

```
;; Title:         <FILENAME>
;; Type:          Function | Subroutine | Method | Program | Include file
;; Description:   <BRIEF DESCRIPTION OF ROUTINE ETC.>
;; Author:        <NAME>, Synergex Professional Services Group
;; Copyright© 2009 Synergex International Corporation. All rights reserved.
;; WARNING:       All content constituting or related to this code ("Code")
;;                is the property of Synergex International Corporation ("Synergex")
;;                and is protected by U.S. and international copyright laws. If you
;;                were given this Code by a Synergex employee then you may use and
;;                modify it freely for use within your applications. However, You may
;;                not under any circumstances distribute this Code, or any modified
;;                version or part of this Code, to any third party without first
;;                obtaining written permission to do so from Synergex. In using this
;;                Code you accept that it is provided as is, and without support or
;;                warranty of any kind. Neither Synergex nor the author accept any
;;                responsibility for any losses or damages of any nature which may arise
;;                from the use of this Code. This header information must remain
;;                unaltered in the Code at all times. Possession of this Code, or
;;                any modified version or part of this Code,indicates your acceptance of
;;                these terms.
```

2.2 Temporary comments should begin with a single semicolon (;)

These comments should also start in column 1.

2.3 Comments should begin with two semicolons (;;)

2.4 Documenting comments should begin with three semicolons (;;;)

2.5 All comments should be written in US English.

2.6 All comments should be aligned with the code.

A comment should line up with the beginning of the code it's commenting on. See 2.8 for an exception to this guideline.

2.7 No comments should appear at the end of a line of code.

With the additional indentation, the comments are easily lost at the edge of the editor window. Rule CD008 is an exception to this rule.

2.8 Comments on data definitions should appear on the same line.

A data definition comment should appear on the same line as the definition itself (regardless of whether the definition is in the data division or part of a DATA statement). The comment should begin with two semicolons.

2.9 Use XML tags for documenting types and members.

All public and protected types, methods, properties, etc., should be documented using XML tags. Using these tags will allow IntelliSense to provide useful details while using the types. Also, automatic documentation generation tools rely on these tags.

Section tags define the different sections within the type documentation.

SECTION TAGS	DESCRIPTION	LOCATION
<summary>	Short description	Type or member
<remarks>	Describes preconditions and other additional information	Type or member
<param>	Describes the parameters of a method	Method
<return>	Details the return value of a method	Method
<exception>	Lists the exceptions that a method or property can throw	Method or property
<value>	Describes the type of the data a property accepts and/or returns	Property
<example>	Contains example (code or text) related to a member or type	Type or member
<overload>	Provides a summary for multiple overloads of a method	First method in an overload list

Exceptions: Private and nested classes do not have to be documented in this manner.

2.10 Internal routines (labels) are prefixed with a separating comment of 78 "-" characters.

This helps to break up the code and identify internal routines.

2.11 Only the header and labeling comments may start in column 1.

3. Coding Structure

3.1 The namespace declaration should begin in column 1.

3.2 The class definition should be indented one level from the namespace.

3.3 A public class name should begin with a capital letter.

3.4 A private or protected class name should begin with a lowercase letter.

```
;;example namespace
namespace SynPSG.ChronoTrack.DataEntities
;;*****************************************
;;A public class
;;*****************************************
public abstract class DataEntity
;;*****************************************
;;A private class
;;*****************************************
private class ioTypes
endclass
endclass
endnamespace
```

3.5 Declare all class fields at the beginning of the class.

All class fields should be declared at the beginning of the class definition, in the following order:

- Public fields
- Protected fields
- Private fields

3.6 Member fields should be indented one level from the class declaration.

3.7 Public field names should start with an uppercase letter.

3.8 Private field names that represent public properties should begin with a lowercase m.

This will assist in distinguishing between private data and the public property.

3.9 Group together methods and properties with same access levels.

Include them in the following order:

- Public
- Protected
- Private

3.10 Method declarations should be indented one level from the class declaration.

3.11 The method accessibility should be included.

3.12 Public method names should begin with an uppercase letter.

3.13 Private and protected methods should begin with a lowercase letter.

3.14 The return type of the method should be tabbed one indent from the method name and preceded with a comma.

3.15 Arguments should be indented one level from the method declaration.

3.16 The argument name should begin with a lowercase character.
Arguments are private to the method and so should begin with a lowercase letter.

3.17 Every argument must list its modifiers (REQ, INOUT, etc).

3.18 The argument types should match the indent level of the method type.

3.19 The end of the argument list should be identified with an **`endparams`** statement.
The **`endparams`** should be indented one level from the method declaration.

3.20 Private method data definitions should be indented one level from the method declaration.

3.21 Property declarations should be indented one level from the class declaration.

4. Exception Handling

4.1 Only throw exceptions in exceptional situations.
Do not throw exceptions in situations that are normal or expected (e.g., end-of-file). Use return values or status enumerations instead. In general, try to design classes that do not throw exceptions in the normal flow of control. However, do throw exceptions that a user is not allowed to catch when a situation occurs that may indicate a design error in the way your class is used.

4.2 Do not throw exceptions from inside destructors.
This is an issue for .NET. When you call an exception from inside a destructor, the CLR will stop executing the destructor, and pass the exception to the base class destructor (if any). If there is no base class, then the destructor is discarded.

4.3 Only re-throw exceptions when you want to specialize the exception.
Only catch and re-throw exceptions if you want to add additional information and/or change the type of the exception into a more specific exception. In the latter case, set the **`InnerException`** property of the new exception to the caught exception.

4.4 List the explicit exceptions a method or property can throw.
Describe the recoverable exceptions using the **`<exception>`** tag.

Explicit exceptions are the ones that a method or property explicitly throws from its implementation and which users are allowed to catch.

4.5 Allow callers to prevent exceptions by providing a method or property that returns the object's state.

For example, consider a communication layer that will throw an **`InvalidOperationException`** when an attempt is made to call **`Send()`** when no connection is available. To allow preventing such a situation, provide a property such as **`Connected`** to allow the caller to determine if a connection is available before attempting an operation.

4.6 Throw informational exceptions.

When you instantiate a new exception, set its **`Message`** property to a descriptive message that will help the caller to diagnose the problem. For example, if an argument was incorrect, indicate which argument caused the problem.

4.7 Throw the most specific exception possible.

Do not throw a generic exception if a more specific one is available.

4.8 Only catch the exceptions explicitly mentioned in the documentation.

Do not catch the base class **`Exception`** or **`ApplicationException`**. Exceptions of those classes generally mean that a non-recoverable problem has occurred.