RESOURCE AND PATIENT MANAGEMENT SYSTEM

Ensemble Programming Standards and Conventions

Version 1.0
September 2010

Office of Information Technology (OIT)
Division of Information Resource Management
Albuquerque, New Mexico

Table of Contents

1.0 Purpose, Policy, and Standards and Conventions

1.1 Purpose

The purpose of this document is to provide a set of standards and conventions for Cache and Ensemble development. It assumes an understanding of and compliance with object-oriented programming practices.

There are many books and Web sites available that provide object-oriented design and programming information, including Oracle's guide at http://download.oracle.com/javase/tutorial/java/concepts/index.html for object-oriented programming concepts and the Corelinux Consortium's document at http://www.literateprogramming.com/oostnd.pdf for object-oriented design principles.

1.2 Policy

1.2.1 Conformance

All Cache and Ensemble software developed for the Indian Health Service (IHS) will conform to the Ensemble Standards and Conventions. In areas where standards are not specified, software development will conform to other applicable IHS programming standards and conventions and industry standards.

2.0 Cache Programming Standards and Conventions

All Cache-based IHS software will meet the following standards and comply with the spirit of the conventions.

2.1 Document Definitions

2.1.1 Camel Case

Style of writing in which multiple words are joined without spaces and the initial letter of each word is capitalized while the other letters are in lower case. In upper camel case, the first letter of the compound word is capitalized, e.g. UpperCamelCase. In lower camel case, the first letter of the compound word is lower case, e.g. lowerCamelCase.

2.1.2 K&R-Style Indentation

Indentation style in which the opening brace of a conditional or loop block is on the same line as the control statement and the closing brace is on its own line at the same level of indentation as the control statement. For example:

```
if (x=0) {

    // One or more indented commands

}
```

The opening brace following a declaration is on its own line at the same level of indentation as the declaration. For example:

```
ClassMethod GetAge(DOB As %Date) As %Integer

{

    // One or more indented commands

}
```

2.2 Class Structure and Format

2.2.1 First Line

The first line of a class definition contains a comment describing the class, its purpose, and its use. Any markup required for clarity should use standard HyperText Markup Language (HTML) tags.

2.2.2 Second Line

The second line of a class definition contains the class declaration, which has the following format:

Class (package).(classname) Extends (parent-classes) [(class-keywords)]

Where

(package).(classname) is the full name of the class

Extends (parent-classes) defines the class inheritance. It is optional for classes consisting of only utility methods.

(class-keywords) are any keywords defining the class, such as ProcedureBlock.

2.2.3 Third Line

The third line of a class definition consists of the opening brace ("{"). There are no other characters on this line.

2.2.4 Fourth Line

The fourth line of a class definition is a blank line.

2.2.5 Parameters Section

Beginning on the fifth line of a class definition is the optional parameters section. The parameters section consists of zero or more class parameter declarations, separated by a blank line. Comments are optional but should be included for custom parameters or in other instances where the parameter's use is not obvious. Each parameter declaration has the following format:

/// Optional comment

Parameter (parameter-name) As (type) = "(value)";

where:

(parameter-name) is the name of the parameter. Parameter names are in uppercase;

As (type) is the optional type declaration for the parameter;

(value) is the value assigned to the parameter.

There is a blank line between the parameters section and the subsequent section.

2.2.6 Properties Section

Following the parameters section in a class definition is the optional properties section. The properties section consists of zero or more property declarations, separated by a blank line. Comments are optional but should be included for custom parameters or in other instances where the parameter's use is not obvious. Each property declaration has the following format:

/// Optional comment

Property (property-name) As (type) [(property-parameters)];

where:

(property-name) is the name of the property in upper camel case (mixed case with a capital first letter);

As (type) is the optional type declaration;

[(property-parameters)] is an optional declaration of the property's parameters.

There is a blank line between the properties section and the subsequent section.

2.2.6.1 Relationship Properties

Relationship properties are included in the properties section along with the other properties. They may be grouped together for clarity.

2.2.7 Indices Section

Following the properties section in a class definition is the optional indices section. The indices section consists of zero or more index declarations, separated by a blank line. Comments are optional. Each index declaration has the following format:

/// Optional comment.

Index (name) On (property-list) [(index-parameters)]

where:

(name) is the name of the index in upper camel case, suffixed with "IDX" in uppercase.

2.2.8 Methods Section

Following the properties section in a class definition is the optional methods section. The methods section consists of zero or more method definitions, separated by a blank line. Each method definition has the following format:

/// Optional comment.

(method-type) (name) (parameters) As (return type) [(method-parameters)] {

(method-code)

}

where:

(method-type) is either *ClassMethod* or *Method*, depending on the type of method being defined;

(name) is the name of the method in upper camel case;

(parameters) is the list of any parameters for the method;

(return type) defines the type of value returned by the method;

(method-parameters) defines any method parameters for the method;

(method-code) is the code for method.

The opening brace ("{") for the method is on the same line as the method declaration, preceded by a space. The closing brace ("}") is on its own line.

2.2.9 Comments

A comment preceding a class, parameter, property, index, or method declaration consists of three slashes and a space, followed by the comment. Any markup should use standard HTML tags. For example:

/// Number of seconds to wait for the document to be generated before returning an error.
Parameter DOCUMENTTIMEOUT = 60;

2.2.10 ProcedureBlock

Classes must use the ProcedureBlock keyword except when its omission is required for backwards compatibility.

2.3 Name Requirements

2.3.1 Packages

The first level of the package name for classes must be the namespace assigned to the project. Subsequent levels of the package name shall be used to group classes into functional groups within the project. The first level (namespace) is in uppercase, and the subsequent levels are in upper camel case. Package names must be meaningful and concise.

2.3.2 Classes

Class names must be meaningful and concise. Classes that represent an item should be in the singular, e.g. BCDE.Admit.Patient, not BCDE.Admit.Patients.

2.3.3 Declarations

Parameter, Property, Index, Method, ClassMethod, and other declaration keywords are in upper camel case.

2.3.4 Parameters

Parameter names are in all uppercase.

2.3.5 Properties

Property names are in upper camel case. The name should represent a thing or a flag.

2.3.6 Indices

Index names are in upper camel case and have the suffix “Idx”. The name should summarize the fields being indexed.

2.3.7 Methods

Method names are in upper camel case and may be no longer than 31 characters. The name should describe the action the method performs, e.g. GetAge(). The name of a method that returns a Boolean should express an assertion, e.g. IsMinor().

2.3.8 Variables

Variable names are in lower camel case and may be no longer than 31 characters. Nondescriptive variable names, such as x, i, and obj, may be used only over a small section of code. Variables used over a longer section of code must be given meaningful names.

2.4 Classes

2.4.1 Use

Classes, properties, and methods should be used instead of routines, globals, and functions whenever possible. Routines should be used only when necessary for compatibility. Globals should not be used directly, except when using process-private globals for sorting.

2.4.2 Distribution

Classes should be distributed using the class distribution software in conjunction with proper package mapping. See the class distribution documentation for more information.

2.4.3 Deleting Objects

The methods for deleting objects, such as %Delete() and %DeleteExtent(), should normally be called with "0" for the *concurrency* parameter. Calling these methods with the default concurrency values causes a large number of locks to be taken out, potentially filling up the lock table.

2.4.4 Persistence and Instantiation

A class must not extend %Persistent or any other persistent class unless it is necessary for its objects to be persisted. A class that is not intended to be instantiated, such as a class of utility methods, must not extend %RegisteredObject.

2.4.5 Source Code

The "k" ("keep source") flag should be used when importing or compiling classes to retain the source code.

2.4.6 Storage Strategy

The default storage strategy for a class should not be overridden without good reason. The "SQL Storage" strategy should not be used unless mapping a persistent class to a pre-existing global structure.

2.4.7 Transaction Processing

By default, the *%Save()* method of a persistent class uses transaction processing, which will enable journaling even if journaling is disabled. This behavior may be overridden in Cache/Ensemble version 2010.1 or later by doing *$System.OBJ.SetTransactionMode(0)*. In versions prior to 2010.1, the command is *do $$SetTransactionMode^%apiOBJ(0)*. The use of *^%apiOBJ* is deprecated in version 2010.1 and later and must be replaced with the *$System.OBJ* call, as *^%apiOBJ* may be changed or removed without warning.

2.4.8 Web Service

A web service class should specify a *LOCATION* parameter for use when generating a WSDL via the *FileWSDL()* method. When not generating a WSDL, the parameter must be commented out.

A web service class must set the *USECLASSNAMESPACES* parameter to 1 unless it is required to be set to 0 for backwards compatibility. The *SOAPSESSION* parameter must be set to 0 unless the use of session headers is required.

2.5 Cache ObjectScript

2.5.1 Abbreviations

Abbreviated commands may not be used. The full name of the command must be used. Intrinsic functions and system variables may be abbreviated.

2.5.2 Commands per Line

There should be only one command per line. Commands with multiple related arguments are permitted, e.g. set x=0, y=1.

2.5.3 Comments

Comments should be indicated by "//" unless a special comment indicator is needed, such as "/// " for class documentation, "#;" for comments to be removed from generated .INT routines, or ";;" for comments to be retained in object code for use by the $text function.

Code that is commented out but is to be retained must have a comment documenting the purpose of the code and how it is to be used. For example, the LOCATION parameter in a Web service must not be present except when generating a Web Service Definition Language (WSDL) via the FileWSDL() method. The parameter must have a comment indicating that the parameter is to be uncommented when generating a WSDL.

2.5.4 Dot Structure

The argumentless *do* command followed by a "dot" structure is deprecated. Braces ("{ }") should be used instead.

2.5.5 For

Code executed within a *for* loop must be enclosed in braces ("{ }"). *While* loops should be used in place of argumentless *for* loops. When the braces enclose a single command, the entire *for* command, including braces may be on one line. Otherwise, the opening brace is on the same line as the *for* command, and the closing brace is on its own line at the same level of indentation as the associated *for* command.

2.5.6 If/Else

Code executed conditionally after i*f* or *else* must be enclosed in braces ("{ }"). When the braces enclose a single command, the entire *if* or *else* command, including braces may be on one line. Otherwise, the opening brace is on the same line as the *if* or *else* command, and the closing brace is on its own line at the same level of indentation as the associated *if* or *else* command. *else* must appear on a separate line from any preceding closing brace.

2.5.7 Indirection

Indirection must not be used to get or set an object property. Because variables in indirection are scoped outside of the method, care must be exercised when using indirection to ensure the desired functionality is achieved.

2.5.8 Kill

Killing of local variables is generally not required and should not be used unless necessary, such as when killing an OREF before reloading the object in order to get an updated instance.

2.5.9 Logical Operators

The "&&" and "||" operators must be used for "and" and "or" unless there is a specific need to use the "&" and "|" operators.

2.5.10 New

The *new* command must not be used.

2.5.11 Streams

The classes %FileBinaryStream, %FileCharacterStream, %GlobalBinaryStream, and %GlobalCharacterStream are deprecated. %Stream.FileBinary, %Stream.FileCharacter, %Stream.GlobalBinary, and %Stream.GlobalCharacter should be used instead.

2.5.12 % Variables

% variables may be used only for system-wide variables that must persist beyond a given procedure block.

2.5.13 White Space

Blank lines should be used to clarify code by grouping related commands together. Two or more consecutive blank lines should not be used. Indentation must be consistent to assist the reader in identifying the code structure.

2.5.14 Xecute

Because variables in an *xecute* command are scoped outside of the method, care must be exercised when using the *xecute* command to ensure the desired functionality is achieved.

2.5.15 $Zutil

$Zutil must not be used when there are higher-level methods to perform the same functionality, such as *##class(%File).DirectoryExists()* instead of *$zutil(140,4)*. All other uses of *$zutil* require SACC approval.

3.0 Ensemble Programming Standards and Conventions

3.1 Name Requirements

3.1.1 Classes

Name conventions for Ensemble business hosts should follow the same conventions as other Cache classes, except for appending a suffix indicating the host type. Business services must end in “BS”, business processes in “BP”, business operations in “BO”, and data transformations in “DTL”, unless the business host type is indicated in the package name. For example, BCDE.Prod.FileDocumentBO requires a suffix, while BCDE.Prod.Operations.FileDocument does not.

3.2 Business Processes

3.2.1 BPL

Business processes should use BPL instead Cache ObjectScript-based classes unless the requirements of the business process make it prohibitively difficult to do so.

3.2.2 <code>

Each <code> activity must perform a single logical action. If a <code> activity’s action cannot be easily summarized, it should be divided into multiple <code> activities, each representing a single logical action.

3.2.3 <sequence>

<sequence> blocks should be used to group multiple activities that constitute a logical action in order to make the diagram easier to follow and maintain.

3.3 Data Transformations

3.3.1 DTL

Data transformations should use DTL instead of Cache ObjectScript-based classes unless the requirements of the data transformation make it prohibitively difficult to do so.

3.3.2 <code>

<code> elements should be used for small sections of Cache ObjectScript. A large block of Cache ObjectScript logic should be placed in a method and called from a <code> element.