Singularity Design Note 7

Software Development Standards

Standards for how the Singularity system and applications are constructed and maintained by their developers

This design note describes the tools, processes, and standards by which Singularity developers develop, modify, store, and publish information about the Singularity system and its applications.

1. Introduction

The purpose of this document is to specify tools, procedures, and conventions to try to ensure that:

- What someone else can build and run, you can build and run, both now and in the future, with the same results.
- When people check in code, after the check-in the functionality that worked before still works.
- When people check in code, the code changes are announced and described to other project members.
- Coding standards allow people to edit one another's code in a seamless fashion.
- Developers can use state-of-the-art development tools.

These standards are intended to cover both the development process for builders of the Singularity system and the development and usage environments for users of Singularity.

The standards cover five areas: ***Build Environment***, ***Checkins***, ***Pre- and Post-Checkin Testing***, ***Coding Standards***, and ***Bug Tracking***.

2. Build Environment

When building Singularity the following software versions are recommended for your build machine:

CLR version 1.1 RTM

Note that while it is not needed for building Singularity, if you have installed Visual Studio .NET Professional 2003, you will already have version 1.1 of the CLR installed.

While it is possible that other configurations may work, the above set constitutes the only tested build environment.

3. Revision Control and Development

Source Depot is used as the revision control application for Singularity. When developing code, developers are encouraged to have at least two enlistments in the source depot. The first enlistment should be for developing code in and the second for testing changes made in the first before they are submitted to the depot. Tools exist for packaging up changes in one tree and applying them to another. These allow developers to perform buddy builds on their own enlistments and also package up changes for peer review. The preferred tool for packaging changes for Singularity is `jjpack`. Section 7 contains more information on `jjpack`.

3.1. Multiple Enlistments

Creating multiple enlistments is straightforward. When enlisting, the developer need only supply a client name to uniquely identify enlistment. This can be done by setting the variable in the sd.ini file used when originally enlisting, ie,

```
SDPORT=BGIT-SDSingular:4060
SDCLIENT=UniqueClient123
```

The user then runs `sd client` in the usual way to obtain an enlistment.

4. Pre- and Post- Checkin Testing

Before checking in, each developer is required to run the build verification tests to make sure that their checkin won't break anything. Currently this consists of just `bvt.cmd`. You can also invoke the BVT from the Singularity shell by typing "bvt".

Before checking in, each developer should perform a buddy build on their code in a different enlistment. Ideally the buddy build will be a clean build to detect missing dependencies in makefiles. While not strictly required, booting the buddy build and running the BVT is generally regarded as a good practice.

After checking in, a developer may request a buddy build from another project member. Include the identity of the buddy builder in your check-in message.

5. Check-ins

The `sd submit` command checks code into the repository (see `sd help submit`). The check-in process presents the user with a list of changed files, the changelist, and prompts the user for a changelist description. The changelist description should begin with a 1-line overview. This makes the output of source depots change review tools more informational. The rest of the description should describe the changes and provide any motivational information that might be useful in the future. The motivation information is particularity useful where the reason for the change may be not be apparent from the description. For instance a change may be from stack to heap allocated buffers and the motivation is to prevent buffer over-runs.

A check-in message may be edited after submission using `sd change -f <changelist#>`, see `sd help change` for more information.

After the code is checked into the repository email should be sent to the checkin alias. The email may contain one, or more, related changelist descriptions batched together in the same email. The checkin email body should include the ***subsystem name, change type***, ***Product Studio bug numbers***, ***buddy builder***, a ***text description of the change***, the ***source depot change identifier***, and the ***files touched***.

The subject line of the checkin email should name the subsystem, the word "checkin", and a brief summary of the change. For the change type, please select from the set:

```
New Feature
Bug Fix
Code Cleanup/Refactoring
Comments/Style (no code change)
Performance Improvement
```

Such an email might look like:

```
Subject: Kernel checkin: Fixed BVT.cmd script

Subsystems:  Kernel
Change Type:  Bug fix
Bug Number:  123
Buddy Build:  mbj3

Change 462 by REDMOND\galenh@GALENH0 on 2004/04/01 11:53:15

Updated BVT script to use kd.exe now that we don't log to serial port.

Mark correctly pointed out that I had forgotten to fix the BVT script.
It now uses kd.exe to collect the log information.  This resolves bug
number 123.

Affected files ...

... //depot/main/base/build/bvt.cmd#2 edit
```

Note that much of this information can be obtained from the `sd describe -s` command, or can be copied from the `sd submit` command output. Please refrain from writing additional information describing the change in the email that is not in the changelist description. If there is additional information, update the changelist and recapture the output for inclusion in the email.

6. Coding Standards

6.1. Singularity Coding Standards Synopsis

This section presents a summary of the coding standards that Singularity code is expected to adhere to. This section summarizes and borrows from several other coding standards documents produced within the company.

6.1.1. Lines Widths, Spacing, and Tabs

To make it possible for people to easily view and edit code written by others, all Singularity text files adhere to the following standards:

1. All indentation will be accomplished by the insertion of ***spaces*** -- not tab characters.
2. All blocks will be indented by ***four characters***.
3. All text lines should be no more than 80 characters wide.

In VS.Net you can configure four character spaces, instead of tabs, with the menu settings below:

```
Tools / Options / Text Editor / All Languages / Tabs / Insert spaces.
```



```
Tools / Options / Text Editor / All Languages / Tabs / Tab Size 4.
```

In Emacs, you can configure four character spaces, instead of tabs, your .emacs file with:

```
(set-default 'indent-tabs-mode nil)
(set-default 'tab-width 4)
```

6.1.2. Naming and Usage Consistency Guidelines

The following symbols represent FxCop's coverage of each guideline:

- ● Fully covered by FxCop rules.
- ◉ Partially covered by FxCop rules.
- ○ Not covered by FxCop rules.

6.1.2.1. Casing & Naming Guidelines

- ● 1. Do use PascalCasing for all public identifiers (except for parameter names) consisting of compound words. For example, use `TextColor` rather than `Textcolor` or `Text_color`.
- ● 2. Do use camelCasing for all parameter names, local variables, and private field names.
- ◉ 3. Do not use Hungarian notation.
- ◉ 4. Do not use underscores, hyphens, or any other non-alphanumeric characters.
- ● 5. Do not use of shortenings or contractions as parts of identifier names. For example, use `GetWindow` rather than `GetWin`.
- ◉ 6. Do not use acronyms that are not generally accepted in the field.
- ◉ 7. Do use only when absolutely necessary and only for well-known acronyms. For example, use UI for User Interface and Olap for On-line Analytical Processing.
- ● 8. When using acronyms, do use PascalCasing or camelCasing for any acronyms two or more characters long. For example, use HtmlButton rather than HTMLButton, System.Io instead of System.IO, and FsContract instead of FSContract.
- ○ 9. Do name classes and properties with nouns or noun phrases.
- ○ 10. Do name methods using a verb.
- ◉ 11. Consider naming events with a verb. Examples: Clicked, Painting, DroppedDown, etc.
- ● 12. Do use the following prefixes:
  - ● a. "I" for interfaces.
- ● 13. Do use the following postfixes:
  - ● a. "Exception" for types inheriting from System.Exception
  - ● b. "Collection" for types implementing System.Collections.ICollection.
  - ● c. "EventArgs" for types inheriting from System.EventArgs.
  - ● d. "EventHandler" for types inheriting from System.Delegate.
  - ● e. "Attribute" for types inheriting from System.Attribute.
- ○ 14. Do use the following template for naming namespaces: <Company>.<Technology>[.<Feature>][.Design]. For example,

Microsoft.Office.ClipGallery. Operating System components should use System namespace instead for the Microsoft namespace.

- ○ 15. Do not use organizational hierarchies as the basis for namespace hierarchies.

6.1.2.2. Usage Guidelines

- ● 16. Do ensure your library is CLS compliant. Apply CLSCompliantAttribute attribute to your assembly.
- ● 17. Do not expose public fields. Use properties instead.
- ◉ 18. Do prefer properties for concepts with logical backing store but use methods in the following cases:
  - ○ a. The operation is a conversion (such as **Object.ToString()**)
  - ○ b. The operation is expensive (orders of magnitude slower than a field set would be).
  - ○ c. Obtaining a property value using the **Get** accessor has an observable side effect.
  - ○ d. Calling the member twice in succession results in different results.
  - ◉ e. The member returns an array. Note: Members returning arrays should return copies of an internal master array, not a reference to the internal array.
- ○ 19. Do allow properties to be set in any order. Properties should be stateless with respect to other properties.
- ○ 20. Do not ship interfaces without also providing some default implementations (concrete base classes implementing the interface).
- ○ 21. Avoid public nested types.
- ○ 22. Do implement IDisposable for all types allocating native resources.
- ● 23. Do strongly prefer collections over arrays in public API.
- ● 24. Do provide strongly typed collections.
- ○ 25. Do use exceptions to report failures. Do not use return values as error codes.
- ○ 26. Do not catch or throw the Exception base type.
- ○ 27. Do throw common general purpose exceptions: ArgumentNullException, ArgumentOutOfRangeException, InvalidOperationException.
- ● 28. Do provide delegates with signatures following the pattern below far all events: <EventName>EventHandler(object sender, <EventArgs> e)

6.1.3. Code Formatting Guidelines

6.1.3.1. Tabs & Indenting

Tab characters (\0x09) should not be used in code. All indentation should be done with 4 space characters.

6.1.3.2. Bracing

Open braces for statements should always follow the closing parenthesis of the statement (with one space separating them). Braces should never be considered optional. Only a `while` should ever

follow a closing brace. Even for single statement blocks, you should always use braces. This increases code readability and maintainability; especially for if and else statements.

Right:

```
if (test) {
    statement;
}
else {
    statement;
}
```

Wrong:

```
if (test)
    statement;
else
    statement;
```

Wrong:

```
if (test)
{
    statement;
}
else
{
    statement;
}
```

Open braces for namespace, classes, structs, and methods should should always be at the beginning of the line after the statement that begins the block.

Right:

```
class Foo
{
    void Bar()
    {
    }
}
```

Wrong:

```
class Foo {
    void Bar() {
    }
}
```

6.1.3.3. Commenting

Comments should be used to describe intention, algorithmic overview, and/or logical flow. It would be ideal, if from reading the comments alone, someone other than the author could understand a function's intended behavior and general operation.

6.1.3.4. Comment Style

The // (two slashes) style of comment tags should be used in most situations. Where ever possible, place comments above the code instead of beside it. Here are some examples:


```
// This is required for WebClient to work through the proxy
GlobalProxySelection.Select = new WebProxy("http://itgproxy");

// Create object to access Internet resources
//
WebClient myClient = new WebClient();
```

Comments can be placed at the end of a line when space allows:

```
public class SomethingUseful
{
    private int          itemHash;            // instance member
    private static bool  hasDoneSomething;    // static member
}
```

6.1.3.5. Spacing

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

- Do use a single space after a comma between function arguments.
  Right: `Console.In.Read(myChar, 0, 1);`
  Wrong: `Console.In.Read(myChar,0,1);`
- Do not use a space after the parenthesis and function arguments
  Right: `CreateFoo(myChar, 0, 1)`
  Wrong: `CreateFoo( myChar, 0, 1 )`
- Do not use spaces between a function name and parenthesis.
  Right: `CreateFoo()`
  Wrong: `CreateFoo ()`
- Do not use spaces inside brackets.
  Right: `x = dataArray[index];`
  Wrong: `x = dataArray[ index ];`
- Do use a single space before flow control statements
  Right: `while (x == y) {`
  Wrong: `while(x==y)`
- Do use a single space before and after comparison operators
  Right: `if (x == y) {`
  Wrong: `if (x==y)`
- Do use a single space between parenthesis and opening braces in flow control statements
  Right: `while (x == y) {`
  Wrong: `while (x == y){`

6.1.3.6. File Organization

Source files should contain only one public type, although multiple internal classes are allowed

Source files should be given the name of the public class in the file

Directory names should follow the namespace for the class

For example, I would expect to find the public class “System.Windows.Forms.Control” in “System\Windows\Forms\Control.cs”…

- Classes member should be grouped into sections (Fields, Constructors, Properties, Events, Methods, Private interface implementations, Nested types)
- Using statements should be outside the namespace declaration.

```
using System;
```



```
namespace MyNamespace
{
   public class MyClass : IFoo
   {
      // fields
      int foo;

      // constructors
      public MyClass() { … }

      // properties
      public int Foo { get { … } set { … } }

      // events
      public event EventHandler FooChanged { add { … } remove { … } }

      // methods
      void DoSomething() { … }
      void FindSomethind() { … }

      //private interface implementations
      void IFoo.DoSomething() { DoSomething(); }

      // nested types
      class NestedType { … }

   }
}
```

6.2. FxCop Tool

The Build & Checkin Process group is considering a recommendation that the FxCop tool be run over your code to verify the style and formatting guidelines. Based on preliminary runs against the Singularity code base, it seems that some rule customizations for the Singularity project will likely be needed before this recommendation is put into place.

7. JJPack

Several tools exist for packaging up changes, though we recommend using `jjpack` for Singularity since it is written in C# and as such only depends on the CLR libraries already in the development tree. Other tools also exist internally, such as b`bpack`, but these have dependencies, like Perl, that we'd like to avoid in the project.

7.1. jjpack Setup

`jjpack` is present in the Singularity source tree in the `build\` directory. The `setenv.cmd` script adds the directory to the path when it is run.

Before running jjpack for the first time sdapi.dll needs to be registered with the system.

```
cd %SINGULARITY_ROOT%
regsvr32 /s build\sdapi.dll
```

7.2. Using jjpack

`jjpack` creates packages containing changes present between an enlistment and the source depot. It also provides a means of inspecting those changes and applying them to another enlistment. Running `jjpack -?` displays the basic usage information and additional usage detail is available for each jjpack command subcommand.

A typical usage for a developer with two Singularity enlistments in tree1 and tree2, might read:

```
> cd tree1
> sd sync

… developer makes changes within tree1 …

> jjpack ..\tree2\changes.jpk

> cd ..\tree2
> nmake clean
> sd sync
> jjpack unpack changes.jpk
jjpack: v7       http://jaybaz/jjpack
jjpack: info: Change 1403 created.
jjpack: //depot/main/base/build/boottest.cmd#4 - opened for edit

>nmake
… run bvt.cmd and any additional tests …
```

3.5 jjpack and changelists

When applying a `jjpack` package to an enlistment, jjpack creates a source depot changelist in the enlistment. The changelist can be used to back out the changes applied to the enlistment, leaving the enlistment in its prior state.

In the above example the changelist created has number 1403. The changelist description can be seen using:

```
> sd describe -c 1403
Change 1403 by REDMOND\ohodson@OHodson2Head on 2005/01/13 10:51:37
*pending*
        DO NOT SUBMIT - Created by JJPack - please enter a description
with 'sd change'
Affected files ...
... //depot/main/base/build/boottest.cmd#4 edit
```

The changes in the changelist can be backed out using:

```
> sd revert -c @1403 ...
//depot/main/base/build/boottest.cmd#4 - was edit, reverted
```

The changelist should then be deleted using:

```
> sd change -d @1403
Change 1403 deleted.
```

