

Quick Reference

"Preparing Code for Automated Design Spec. Tools"

Document Version 2.0 June 2010	Medidata Solutions, Inc. Corporate Office 79 Fifth Avenue New York, N.Y. 10003 +1 212 918 1800
<p>Medidata Solutions, Inc. Proprietary – Internal Use Only.</p> <p>Information in this document is subject to change without notice. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including, but not limited to, photocopying and recording, for any purpose without the express permission of Medidata Solutions, Inc.</p> <p>Medidata Solutions Worldwide, Medidata CRO Contractor, Medidata Designer, Medidata Grants Manager, Medidata Rave, Medidata Services, Medidata University and their respective logos are trademarks or registered trademarks of Medidata Solutions, Inc. All other brands or product names used in this document are trademarks or registered trademarks of their respective owners.</p>	
© Copyright 2010 Medidata Solutions, Inc. All rights reserved.	

1 Introduction

This Quick Reference provides information for developers on what needs to be included - and in what format - in source code to provide the correct information for consumption by automated Design Specification tools. Information on the tools is provided, together with directions on what is expected in the commented code and good commenting practices.

There are currently a number of tools in use by Technical Communications:

1. **Charlotte** is a Windows Forms application that extracts comments from C# code for insertion in Design Specifications by Design Helper. Charlotte traverses a project, collecting a list of the classes in the project and extracting class level comments for use in a Design Specification. **Design Helper** is a Microsoft Word add-in that consumes the XML files produced by Charlotte and produces the object tables found in Design Specifications for web pages, C# classes, and stored procedures.
2. **Reaper** is a Windows Forms application that is used to check the comment status of a Ruby application. It will check all of the *.rb* files in a folder and all of its sub-folders for comments that need to be associated with a class and with the methods defined for that class.
3. **nDoc** generates class library documentation from .NET assemblies and the XML documentation files generated by the C# compiler.
4. **rDoc** generates structured HTML documentation from Ruby source.
5. **Doxygen** is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), FORTRAN, VHDL, PHP, C#, and to some extent D.

2 Charlotte

Charlotte is a small C# application that will extract information from C# and SQL Server stored procedure files to be used in Medidata Design Specifications.

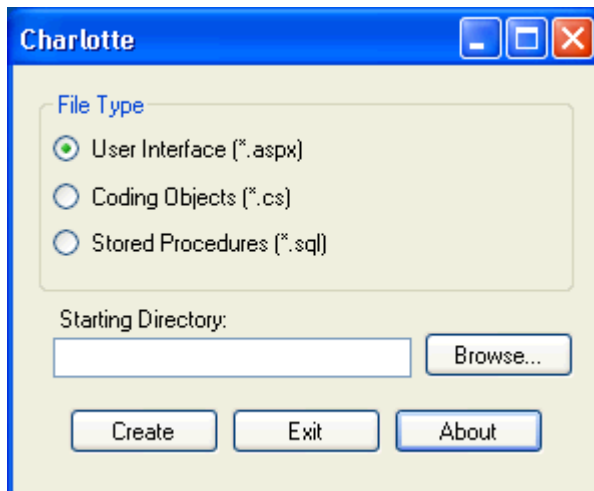


Figure 1: Charlotte dialog box

The information extracted by Charlotte is placed in XML files that are then imported by Design Helper, a Microsoft Word Add-in, and inserted into the appropriate place in the Design Specification. The buttons to insert the data are located on the Tech. Comm. tab of the Word 2007 ribbon.

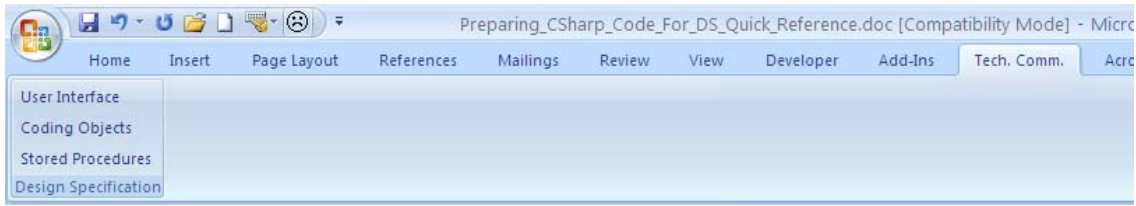


Figure 2: The Tech. Comm. Ribbon tab on the Word 2007 Ribbon.

1.1 Charlotte and Design Helper Requirements

Charlotte is a Windows Forms application that will work on Windows XP, Windows Vista, or Windows 7.

- Charlotte requires the .Net Framework 3.5
- Design Helper requires the .Net Framework 3.5 and Microsoft Word 2007.

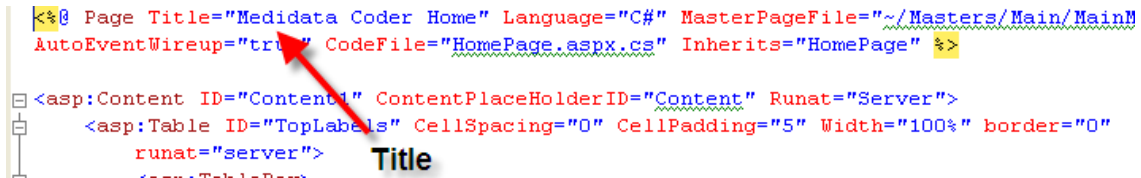
There are no special hardware requirements for any of these programs beyond the ability to run Windows XP or later versions of Windows.

1.2 Preparing C# Files for Charlotte

1.2.1 ASPX Pages

The Charlotte application expects to find the following information in an aspx web page:

1. Either the Title= attribute in the @ Page directive at the top of the file or in a <title> tag in the <head> section of the page.

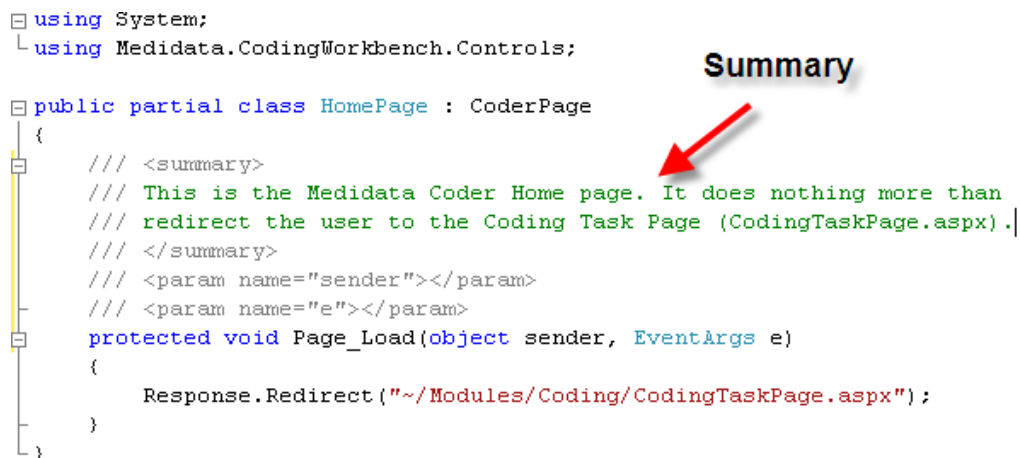


```
<%@ Page Title="Medidata Coder Home" Language="C#" MasterPageFile="~/Masters/Main/MainM
AutoEventWireup="true" CodeFile="HomePage.aspx.cs" Inherits="HomePage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="Content" Runat="Server">
  <asp:Table ID="TopLabels" CellSpacing="0" CellPadding="5" Width="100%" border="0"
    runat="server">
```

Figure 3: ASPX Page with Title in @ Page Directive.

2. An XML Documentation comment in the associated .aspx.cs file that describes the page. The program specifically looks for and extracts the first <Summary> tag from the .aspx.cs file.



```
using System;
using Medidata.CodingWorkbench.Controls;

public partial class HomePage : CoderPage
{
    /// <summary>
    /// This is the Medidata Coder Home page. It does nothing more than
    /// redirect the user to the Coding Task Page (CodingTaskPage.aspx).
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Redirect("~/Modules/Coding/CodingTaskPage.aspx");
    }
}
```

Figure 4: An ASPX code-behind file with a Summary Documentation Comment.

The information highlighted in Figure 5 and Figure 6 is used to create the User Interface table in the Design Specification as shown in Figure 7 below. The items in the list are sorted by path and file name.

3.1 User Interface

Screen	Change	Description	Code Entry Point	Unit Test
Error	New	Page that displays the current error (passed as query parameter).	C:\MedidataCode\CoderWeb\Error.aspx	10/12/2009
Medidata Coder Home	New	This is the Medidata Coder Home page. It does nothing more than redirect the user to the Coding Task Page (CodingTaskPage.aspx).	C:\MedidataCode\CoderWeb\HomePage.aspx	10/21/2009

Title (points to 'Medidata Coder Home') **Summary** (points to 'This is the Medidata Coder Home page...')

Figure 5: The UI Section of a Design specification.

1.2.2 C# Class Files

Charlotte expects to find the following information in C# class files:

1. Charlotte extracts the namespace from the namespace statement.
2. Charlotte extracts the description of the class from the first <Summary> documentation tag in the .cs file.

```

namespace Medidata.Framework
{
    /// <summary>
    /// Indicates the name of the field in a <see cref="T:System.Data.DataRow"/> to
    /// bind to the property this attribute is placed on (within a class that
    /// extends <see cref="T:DataboundObject"/>).
    /// This class can not be inherited.
    /// </summary>
    [AttributeUsage(AttributeTargets.Property, Inherited = true, AllowMultiple = true)]
    public class ColumnBindingAttribute : Attribute
    {
        properties
        Public Constructors
    }
}

```

Namespace (points to 'namespace Medidata.Framework') **Summary** (points to 'Indicates the name of the field...')

Figure 6: The information extracted from a C# file by Charlotte.

The information in Figure 8 above is used to create the Coding Objects table in the Design Specification as shown in Figure 9 below. Entries in this table will be sorted by Namespace and then filename excluding the path.

File Name	Change	Description	Unit Test
2.5.2 Medidata.Framework ← Namespace			
2.5.2.1 <u>AuditedObject.cs</u>	New	Base class for data bound objects which are audited. This class contains methods and properties that are common to all data bound objects that need to be audited.	10/21/2009
2.5.2.2 <u>ColumnBindingAttribute.cs</u>	New	Indicates the name of the field in a System.Data.DataRow to bind to the property this attribute is placed on (within a class that extends DataboundObject). This class can not be inherited.	10/21/2009

↑ **File Name**

↑ **Summary**

Figure 7: Information from C# classes inserted in a Design Specification.

1.2.3 Preparing Stored Procedures for Charlotte

For stored procedures and functions, Charlotte looks for a Summary tag similar to that used in C# files. For example:

```

/* -----
// Copyright(c) 2008, Medidata Solutions, Inc., All Rights Reserved.
//
// This is PROPRIETARY SOURCE CODE of Medidata Solutions Worldwide. The contents of
// this file may not be disclosed to third parties, copied or duplicated in
// any form, in whole or in part, without the prior written permission of
// Medidata Solutions Worldwide.
//
// Author: John Doe jdoe@mdsol.com
//
// <Summary>
// spApplicationRFetch retrieves all of the rows from the ApplicationR
// table for the specified ApplicationID.
// </Summary>
// -----

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND name = 'spApplicationRFetch')
    DROP PROCEDURE spApplicationRFetch
GO
create procedure dbo.spApplicationRFetch(@ApplicationRId int)
as
select * from ApplicationR where ApplicationID=@ApplicationRId
GO

```




Figure 8: Stored Procedure with <Summary> added.

This gives Charlotte markers to look for when extracting the Summary information. Once extracted, Charlotte will create the Stored Procedures and Functions table like the example in Figure 11 below. The files in the stored procedure table will be ordered by stored procedure name.

3.2.3 Stored Procedures & Functions

A number of new Stored Procedures will be added/modified to work with the new feature's tables, as follows:

Procedure Name	Change	Purpose	Unit Test
spApplicationRFetch.sql	New	spApplicationRFetch retrieves all of the rows from the ApplicationR table for the specified ApplicationID.	10/12/2009

Summary




Figure 9: Stored Procedure & Function Table in a Design Specification.

3 Reaper

Reaper is a C# application that counts the classes and methods in a Ruby project. The program provides a file-by-file count so that developers can easily tell which files are not completely commented. Reaper stored the results in an HTML file and then displays that file in the Report Preview section of the application.

1.3 Reaper Requirements

Reaper is a Windows Forms application that will work on Windows XP, Windows Vista, or Windows 7.

- Reaper requires the .Net Framework 4.0

There are no special hardware requirements for any of these programs beyond the ability to run Windows XP or later versions of Windows.

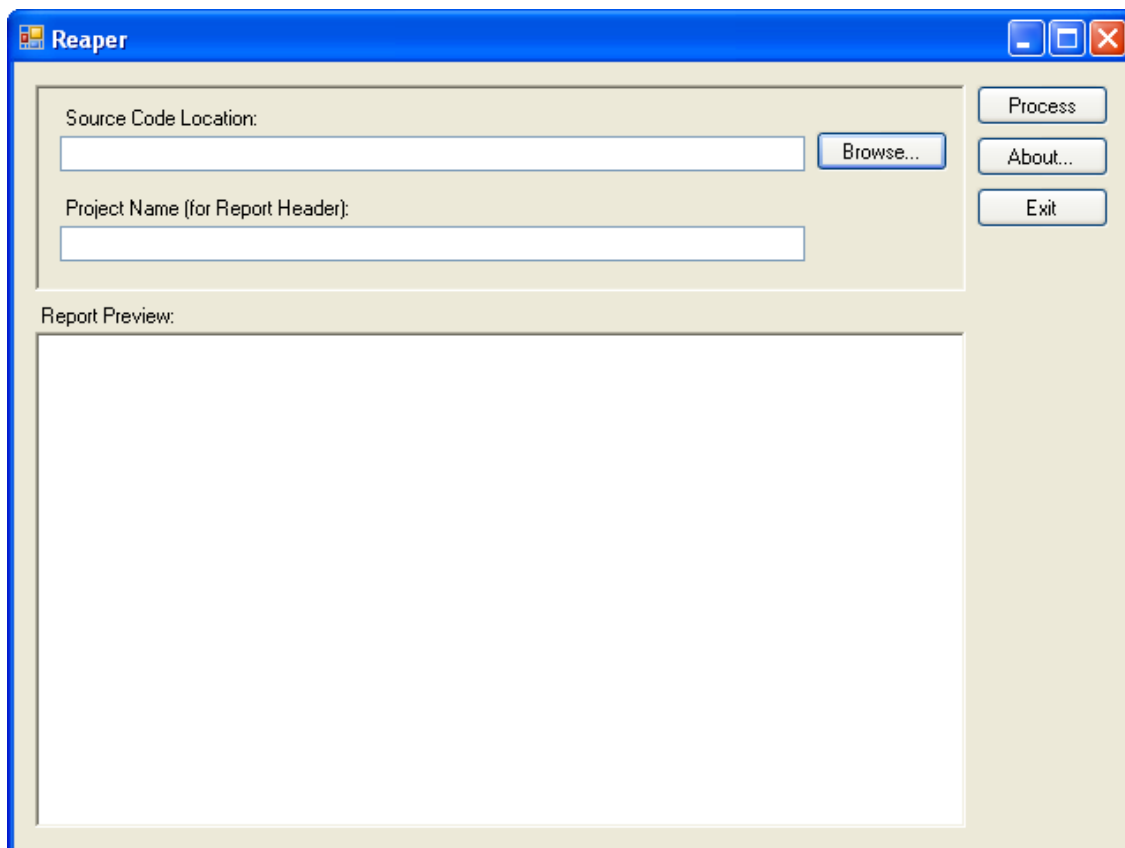


Figure 10: The Reaper application.

Reaper provides a report, in HTML format, that lists the name of each file in the project and a count of the number of classes and methods in each file. Reaper also calculates the percentage of the classes and methods in the project that already have comments.

Figure 2 shows a sample of a Reaper comments report.

Grants Manager 3.0

File Name (relative to root folder)	# of Classes	Commented	# of Methods	Commented
\controllers\application_controller.rb	1	1	25	22
\controllers\arms_controller.rb	1	0	10	9
\controllers\arm_costs_controller.rb	1	0	3	1
\controllers\bean_user_view_preferences_controller.rb	1	0	2	0
\controllers\budget_summaries_controller.rb	1	0	13	13
\controllers\budget_summary_view_preferences_controller.rb	1	0	2	0
\controllers\country_sets_controller.rb	1	0	4	3
\controllers\empty_controller.rb	1	1	0	0

Figure 11: Reaper's Comments Report

1.4 Preparing Ruby Code for Reaper

Reaper counts classes and methods in Ruby .rb files and calculates the percentage of the classes and methods in a project that have been properly commented so that when documentation comments are extracted using rDocs or some other tool, the correct information will be in place.

In order for Reaper to be able to get an accurate count, it is necessary to follow certain conventions when adding comments to Ruby code. Reaper expects the following conventions to be observed when placing comments in source code:


1. A comment that applies to a class or method must be above the line that begins the class or method definition.
2. There should be no blank lines between the comment and the class or method to which it refers.
3. Each line of the comment must start with the comment character (#) followed by a space. Like this:
This is a comment.

If there is no space following the comment character, the application will not recognize it as a comment character. Therefore, method or class comments that end with a single comment character on an otherwise blank line will not be recognized.

4. The comment will always be preceded by either a blank line or the beginning of the file.

Figure 12 shows a class with a properly formatted comment. The comment is directly before the class definition and has no blank lines between the comment and the class definition.

```
# A controller used to access user-defined sets of Visits used in VisitSchedules
class VisitSetsController < ApplicationController
  protect_from_forgery :except => :new
```

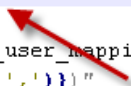


Class description comment

Figure 12: Ruby Class with descriptive comment.

Figure 13 shows a method definition with a properly formatted comment. Once again, the comment falls directly before the line containing the beginning of the method and there are no blank lines between the comment and the Def statement.

```
# Accepts a Hash consisting of issue_id and user_name and then
# re-assigns Issues to the Specified User or WorkQueue.
def self.bulk_assign_issues(issue_user_mappings)
  return [] if issue_user_mappings == nil or issue_user_mappings.keys.length == 0
  where_clause = "#{issue_user_mappings.keys.join(',')}"
  issues = Issue.find(:all, :conditions => "id in #{where_clause}")
  user_names = issue_user_mappings.values
```



Method Description Comment

Figure 13: Ruby method definition with descriptive comment.

4 Writing Good Comments

Each class should have a comment describing the class and how it fits into the application. The description should include general information about the class and any dependencies the class might have.

The following information should be included in class and method comments:

- Describe the purpose of the class or method.

Good: The `ConnectionStringSetting` class represents the configuration information for a database connection defined by the connection strings configuration section.

Bad: configuration information

- For methods, include a description of the parameters, if any, and if there is a return value, describe it too. (Note: The Microsoft Documentation comments allow parameters to be described using the `<param>` tag.)

Good: When the user clicks the browse button, this method will display the directory selection dialog box and allow the user to select the root directory to be scanned. All directories under the root will be scanned for the desired files as well. Parameters: Sender - The object that raised the click event. E - The event arguments.

Bad: Executed when the user clicks the button.

5 Revision History

Version	Date	Changed by	Description of Changes
1.0	October 2009	Irene Smith	Original Release
2.0	June 2010	Irene Smith	<p>The following changes were made to this document:</p> <ul style="list-style-type: none">• Changed title to "Preparing Code for Automated Design Spec. Tools"• Added information for commenting Ruby files.