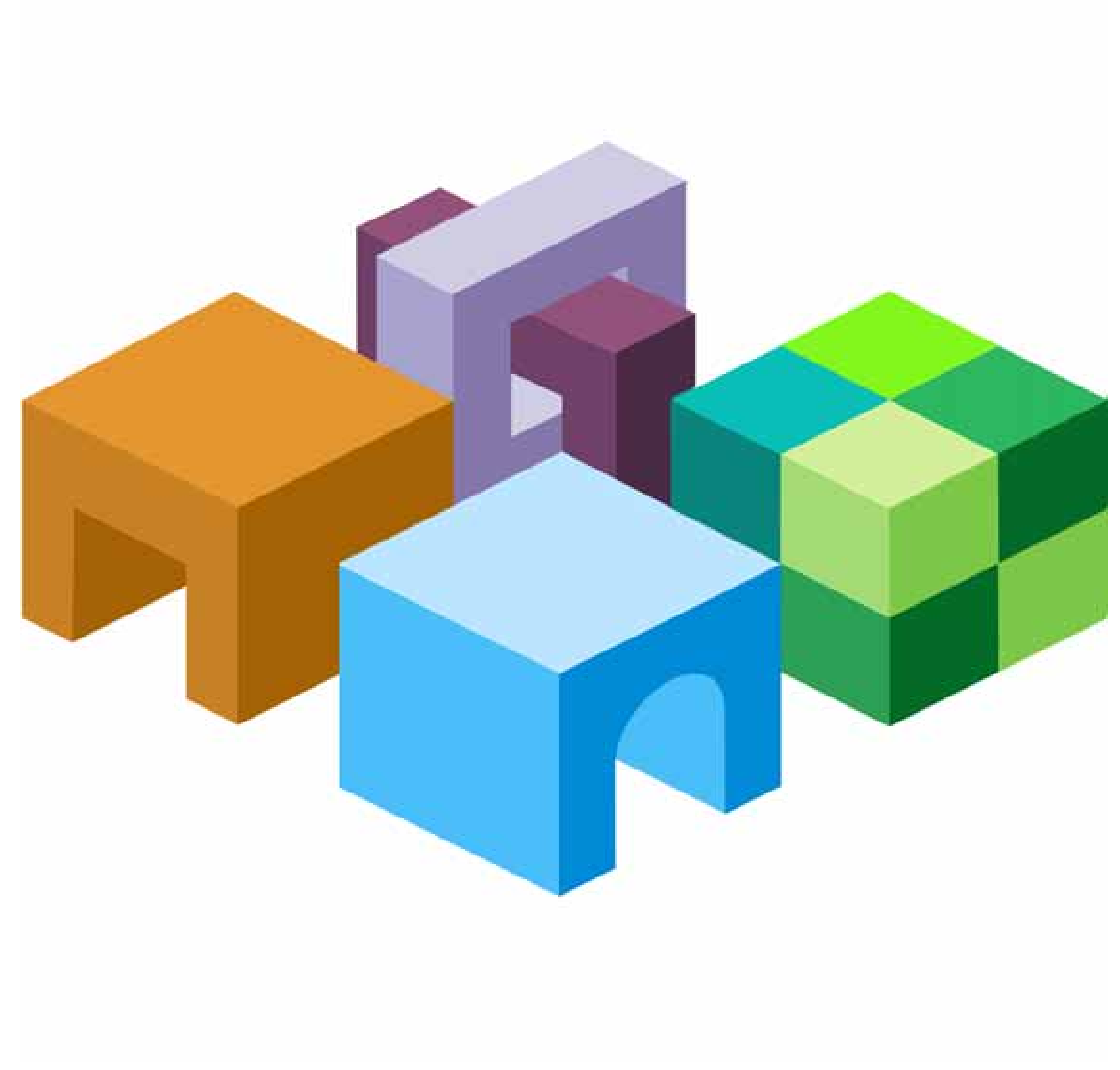


HYPERION® SYSTEM™ 9 BI+™

INTERACTIVE REPORTING™

RELEASE 9.2

OBJECT MODEL AND DASHBOARD DEVELOPMENT SERVICES DEVELOPER'S GUIDE

Hyperion

VOLUME VI: DASHBOARD ARCHITECT

Copyright 2000–2006 Hyperion Solutions Corporation.
All rights reserved.

“Hyperion,” the Hyperion logo, and Hyperion’s product names are trademarks of Hyperion. References to other companies and their products use trademarks owned by the respective companies and are for reference purpose only.

No portion hereof may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the recipient’s personal use, without the express written permission of Hyperion.

The information contained herein is subject to change without notice. Hyperion shall not be liable for errors contained herein or consequential damages in connection with the furnishing, performance, or use hereof.

Any Hyperion software described herein is licensed exclusively subject to the conditions set forth in the Hyperion license agreement.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Hyperion license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14, as applicable.

Hyperion Solutions Corporation
5450 Great America Parkway
Santa Clara, California 95054

Printed in the U.S.A.

Contents

Preface vii
Purpose vii
Audience viii
Document Structure viii
Related Documents viii
Where to Find Documentation ix
Help Menu Commands x
Conventions x
Additional Support xi
  Education Services xi
  Consulting Services xi
  Technical Support xii
Documentation Feedback xii

CHAPTER 1 Introduction and Concepts 13
JavaScript Concepts 14
  Definition of JavaScript 14
  How Interactive Reporting Studio Supports JavaScript 14
  Document—Level Customization 14
  Dashboard Section—Level Customization 15
Object Oriented Concepts 16
  Objects 16
  Examples of Interactive Reporting Studio Objects 16
  Methods 17
  Examples of Interactive Reporting Studio Methods 18
  Properties 18
  Examples of Interactive Reporting Studio Properties 18
  Addressing Objects, Properties, and Methods 18
Dashboard Architect Concepts 18
  The Dashboard Development Environment 19
  About Dashboard Architect 20

Architecture 21
Creation of a Project from an Interactive Reporting Document 21
About Editing JavaScript 22
Testing Capability 22
Debugging Capability 22
Synchronization with Interactive Reporting Studio 23
Recreation of an Interactive Reporting Document 23

CHAPTER 2 Dashboard Architect Features 25
The Dashboard Architect User Interface 26
Menu Commands, Shortcuts, and Buttons 27
The File Menu 28
The Edit Menu 28
The View Menu 29
The Project Menu 30
The Debug Menu 31
The Run Menu 31
The Tools Menu 32
The Help Menu 32
The Options Dialog Box 33

CHAPTER 3 Creating a Project 35
Creating Projects 36
Duplicating Projects 38

CHAPTER 4 Making an Interactive Reporting Document 39

CHAPTER 5 Editing 41
General Notes About Editing 42
Navigation by Using the Script Outliner 43
Code Generation by Using the Object Browser 43
The Find Dialog Box 45
The Search Feature 45
The Options Feature 45
The Find Next Option 48
The Find All Option 48
Using the Floating Find Menu 50
The Replace Feature 51
Using the Printing Command 51
Using the Match Brace Feature 52
The Auto–Code Feature 53

Macros 54
About Defining a Macro 55
Simple Macros 55
Multiple–Line Macros 56
Macro Parameters 56
Invoking Macros 58
Macro Control Codes 58
Importing Sections from Other Interactive Reporting Documents 60

CHAPTER 6 The Testing and Debugging Processes 63
About Testing 64
Testing Rule 64
Procedural and Declarative JavaScript 64
About Debugging 66
Breakpoints 66

CHAPTER 7 Adding and Removing Objects 69
Resynchronizing 70
Adding Controls 70
Duplicating Controls 70
Creating Controls 71
About Deleting Controls 72
About Renaming Controls 72
Adding and Duplicating Sections 72
About Renaming Sections 73
About Deleting Sections 74

CHAPTER 8 Documentation 75
Documentation of Code 76
Documentation Comments 76
Documentation of Variables 77
Documentation of Functions 78
Documentation of Classes 78
Documentation of Dashboard Development Services Components 80
Namespace Scope of Entities and the @scope Tag 80
Documentation Grouping by Using the @scope Tag 82
Documentation Comment Tag Reference 83
Dashboard Development Services Component—Specific Features 84

Generating Documentation 85
Inclusion and Exclusion of Documentation Groups 85
Inclusion and Exclusion of Unscoped Documentation 85
Dashboard Development Services Component—Specific Features in HTML Documentation 86

CHAPTER 9 Using the Dashboard Development Services Update Utility 87
About Dashboard Development Services Update Utility 88
Update Workflow 90
Creating New Sections Files 91
Updating New Sections Files 91
Configuring INI Files 92
Adding and Removing Sections 93
Updating Documents 94
Using the Upgrade One Method 94
Using the Upgrade Many Method 96
Command Line Updates 97
Updating Document Sets 98

Glossary 103

Index 105

Preface

Welcome to the *Hyperion System 9 BI+ Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 6: Dashboard Architect*. This preface discusses the following topics:

- "Purpose" on page vii
- "Audience" on page viii
- "Document Structure" on page viii
- "Related Documents" on page viii
- "Where to Find Documentation" on page ix
- "Help Menu Commands" on page x
- "Conventions" on page x
- "Additional Support" on page xi
- "Documentation Feedback" on page xii

Purpose

This guide provides an overview of Dashboard Architect, which is an integrated development environment for Interactive Reporting™ Studio™. Dashboard Architect enables you to swiftly test, debug, and build Interactive Reporting Studio applications, freeing up development time and increasing productivity. The guide explains the Dashboard Architect features and contains the concepts, processes, procedures, and examples that are required to use the software.

Audience

This guide is aimed at JavaScript programmers who build or maintain JavaScript applications that run under Interactive Reporting Studio.

What the guide is not:

- A JavaScript reference
- An introduction to JavaScript
- An introduction to programming

This guide describes how Dashboard Architect is used to develop applications in the context of Interactive Reporting Studio.

Document Structure

The Guide is one of seven books that explain how to use Hyperion System 9 BI+ Interactive Reporting Studio (see "Related Documents" on page viii). This book discusses the integrated development environment for Interactive Reporting Studio, that can be used to edit and debug JavaScript.

Related Documents

In addition to the *Hyperion System 9 BI+ Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 6: Dashboard Architect* the following documentation is available:

Hyperion System 9 BI + Interactive Reporting Studio User' s Guide provides an overview of Hyperion System 9 BI+ Interactive Reporting Studio and explains the user interface and basic commands. It includes how to retrieve data, how to query new data and change existing queries, and how to query a single database as well as multiple databases. It also covers how to work with query results.

Hyperion System 9 BI +Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 1: Dashboard Design Guide describes how to create custom applications in the Dashboard section, how to use JavaScript to script and control Hyperion System 9 BI+ Interactive Reporting Studio and web clients documents, how JavaScript programs are interpreted by the Interactive Reporting engine, how JavaScript programs are used to provide dynamic control of a Hyperion System 9 BI+ Interactive Reporting Studio document, how documents enhanced with JavaScript are able to respond to user interaction, and how JavaScript is used within Hyperion System 9 BI+ Interactive Reporting Studio and the web client to respond to user events and the document lifecycle.

Hyperion System 9 BI +Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 2: Object Model Guide to Objects and Collections describes the objects used in the Interactive Reporting Object Model.

Hyperion System 9 BI +Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 3: Object Model Guide to Methods describes the methods used in the Interactive Reporting Object Model.

Hyperion System 9 BI +Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 4: Object Model Guide to Properties and Constants describes the properties and constants used in the Interactive Reporting Object Model.

Hyperion System 9 BI +Interactive Reporting Object Model and Dashboard Development Services Developer's Guide, Volume 5: Dashboard Studio Dashboard Studio presents the Wizard that works with a set of extensible and customizable templates to create interactive, analytical dashboards without the need to write programming logic.

Where to Find Documentation

All Interactive Reporting documentation is accessible from the following locations:

- The HTML Information Map is available from the Interactive Reporting Help menu for all operating systems; for products installed on Microsoft Windows systems, it is also available from the Start menu.
- Online help is available from within Interactive Reporting after you log on to the product, you can access online help by clicking the Help button or selecting Help from the menu bar.
- The Hyperion Download Center can be accessed from the Hyperion Solutions Web site.

➤ To access documentation from the Hyperion Download Center:

1 Go to the Hyperion Solutions Web site and navigate to **Services > WorldWide Support > Download Center.**

Note: Your Login ID for the Hyperion Download Center is your e-mail address. The Login ID and Password required for the Hyperion Download Center are different from the Login ID and Password required for Hyperion Support Online through Hyperion.com. If you are not sure whether you have a Hyperion Download Center account, follow the on-screen instructions.

2 In the **Login ID** and **Password** text boxes, enter your e-mail address and password.

3 In the **Language** list box, select the appropriate language and click **Login.**

4 If you are a member on multiple Hyperion Solutions Download Center accounts, select the account that you want to use for the current session.

5 To access documentation online, from the Product List, select the appropriate product and follow the on-screen instructions.

Help Menu Commands

Table i describes the commands that are available from the Help menu in Interactive Reporting.

Table i Help Menu Commands

Command	Description
Help on This Topic	Launches a help topic specific to the window or Web page.
Contents	Launches the Interactive Reporting help.
Information Map	Launches the Interactive Reporting Information Map, which provides the following assistance: ● Online help in PDF and HTML format ● Links to related resources to assist you in using Interactive Reporting
Technical Support	Launches the Hyperion Technical Support site, where you submit defects and contact Technical Support.
Hyperion Developer's Network	Launches the Hyperion Developer Network site, where you access information about known defects and best practices. This site also provides tools and information to assist you in getting starting using Hyperion products: ● Sample models ● A resource library containing FAQs, tips, and technical white papers ● Demos and Webcasts demonstrating how Hyperion products are used
Hyperion.com	Launches Hyperion's corporate Web site, where you access a variety of information about Hyperion: ● Office locations ● The Hyperion Business Intelligence and Business Performance Management product suite ● Consulting and partner programs ● Customer and education services and technical support
About Interactive Reporting	Launches the About Interactive Reporting dialog box, which contains copyright and release information, along with version details.

Conventions

The following table shows the conventions that are used in this document:

Table ii Conventions Used in This Document

Item	Meaning
➤	Arrows indicate the beginning of procedures consisting of sequential steps or one-step procedures.
Brackets [ ]	In examples, brackets indicate that the enclosed elements are optional.

Table ii Conventions Used in This Document *(Continued)*

Item	Meaning
Bold	Bold in procedural steps highlights user interface elements on which the user must perform actions.
CAPITAL LETTERS	Capital letters denote commands and various IDs. (Example: CLEARBLOCK command)
Ctrl+0 Ctrl+Q, Shift+Q	Keystroke combinations shown with the plus sign (+) indicate that you should press the first key and hold it while you press the next key. Do not type the plus sign. For consecutive keystroke combinations, a comma indicates that you press the combinations consecutively.
`Example text`	Courier font indicates that the example text is code or syntax.
`Courier italics`	Courier italic text indicates a variable field in command syntax. Substitute a value in place of the variable shown in Courier italics.
`ARBORPATH`	When you see the environment variable *`ARBORPATH`* in italics, substitute the value of ARBORPATH from your site.
n, x	Italic *n* stands for a variable number; italic *x* can stand for a variable number or a letter. These variables are sometimes found in formulas.
Ellipses (...)	Ellipsis points indicate that text has been omitted from an example.
Mouse orientation	This document provides examples and procedures using a right-handed mouse. If you use a left-handed mouse, adjust the procedures accordingly.

Additional Support

In addition to providing documentation and online help, Hyperion offers the following product information and support. For details on education, consulting, or support options, click the Services link at the Hyperion Solutions Web site.

Education Services

Hyperion offers instructor-led training, custom training, and e-Learning covering all Hyperion applications and technologies. Training is geared to administrators, end users, and information systems professionals.

Consulting Services

Experienced Hyperion consultants and partners implement software solutions tailored to clients' particular reporting, analysis, modeling, and planning requirements. Hyperion also offers specialized consulting packages, technical assessments, and integration solutions.

Technical Support

Hyperion provides enhanced telephone and electronic-based support to clients to resolve product issues quickly and accurately. This support is available for all Hyperion products at no additional cost to clients with current maintenance agreements.

Documentation Feedback

Hyperion strives to provide complete and accurate documentation. Your opinion on the documentation is of value, so please send your comments by going to http://www.hyperion.com/services/support_programs/doc_survey/index.cfm.

1 Introduction and Concepts

This guide is aimed at JavaScript programmers who build or maintain JavaScript applications that run under Interactive Reporting Studio. This chapter describes how Dashboard Architect can be used to develop applications in the context of Interactive Reporting Studio.

In This Chapter

JavaScript Concepts. 14

Object Oriented Concepts 16

Dashboard Architect Concepts 18

Architecture. 21

JavaScript Concepts

This topic covers JavaScript concepts and the relationship with Interactive Reporting Studio. It discusses the differences between document–level and dashboard section–level customization.

Definition of JavaScript

JavaScript is an interpreted programming language, which enables executable content to be included in Interactive Reporting documents. This scripting language contains a small vocabulary and a simple programming model, which enables developers to create interactive and dynamic content by using syntax loops, conditions, and operations. While JavaScript is based on the syntax for Java, it is not Java.

How Interactive Reporting Studio Supports JavaScript

Interactive Reporting Studio supports the Netscape JavaScript Interpreter 1.4. Netscape JavaScript is a superset of the ECMA–262/ISO–16262 standard scripting language, with mild differences from the published standard. The code in the tree supports JavaScript 1.1, 1.2, 1.3, and 1.4. JavaScript 1.4 includes support for some ECMAv2 features (for example, exception handling and new switch behavior).

JavaScript enables developers to customize Interactive Reporting documents at the document, dashboard section (including dashboard controls), computed items, and menu levels.

Document—Level Customization

A document can include queries, tables, pivots, charts, reports, or dashboard sections. A dashboard section can be inserted into a document or can be standalone. Scripts can be included within Interactive Reporting documents to initiate application functionality, to process queries, activate sessions, and create filters. A document–level action is initiated:

- When the `OnStartUp` event is fired in a document
- When the `OnShutDown` event is fired in a document

JavaScript enables the developer to define event handlers. Event handlers are code that is executed when an event occurs. These events are usually initiated by the user when clicking a button or can be set to commence when a document is opened or when a document section is activated.

Dashboard Section–Level Customization

The dashboard is an interface for users to view data, create ad hoc queries, and print reports. Developers use templates, charts, or graphs to create user–friendly dashboard sections. These sections are the hypertext, push–button interface to the entire query and reporting process. A dashboard section–level action is initiated:

- When the `OnActivate` event is fired in a section
- When the `OnDeactivate` event is fired in a section

In a dashboard section, JavaScript controls the action and behavior of objects such as command buttons, list boxes, and graphic controls. Typically, these objects are scripted to perform an action based on an event, such as `OnClick`, `OnSelection`, or `OnDoubleClick`.

Dashboard Controls

- Command Buttons (`OnClick` event is supported)
- Option Buttons (`OnClick` event is supported)
- Check Boxes (`OnClick` event is supported)
- List Boxes (`OnClick`, `OnDoubleClick` events are supported)
- Drop–down List Boxes (`OnSelection` event is supported)
- Text Boxes (`OnChange`, `OnEnter`, `OnExit` events are supported)

Dashboard Graphics

- Lines (`OnClick` event is supported)
- Rectangles (`OnClick` event is supported)
- Round rectangles (`OnClick` event is supported)
- Ovals (`OnClick` event is supported)
- Text labels (`OnClick` event is supported)
- Pictures (`OnClick` event is supported)
- Results (`OnClick` and `OnRowDoubleClick` events are supported)
- Tables (`OnClick` and `OnRowDoubleClick` events are supported)
- Pivots and charts (`OnClick` event is supported)

Object Oriented Concepts

This topic explains key Object Oriented (OO) concepts, provides examples and discusses how to address objects, methods, and properties.

Objects

In Object Oriented Programming (OOP) objects represent the real world. An object is a software package that contains a collection of related characteristics and behaviors. These characteristics and behaviors are also known as properties and methods. Methods are also called *functions* and *event handlers*.

JavaScript objects can package multiple properties and methods together and provide built–in and dynamic objects, properties, and methods.

Properties are characteristics that describe data or other objects. For example, a door is an object in the real world and a door can be modeled in a software application.

An example of an object property is a door that can be in one of two states—open or closed. A door may or may not have a latch. A latch is also a property of the door but because a latch can be locked or unlocked, it is also an object in its own right.

Methods or event handlers are functions associated with objects that can alter the state of the object. For example, a door can be opened or closed. Open and Close are examples of methods, some sequence of events needs to happen so that the door can move from one state to another. In the real world, a person applies pressure to the door in a direction to close it. In Dashboard Architect, some routine animates the closing of the modeled door.

Objects, methods and properties define the characteristics of an object and its behavior.

Examples of Interactive Reporting Studio Objects

Within Interactive Reporting Studio, documents can be accessed from within the scripting environment as a hierarchy of objects. For example, the Interactive Reporting Studio button object contains the following properties and methods.

Table 1 Example Button Objects

Characteristic	Type	Description
Alignment	Property	Align text on button face–Left, Right, Center
Enabled	Property	Determine availability of an object to be clicked
Font	Property	Describe the look and style of the text displayed on a button face (an object in its own right that contains its own characteristics)
Name	Property	Object name to be addressed programmatically
OnClick	Method	Event handler called when a button is clicked
Text	Property	Text displayed on button face

Table 1 Example Button Objects *(Continued)*

Type	Property	A type of object (in this case bqButton)
Vertical Alignment	Property	Align text on button face—Top, Bottom, Middle
Visible	Property	Determines visibility of a button

The Font object described in the table contains the properties:

- Color (Red, Black, Blue, and so on)
- Effect (Superscript and so on)
- Name (Arial, Times New Roman, and so on)
- Size (10pt, 12 pt, and so on)
- Style (Bold, Italic, and so on)

A collection is another type of object supported by Interactive Reporting Studio and JavaScript. A collection usually contains a *count* property to show how many elements are parts of the collection.

A collection contains one or more methods. It provides a way of enabling access to the elements of the collection, sometimes to add elements to the collection, and other generic facilities to sanction change to the state of the collection. Different types of collections may behave differently.

The Interactive Reporting Studio application object is a collection that contains one or more documents which contain one or more sections. Each section contains other objects specific to it. For example, the query section includes:

- A data model
- A Request line
- A Filter line

The Filters include:

- A *Show Values* list
- A *Custom Values* list
- *Custom SQL*

Each object contains unique methods and properties that enable you to leverage the functionality of the object.

Methods

Methods are actions that change the state of the object. Programmers write JavaScript to determine the actions that are executed when an event is fired. Dashboard Architect is concerned almost entirely with the scripting of these actions, which typically involve the execution of one or more methods. For example, when a button is pressed, the Save method in the current document may be invoked.

Examples of Interactive Reporting Studio Methods

- The *Save* method of a document
- The *Process* method of a query
- The *Recalculate* method of a results section

Properties

Properties are characteristics that describe the objects.

Examples of Interactive Reporting Studio Properties

- The *Visible* state of a button or section
- The *Connected* state of a connection
- The *Size* of a font

Addressing Objects, Properties, and Methods

Properties and methods belong to objects and as properties can themselves be objects, this has the makings of a recursive–nested structure. Objects are referenced by including all the names in the chain from the *imaginary root* object right down to the object, property, or method being addressed.

For example, the User Property of a Connection Object in a data model of the query section of the Active Document is expressed as follows:

```
ActiveDocument.Sections[“Query”].DataModel.Connection.User
```

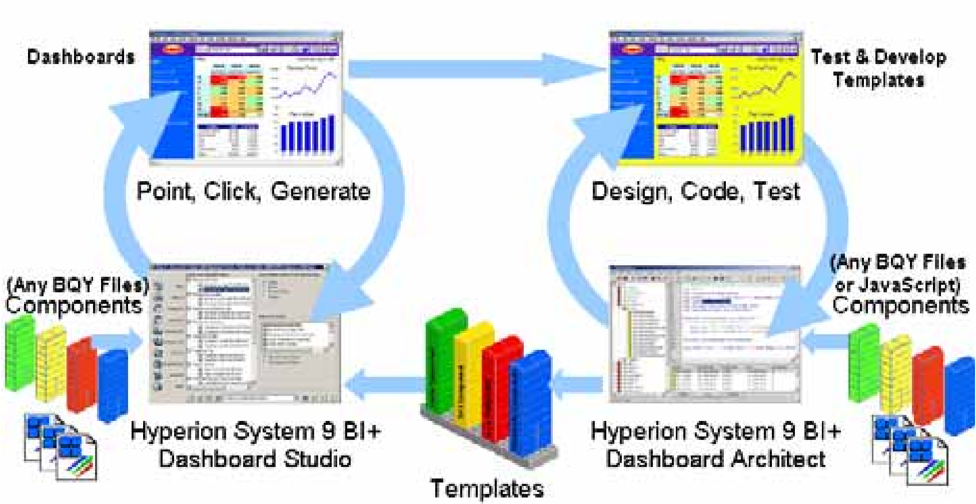
Interactive Reporting Studio and Dashboard Architect provide an object model tree view navigator that can generate this syntax on behalf of the user and ease the task of addressing objects, methods, and properties in this rather verbose – but necessary – way.

Dashboard Architect Concepts

This topic discusses the main Dashboard Architect concepts. These concepts include the environment in which it operates, a description of the software, and the architecture and functions of the program.

The Dashboard Development Environment

The following figure illustrates the Dashboard Development Environment and the role of Dashboard Architect within the test and development cycle.



The test and development cycle of Dashboard Studio templates is shown on the right side of the diagram. Dashboard Architect provides a 4GL–programming environment that uses JavaScript and leverages the object model of Interactive Reporting Studio. On the left side of the diagram, the platform provides a 5GL–development environment that requires no programming at all. Application development is through a template–driven, point–and–click wizard whose templates can be extended with plug–and–play components developed by using Dashboard Architect. See the *Hyperion System 9 BI+ Dashboard Development Services™ Components Reference Guide.*

Dashboard Studio templates contain frameworks that provide most of the basic functions required by analytical applications or dashboards. These functions are implemented as discrete components that can be plugged together in a variety of combinations to form templates.

Templates can be transformed into sophisticated applications through the use of Dashboard Studio, a point–and–click dashboard constructor.

The JavaScript at the core of these templates was developed initially by using the Interactive Reporting Studio JavaScript editor. As the size and complexity of the template grew, a more powerful development environment was required and from this Dashboard Architect was born.

Dashboard Architect is a JavaScript editor and debugger that is integrated with Interactive Reporting Studio. It can be used to create and maintain Interactive Reporting Studio JavaScript applications. Coupled with Dashboard Studio, it provides a great productivity advantage to users.

The right side of the Dashboard Development Environment diagram is optional, and applies only if Dashboard Architect is used to create templates. The left side of the diagram represents a lifecycle that most JavaScript application developers recognize, that of developing applications directly.

The Dashboard Development Environment was founded on the principles:

- A user capable of surfing the web can use a dashboard.
- A user capable of building an Interactive Reporting Studio chart or pivot can build sophisticated dashboards by using a point–and–click paradigm.
- A JavaScript programmer can build Interactive Reporting Studio applications and plug–and–play components in the most productive manner by using state–of–the–art facilities.
- Interactive Reporting Studio users can enjoy the quality and productivity benefits offered through the reuse of data and code sections.

About Dashboard Architect

Dashboard Architect is an integrated development environment for Interactive Reporting Studio. It enables you to swiftly test, debug, and build Interactive Reporting Studio applications, freeing up development time and increasing productivity.

Dashboard Architect enables programmers to work on their JavaScript alongside Interactive Reporting Studio in a fully interactive integrated manner. The main functions are:

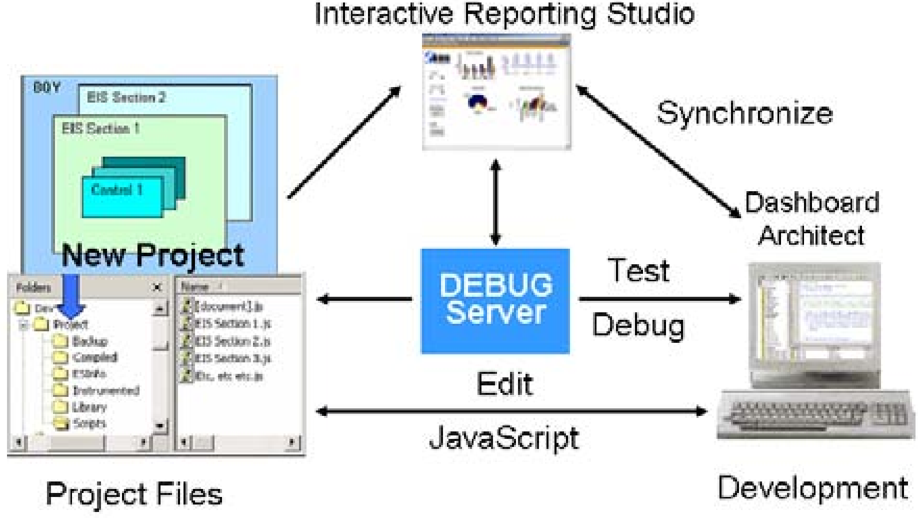
- Create a project by importing an Interactive Reporting document that contains JavaScript.
- Edit the JavaScript in a project by using the following:
  - Powerful search–and–replace
  - Individual and bulk formatting
  - Brace matching to create complete syntax
  - An object model browser to generate syntax
  - Undo and back–tracking
- Use Dashboard Architect to make changes to the Interactive Reporting document or synchronize the state of the Interactive Reporting document with the state of the project, where changes are made to the Interactive Reporting document outside of Dashboard Architect.
- Test and debug code with the assistance of breakpoints and stack traces to verify its correct operation and to fix faulty behavior.
- Make an Interactive Reporting document from the project and deploy it in production environments.

Note: Source code can be managed and version–controlled like other application sources, as it is stored externally in text files.

Architecture

Figure 1 depicts the Dashboard Architect structural design.

Figure 1 Product Architecture



The following sub-topics describe the links identified in this architecture diagram.

Creation of a Project from an Interactive Reporting Document

This is the starting point of a project. An Interactive Reporting document is composed of many sections, one or more of which may be a dashboard. Each dashboard section contains objects that expose scriptable events. The document may also contain JavaScript as part of its document events and its section events.

The regular Interactive Reporting document is disassembled and all JavaScript is read from events in the Interactive Reporting document and is stored in a Scripts folder as a text file—one for each dashboard section plus one for the document scripts. Provision is made for every event in the Interactive Reporting document even if they are empty.

From the regular Interactive Reporting document a file called *instrumented.bqy* is created. This file contains a full set of unmodified sections from the regular Interactive Reporting document including queries, results, tables, charts, pivots, and report sections.

It also contains altered or *instrumented* releases of the original dashboard sections. All of the visual elements remain unchanged but a special line of JavaScript is inserted into all object events in place of the original script. The special JavaScript calls the Debug Server to request its code. In response, the Debug Server sends JavaScript back to Interactive Reporting Studio appropriately in time for live execution of the code.

Note: A project can be duplicated. For the duplication to be successful, the entire directory tree that contains the project must be copied.

About Editing JavaScript

You interact with Dashboard Architect to view and edit the JavaScript that is now outside of the Interactive Reporting document. You have access to all the JavaScript through a window similar to Microsoft Windows Explorer. This includes a tree view of all the dashboard sections and all the objects and events in each dashboard as leaves of the tree.

Testing Capability

Code is available to the Interactive Reporting document and Interactive Reporting Studio in real time, even though it is held outside of the Interactive Reporting document. This happens because every Interactive Reporting document event is a call to the Debug Server asking for its code and passing itself as a parameter. Interactive Reporting Studio itself is used to execute the code. Buttons and items are clicked, the events are fired, and the event handlers call the Debug Server. The Debug Server examines the object and fetches the appropriate JavaScript and returns it to the event handler as a string. The event handler completes the operation by passing the returned string through to the JavaScript `eval()` function and so the latest code is parsed, interpreted, and executed. All this happens in real time. The Interactive Reporting Studio environment is thereby able to execute the code as if it were stored internally. This ensures that the document behaves as if it were running as usual, instead of being instrumented.

Debugging Capability

Debugging is very closely related to testing. When debugging in Dashboard Architect, you typically place one or more breakpoints in strategic places in the code that you are editing. This is so that execution is interrupted in the Interactive Reporting Studio and the current state of the application can be examined at a point within the application logic.

Adding a breakpoint modifies the way the regular JavaScript behaves. Instead of simply executing, code is inserted that makes calls back to the debugger to highlight the line that is about to be executed. When such a line is encountered, Dashboard Architect gives control back to you so the state of the properties and objects can be examined or modified in an attempt to understand the behavior of the code.

Synchronization with Interactive Reporting Studio

The Interactive Reporting Studio programming environment enables many powerful actions to be performed through its programmatic interface. However, it does not sanction the creation or deletion of objects on a dashboard, nor the dynamic addition of JavaScript into the event handlers of those objects.

Dashboard Architect provides a synchronization mechanism, to solve this problem. Actions that can be performed by using the Interactive Reporting Studio user interface only, can be detected and the Dashboard Architect structures can be adjusted to reflect the current state of `instrumented.bqy`. While this is not a seamless way to create and remove objects, it is nevertheless an easy–to–use and elegant mechanism.

Recreation of an Interactive Reporting Document

The final step in the development sequence is the recreation of a regular Interactive Reporting document. This is done by merging the code that was edited inside Dashboard Architect with the sections and controls within `instrumented.bqy`. This creates a document that is no longer connected to Dashboard Architect and which can be deployed in any production environment. No trace of Dashboard Architect can be found in this compiled Interactive Reporting document. See "Making an Interactive Reporting Document" on page 39.

2 Dashboard Architect Features

This chapter describes the Dashboard Architect interface features, including menu commands, shortcuts, buttons, and the Options dialog box.

In This Chapter

The Dashboard Architect User Interface 26

Menu Commands, Shortcuts, and Buttons 27

The Options Dialog Box 33

The Dashboard Architect User Interface

The main Dashboard Architect screen is divided into numbered segments, as illustrated in the following figure. Each segment is described in Table 2.

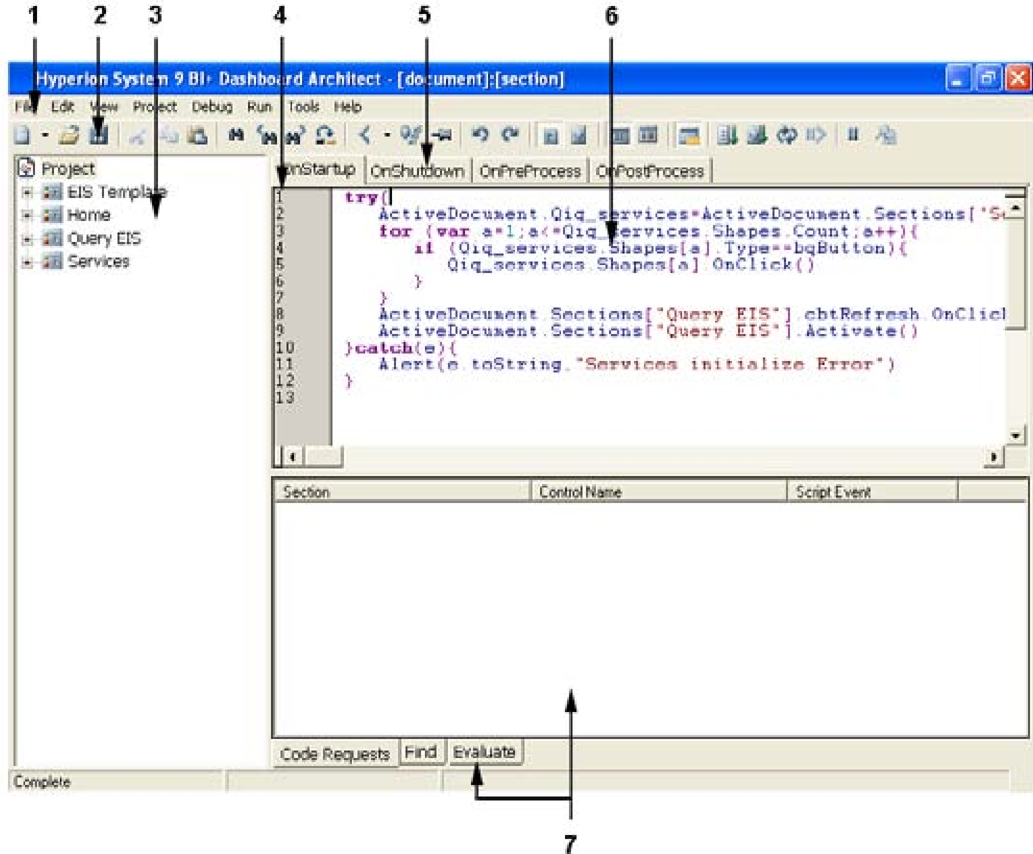


Table 2 Interface Descriptions

Item Label	Description
1. Menu Bar	Provides access to all Dashboard Architect features. Some menu bar entries provide commonly used keyboard shortcuts and some have buttons that activate the functionality.
2. Toolbar	Gives one-click access to the most commonly needed functionality.
3. Navigation Panel	This panel may show: 1. The list of dashboard sections and the programmable elements within them in the form of a tree view that can be expanded and contracted. In this mode the editing window (6) changes to show the JavaScript associated with the selected programmable element. 2. The list of Interactive Reporting Studio objects currently available in the instrumented Interactive Reporting document in the form of an expandable tree view. If you double-click one of the objects, methods, or properties, the syntax associated with the object, method, or property is added at the cursor position in the editing window (6).
4. Line Number	The line number indicator.

Table 2 Interface Descriptions *(Continued)*

5. Event Handlers	Each event handler for a section or control is shown as a content tab at the top of the editing window.
6. Editing Window	This window is where the JavaScript can be edited or where code execution can be examined by using breakpoints.
7. Output Window and Content Tabs	The output window contains three content tabs. 1. Code Requests—tracks the calls made by the `instrumented.bqy` to the Debug Server for the JavaScript that each object needs to evaluate. If an error is detected, the last icon in the list is displayed with the red cog over the yellow folder icon. Clicking a line of code in the output window causes the editing window (6) to show the JavaScript for that event handler. 2. Find—shows the list of all the lines matching a find specification. Clicking a line in the find list causes the editing window (6) to show the JavaScript for that event handler and the cursor is placed where the match was made on the line. 3. Evaluate— enables you to interact with `instrumented.bqy` while execution is paused on a breakpoint. It contains two sub-panes, one in which JavaScript expressions can be entered for evaluation and another in which the results of the evaluation are shown. This window also contains buttons that are useful when debugging. Within the evaluate pane, you can Cut, Copy, or Paste an expression to be evaluated, or click Clear to delete the expression from this window. **Note:** The output window is visible only if Toggle Output Window is selected on the toolbar or Crtl+W is clicked.

Menu Commands, Shortcuts, and Buttons

This topic provides a comprehensive list and description of the Dashboard Architect menu commands, buttons, and shortcuts.

- The File Menu
- The Edit Menu
- The View Menu
- The Project Menu
- The Debug Menu
- The Run Menu
- The Tools Menu
- The Help Menu

The File Menu

The commands available in the File menu are discussed in Table 3.

Table 3 File Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
New		Ctrl+N	Create a project file from an Interactive Reporting document. This creation is the starting point of a project.
Open		Ctrl+O	Open a project to edit and test its code.
Close			Close the currently open project.
Save		Ctrl+S	Save the currently open project. This saves the JavaScript and the associated `instrumented.bqy`.
Print		Ctr+P	Print a list of the JavaScript for the whole project or just a part of it.
Print Setup			Configure printer settings.
Make BQY			Create a regular Interactive Reporting document from `instrumented.bqy` and the JavaScript held in the Dashboard Architect script folder.
<numbered items>			Open a recently used project.
Exit		Ctrl+Q	Close the project and the application.

The Edit Menu

The commands available in the Edit menu are discussed in Table 4.

Table 4 Edit Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
Undo		Ctrl+Z	Revert the state of the document to a previous state. The undo command operates only on actions in the current event. Moving to another event handler and using the Find, Replace, and Save operations causes discontinuity in the undo log.
Redo		Ctrl+Y	The opposite of undo.
Cut		Ctrl+X	Regular Microsoft Windows behavior with regard to selected text.
Copy		Ctrl+C	Regular Microsoft Windows behavior with regard to selected text.

Table 4 Edit Menu Descriptions *(Continued)*

Paste		Ctrl+V	Regular Microsoft Windows behavior with regard to selected text.
Find		Ctrl+F	Activate the Find dialog box and specify the search criteria to match a text string or a pattern.
Find Next		F3	Find the next instance of a matched text string.
Find Previous		Ctrl+F3	Find the prior instance of a matched text string.
Replace		Ctrl+H	Activate the Replace dialog box and specify the criteria to find one or more occurrences of a string. Each match is replaced with a different string.
Match Brace		Ctrl+B	Starting at the current position, locate the next opening parenthesis ((), brace ({), or bracket ([) and find its closing counterpart. If no opening ((), ({), or ([) exists, it scans backwards until one is found and moves forward and finds its closing counterpart. If a match is found, the area between the opening and closing items is highlighted.
Go to Row			Move the cursor to the nominated row number in the current event handler.

The View Menu

The commands available in the View menu are discussed in Table 5.

Table 5 View Menu Descriptions

Command	**Button** (where applicable)	**Shortcut** (if applicable)	**Description**
BQY Script Outline			Turn Object Model mode off and show a list of dashboard sections and their objects in the navigation panel.
Object Model			Turn BQY Script Outline mode off and replace the navigation panel with the Object Model browser.
All Events			Show all objects with event handlers in the navigation panel. **Note:** This command applies when in BQY Script Outline mode.
Scripted Events			Show objects with event handlers that are not empty in the navigation panel. **Note:** This command applies only in BQY Script Outline mode.

Table 5 View Menu Descriptions *(Continued)*

Output Window		Ctrl+W	Hide or reveal the output window.
Back			Go back to the place where the cursor was last clicked within a script. The button includes a drop-down arrow which shows a list of recently visited events for direct navigation to a prior location.

The Project Menu

The commands available in the Project menu are discussed in Table 6.

Table 6 Project Menu Descriptions

Command	Button (where applicable)	Description
Import		Import one or more Interactive Reporting Studio sections from an external Interactive Reporting document. Dashboard sections are instrumented as part of the import.
Add Section		Create a dashboard section in `instrumented.bqy` and a matching set of event handlers in Dashboard Architect.
Rename Section		Rename an Interactive Reporting Studio section. When renaming a dashboard section, the corresponding event handlers in Dashboard Architect are renamed.
Delete Section		Delete a section. When deleting a dashboard section, the corresponding event handlers are deleted in Dashboard Architect.
Add Control		Open a dialog box detailing the process required to create an object on a dashboard. Interactive Reporting Studio does not sanction the creation of this type of object programmatically so the operation involves a two-step process beginning in Dashboard Architect and concluding in Interactive Reporting Studio.
Resynchronize		This function closes `instrumented.bqy`, examines it and instruments new objects. The function synchronizes the state of the event handlers within Dashboard Architect with the current state of `instrumented.bqy`.
Include Control Documentation		This command indicates whether documentation for variables and functions, that are not scoped outside a control, are included in the generated documentation.
Generate Documentation		Generate the HTML documentation for the current Interactive Reporting document.

The Debug Menu

The commands available in the Debug menu are discussed in Table 7.

Table 7 Debug Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
Toggle Breakpoint		F9	Turn a breakpoint on or off. You may must select Run > Fire Current Event before the breakpoint takes effect.
Clear All Breakpoints			Turn off all breakpoints.
Evaluate			This command is available only when paused on a breakpoint. The command evaluates the expression in the left text box of the evaluate pane in the output window. This must be a valid JavaScript expression, otherwise a JavaScript error is generated and execution ceases at the breakpoint.
Stack Trace			This command is available only when paused on a breakpoint. This shows the sequence of function calls that lead to the breakpoint.

The Run Menu

The commands available in the Run menu are discussed in Table 8.

Table 8 Run Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
Fire Current Event			This command causes the current event handler to be called, triggering evaluation of its JavaScript. This command is used to check for syntax errors and to update definitions and breakpoints in functions.
Fire OnStartUp Event			This command causes the `OnStartUp()` event of the document to be fired.
Fire Document Load			Save and close `instrumented.bqy` and reopen it.
Continue		F5	Continue past the current breakpoint.

The Tools Menu

The commands available in the Tools menu are discussed in Table 9.

Table 9 Tools Menu Descriptions

Command	Button (where applicable)	Description
Options		Display the Options dialog box. Refer to "The Options Dialog Box" on page 33, for further information.
Reload Macro Definitions		Reload the macro definitions from the macros.txt file. This command is useful when you develop, test, and modify macros.
Interactive Reporting Studio Type Library Diagnostics		Open the Installation Diagnostics window.
Line numbers		Show or hide line numbers in the editing window. **Note:** There is a discrepancy between the line numbers shown in the Console window and the line numbers shown in Dashboard Architect when working with an instrumented Interactive Reporting document. To translate the line number reported in the Console window: ❍ Subtract 2 for line numbers if there are no breakpoints in the current event, or the breakpoints come after the line being reported. ❍ Subtract a further 5 for each breakpoint.
Always On Top		This command causes the Dashboard Architect window to always be on top. **Note:** Tooltips are unavailable when running in this mode.
		This command causes the application to behave as a typical window.

The Help Menu

The commands available in the Help menu are discussed in Table 10.

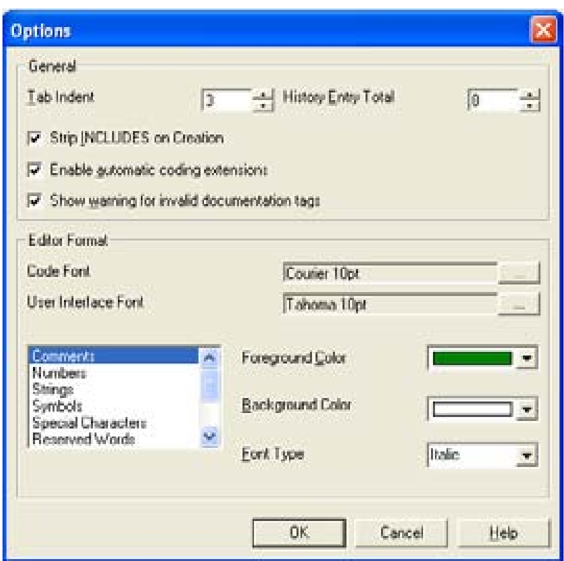
Table 10 Help Menu Descriptions

Command	Description
Contents	The Dashboard Architect help file is displayed.
About BI+ Dashboard Architect	The product release information is displayed.

The Options Dialog Box

The Options dialog box enables you to alter the look and feel of the Dashboard Architect interface.

Select Tools > Options to display the Options dialog box.

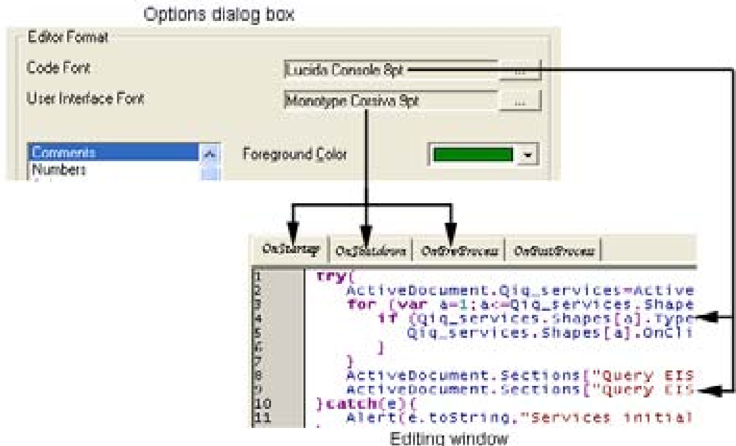


The settings that can be changed by using the Options dialog box are outlined below:

- The indentation of the current line in the editing window is applied to the next line if Tab Indent is not altered. This figure represents the number of spaces to add when the Tab key is pressed.
- History Entry Total contains a limit of 20 entries. Therefore, you can access up to 20 event handlers that were previously visited.
- If Strip INCLUDES on Creation is enabled, all references to INCLUDEs are removed when a project is created.
- Enable automatic coding extensions enables or disables the Auto–Code feature. See “The Auto–Code Feature” on page 53.
- Show warning for invalid documentation tags enables you to enable or disable the warning dialog box that is displayed when an invalid documentation tag is encountered. See “Documentation” on page 75.
- Code Font enables the code text displayed in Dashboard Architect to be customized. Select a font, font style, and size from the Font dialog box, and apply it to the code. For example, in Figure 2, the code in the editing window is set as *Lucida Sans Typewriter 9pt.*

- User Interface Font enables the Dashboard Architect graphical user interface font to be customized. For example, in Figure 2, the Event Handler tabs, above the editing window, bear the font *Lucida Handwriting 9pt*.

Figure 2 Code Font and User Interface Font Examples



- The remaining commands under Editor Format enable modification of the colors and font type used for syntax highlighting. Select a syntax format from the list box, and apply color and font type.

Note: Color is applied to text only as it is being viewed. It is not stored with the text in the JavaScript files.

3 Creating a Project

This chapter explains the process of creating or duplicating a project. It is the starting point for developing or maintaining code with Dashboard Architect. Dashboard Architect does not create Interactive Reporting documents, Interactive Reporting Studio sections, or dashboard objects, because there is no API facility enabling it to do so.

Dashboard Architect complements the strengths of Interactive Reporting Studio and provides value in the scripting and debugging of event handlers in dashboard sections.

In This Chapter

Creating Projects 36
Duplicating Projects. 38

Creating Projects

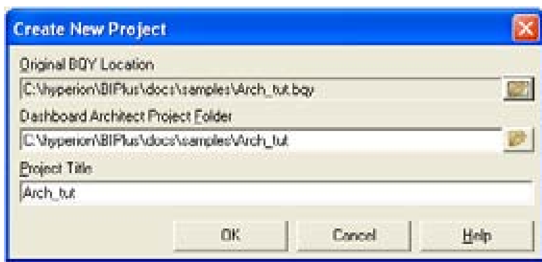
➤ To create a project:

1 **Start Interactive Reporting Studio, on the Welcome to Hyperion window, click Cancel.**

It is recommended that Interactive Reporting Studio be started first to prevent the flicker caused by its menu bar being hidden and revealed whenever section level interactions take place between it and its COM clients. Dashboard Architect is a Interactive Reporting Studio COM client.

2 **Start Dashboard Architect, and select File > New, or click .**

The Create New Project dialog box is displayed.



3 **Click , next to the Original BQY File Location text box, and navigate to the location of the Interactive Reporting document that contains the JavaScript to be to maintained or redeveloped.**

The Interactive Reporting document selected for this example is `Arch_tut.bqy`. This file is available in the Samples directory.

The Dashboard Architect Project Folder text box is filled with a folder location, which assumes that you want to create a Project folder in the folder that contains the initial Interactive Reporting document, and with the same name. The Project Title is automatically added to reflect the Original BQY File chosen. Use the default title or specify a different title.

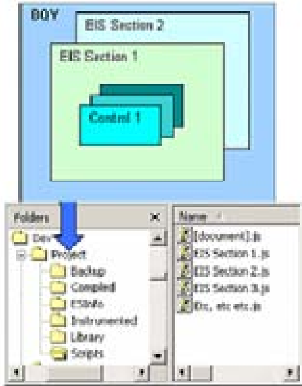
4 **To override the default project folder, click to navigate to another folder.**

The Project folder must be empty because files and folders are added to it to create the folder structure.

5 **Click OK.**

Dashboard Architect examines the selected Interactive Reporting document, disassembles it, and creates a JS file for each dashboard section, in which it makes provision for event handlers for all dashboard objects. JavaScript found in the Interactive Reporting document is placed in the event handler locations within the JS files.

Figure 3 New Project Structure



Dashboard Architect creates an `instrumented.bqy` file, which looks just like the original Interactive Reporting document, except the JavaScript within it is replaced with instrumented debug code. The debug code calls the Debug Server, which returns the JavaScript code from the JS files to the Interactive Reporting document under test, in real time.

The `instrumented.bqy` behaves just like the original Interactive Reporting document in the presence of the Debug Server and Dashboard Architect. The original Interactive Reporting document is left unchanged and a copy is placed in the Backup folder. The file name takes the form of *ORIGINAL–<date and time>*. For example, `ORIGINAL–2004–10–09–18–26–40.bqy`.

At the end of the development cycle, an Interactive Reporting document is recreated from the JavaScript in the JS files and `instrumented.bqy`.

The contents of the Project folder structure is described here.

- Backup holds the original Interactive Reporting document file that was used to create `instrumented.bqy`.
- Compiled holds the starting Interactive Reporting document. Additional Interactive Reporting document files are added each time you select the Make BQY command from the File menu.
- Instrumented holds `instrumented.bqy`.

- Scripts holds one JS script file for each section and one for the document.
- Library, ESInfo, and Doc are reserved for future use.

After the instrumentation process is complete, Dashboard Architect displays the JavaScript for the created project. The project is now ready for testing and editing.

Duplicating Projects

This procedure follows on from "Creating Projects" on page 36.

To duplicate a project, the entire directory tree that contains the project must be copied.

➤ To duplicate a project:

1. **Open Microsoft Windows Explorer.**
2. **Navigate to the location of the project to be duplicated.**

   For example, navigate to the `Arch_tut` folder.
3. **Select the entire directory tree, including the project file, the instrumented Interactive Reporting document, and all the JavaScript files.**

   In short, select everything that the project requires to work.

   Each individual Dashboard Architect project is self–contained and can be copied with no side–effects.
4. **Right-click, and select Copy.**
5. **Create a folder to paste the duplicate project directory into.**

   For example, *`Arch_tut_copy`*.
6. **Paste the copied directory into a folder.**

   For example, paste the files into the folder called *Arch_tut_copy*.
7. **Open the duplicate project file in Dashboard Architect as it is now available for use.**

   The duplicated project functions just as the original does.
8. **Optional:** An alternative duplication method for developers who use source code control systems, is to use a branch in the source code control system. This option is a more traditional software development technique.

4 Making an Interactive Reporting Document

This chapter explains the process of making an Interactive Reporting document (BQY).

At the end of the development process, Dashboard Architect recreates a regular Interactive Reporting document from the `instrumented.bqy` file and the updated JavaScript from the JS files.

It replaces the instrumented debug code with the JavaScript from the JS file for each event in each dashboard. This regular Interactive Reporting document can now run in environments where Dashboard Architect is not available, as a regular Interactive Reporting document. For example, it can be deployed:

- In client environments; for example, Interactive Reporting Studio
- In plug–in environments; for example, Interactive Reporting Web Client™

➤ To create an Interactive Reporting document of the project that is currently open in Dashboard Architect:

1 **Select File > Make BQY.**

The Save As window is displayed. By default the file is saved into the folder which was specified when the project was created. A backup of the file is also created and stored in the Compiled folder of the project.

2 **Click Save.**

No trace of Dashboard Architect instrumentation remains in the compiled Interactive Reporting document.

5 Editing

Edit code in an Interactive Reporting document by making changes in the editing window. Editing the JavaScript in Dashboard Architect is very similar to most editing operations that use programs such as UltraEdit, VisualBasic, and Notepad.

In This Chapter

General Notes About Editing 42
Navigation by Using the Script Outliner 43
Code Generation by Using the Object Browser 43
The Find Dialog Box 45
Using the Floating Find Menu 50
The Replace Feature 51
Using the Printing Command 51
Using the Match Brace Feature 52
The Auto-Code Feature 53
Macros 54
Importing Sections from Other Interactive Reporting Documents 60

General Notes About Editing

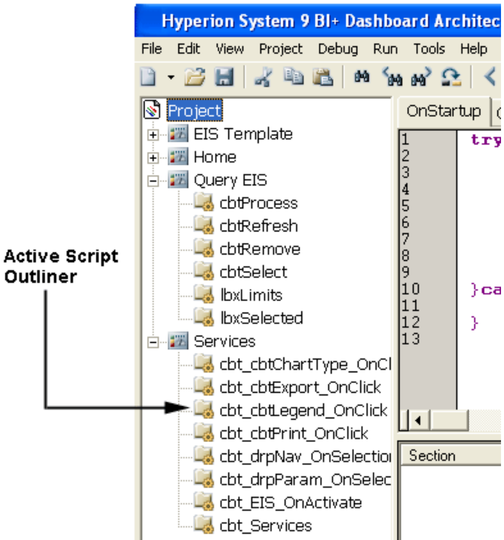
The changes made in the editing window are saved to the JS files in the Scripts folder when the next Save operation is performed. Save operations are performed explicitly when [save icon] is clicked or when the document is reloaded. Code changes you have made are kept in memory and are made available to the Interactive Reporting document when its event handlers request code through the Debug Server.

The keyboard and buttons can be expected to behave in very familiar ways. Some special features to notice are:

- Syntax is color–coded. Dashboard Architect uses a dictionary of reserved words to control how it applies color to parts of the syntax. This color–coding is controlled by the Options dialog box, which is displayed through the Tools menu. The color is applied to the text only as it is being viewed. The color is not stored with the text in the JavaScript files. See "The Options Dialog Box" on page 33.
- Indentation applied to the current line is also applied to the next line. The number of spaces to add when the Tab key is pressed is controlled by the Options dialog box. See "The Options Dialog Box" on page 33.
- Shift+Enter and Enter are not identical. Enter terminates a line of JavaScript. Shift+Enter does not terminate a JavaScript statement; it continues it on the next line. Shift+Enter is not recommended, because breakpoints cannot be set across these continuation lines. If a breakpoint is added to these lines, a JavaScript syntax error is generated. See "Breakpoints" on page 66.
- Back operations take you to event handlers previously visited. Up to 20 event handlers are kept in the execution stack for back–tracking purposes.
- Operations such as Save, Find, Replace, Fire Current Event, Fire StartUp, Reload Document, and navigation to another event handler cause a discontinuity in the undo log. After these operations are executed, undo cannot go past them. You can undo typing, cutting, deleting, and pasting up until, but not past, a point of discontinuity.

Navigation by Using the Script Outliner

The following figure illustrates the script outliner view. The script outliner is used for navigation in Dashboard Architect.



Dashboard Architect uses a mechanism similar to Microsoft Windows Explorer to assist in navigating the project.

Click and select an object in the navigation panel. The JavaScript for the selected object is displayed in the editing window.

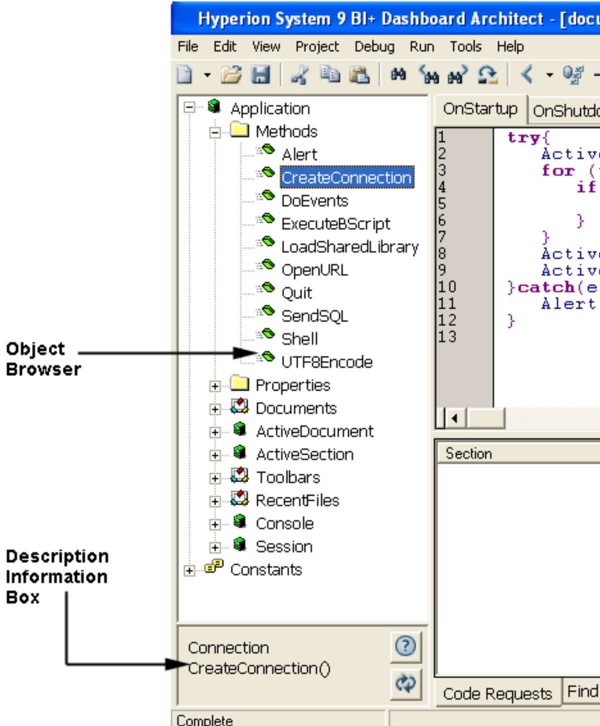
The contents of the editing window can be changed by clicking a row in the Code Requests or Find panes of the output window, in the lower part of the display.

Code Generation by Using the Object Browser

Click to replace the navigation panel with an object browser. This action enables the exploration of the objects and properties that Interactive Reporting Studio exports through its COM interface. The set of objects and properties is very similar—but not identical to—the object browser view available in Interactive Reporting Studio.

Expand and contract the nodes in the tree by clicking [+] or [-]. Or generate the object or property reference in the editing window by double–clicking the object or property.

Figure 4 The Active Object Browser



The object or property declaration is displayed underneath the object browser in the description information box. The following table gives details about the buttons available in the description information box.

Table 11 Description Information Box Buttons

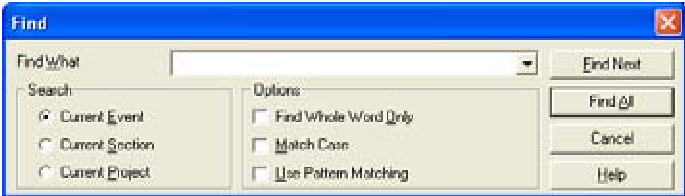
Command	Button	Description
Interactive Reporting Studio Help		Click to open the Interactive Reporting Studio help file with additional information about the currently selected member.
Reset Object Browser		Click to collapse the object tree and clear the local cache. Obtaining the object information is an expensive operation, so the retrieval is done in real time. Dashboard Architect caches the results. If the Interactive Reporting document changes, the cache may be out-of-date and should be reset. For example, a filter or computed item may be added in the part of the tree that is expanded. Refreshing or resetting enables you to view the object by starting from the top.

The Find Dialog Box

Access the Find dialog box from the menu by selecting Edit > Find or by clicking .

The Find dialog box can be divided into three parts:

- Search conditions (The Search Feature).
- Extent of the Find operation (The Options Feature).
- Search preferences; that is, find the next instance or find every instance (The Find Next Option and The Find All Option).



To use the Find feature, enter the string or pattern to look for in the Find What drop–down list box, and click Find Next or Find All. Or use the drop–down list box to select from previously entered Find criteria.

The Search Feature

The scope of the Find operation can be limited to these areas:

- The event handler currently on display
- The dashboard section of which the event handler is a part
- The entire project

The Options Feature

Several options are available for focussing a Find process. These options are discussed in the following topics:

- Find Whole Words Only
- Match Case
- Use Pattern Matching

Find Whole Words Only

This option means whole elements of JavaScript syntax are searched, not sets of characters separated by spaces. This example contains seven separate words:

```
ActiveDocument.Sections["Query"].Limits["Cust_Id"].SelectedList.Count
```

Match Case

This option locates instances of uppercase and lowercase letters, depending upon how the string or pattern was entered in the Find What list box.

Use Pattern Matching

Pattern matching is implemented through regular expressions just as JavaScript recognizes them. A substantial discussion of regular expressions is outside the scope of this guide, because regular expressions are a language unto themselves. For more information, the following resource is available on the internet:

Visit WebReference.com: http://www.webreference.com/js/column5/

Strings and numbers usually represent themselves. Some characters have special meaning and are meta characters. To use these characters in their literal sense, you must escape them with a backslash(\).

The following table shows examples of characters.

Table 12 Characters

Character	Explanation
0–9a–zA–Z	The character itself
.	A character other than a new line
*	Zero or more occurrences of the previous item
+	One or more occurrences of the previous item
?	Zero or one occurrence of the previous item
\f	Form feed or new page
\n	New line
\r	Carriage return
\t	Tab
\/	/
\\	\
\.	.
\Xnn	An ASCII character specified by the hex nn
\w	A word character (a–zA–a0–9)
\W	A non-word character
\s	A white space character
\S	A non-white space character
\d	A digit 0–9

Table 12 Characters *(Continued)*

<table>
<tr><td>$</td><td>End of string—in this case, as the last character in an event handler</td></tr>
<tr><td>^</td><td>Match the start of the string—in this case, the first character in an event handler</td></tr>
<tr><td>|</td><td>Or = an alternate set</td></tr>
<tr><td>(...)</td><td>A grouping</td></tr>
</table>

The following table shows examples of patterns.

Table 13 Patterns

<table>
<tr><th>Pattern</th><th>Explanation</th></tr>
<tr><td>if|else</td><td>Match if or else</td></tr>
<tr><td>m.*(Parent|Name
)</td><td>Match m followed by one or more characters and Name or m followed by one or more characters and Parent</td></tr>
<tr><td>Active.*Name</td><td>Match anything that contains the string Active, followed by one or more characters followed by Name</td></tr>
<tr><td>.*\}</td><td>Match a string that contains zero or more characters before a closing brace (}).
A brace is a special meta character in regular expressions, and it must be escaped by using a backslash (\).</td></tr>
<tr><td>.+\}</td><td>Match a string that contains one or more characters before a closing brace (})</td></tr>
<tr><td>\{\r\n</td><td>Match a line that contains a brace ({) followed by a carriage return and a line feed</td></tr>
<tr><td>e\r\n|e$</td><td>Match a line that ends in an e or an event handler that ends in an e</td></tr>
<tr><td>\r\nf|^f</td><td>Match a line break that is followed by an f, or an event handler that starts with f</td></tr>
</table>

Dashboard Architect supports the Find and Replace utility for the following symbols when pattern matching is *not* enabled.

Table 14 Symbols

Symbol	Function
^t	Matches a Tab character
^p	Matches a new line (CR/LF) (paragraph)
^^	Matches a "^" character

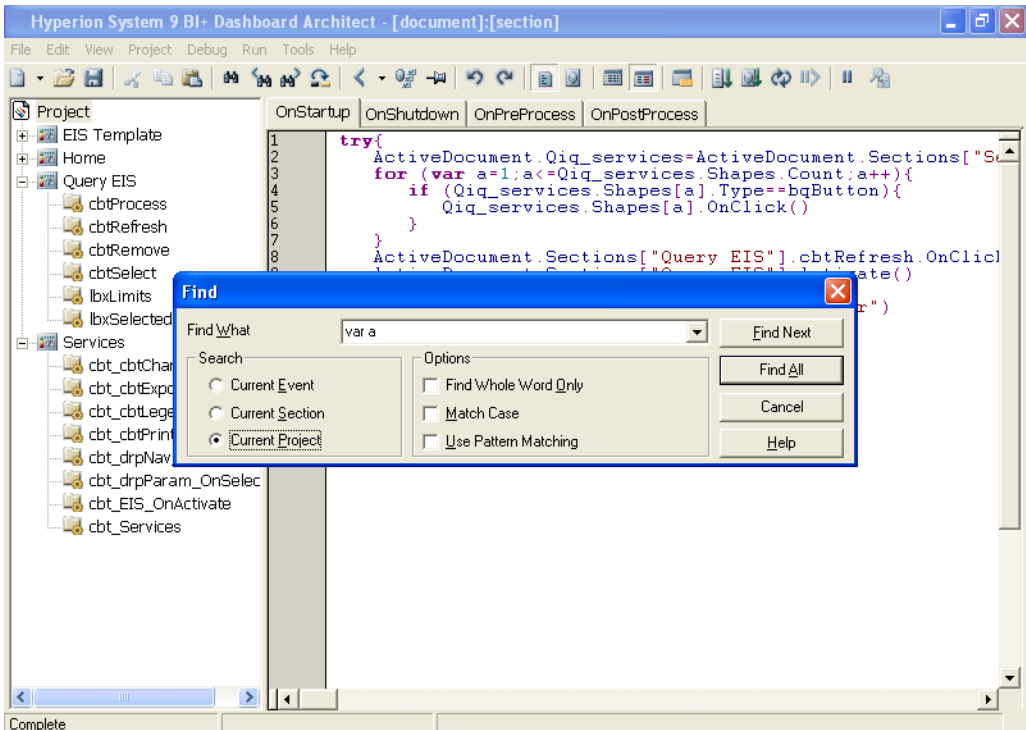
The Find Next Option

This option is used to move to the next instance of a search. The Find dialog box can be made active while you are editing. Move to the next found item by using the button in the dialog box, the menu item, on the toolbar, or the F3 key.

The Find All Option

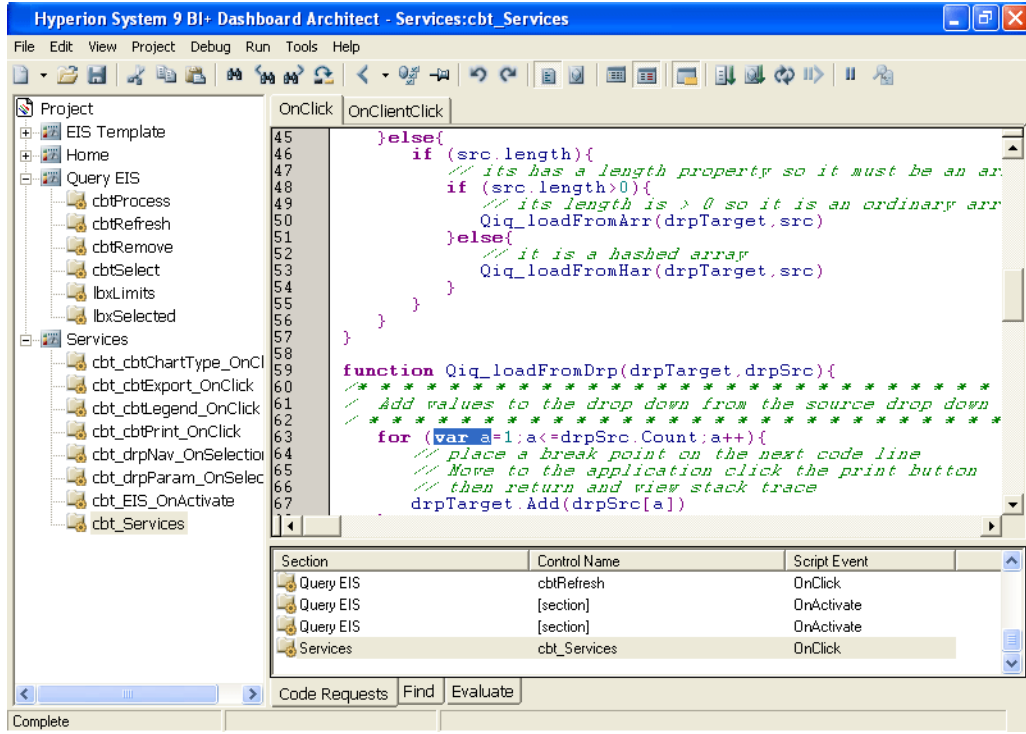
If this button is clicked, a Find operation is executed across the whole Search context (event, section, or project) and a single list of *find results* is returned. To view these results click the Find tab in the output window. For example, the following figure shows a request to find all instances of `var a` in the entire Arch_tut project.

Figure 5 Example of a Find Request



The output window lists the set of items found in the project. The output window can be used as a navigation mechanism. Click a line of interest, and Dashboard Architect displays the event handler, the line in question, and highlights the *found* instance in the editing window. For example, to see the code associated with an instance found, click the line under the Find tab, in the output window. The result is shown in Figure 6.

Figure 6 The Editing Window Displays Code Selected in the Output Window



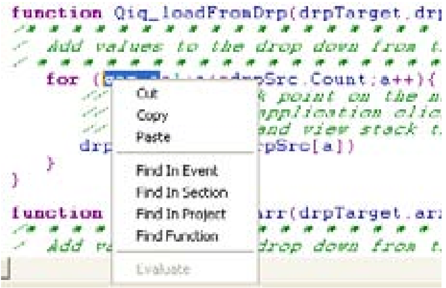
Using the Floating Find Menu

The floating menu provides a number of Find operations.

To find a simple string, highlight the string in the editing window. Right–click, and select the extent of the Find operation.

Dashboard Architect finds the specified string, and the results are presented in the output window.

Figure 7 The Floating Find Menu



The Find Function command is an example in floating Find menu. This command provides a quick and convenient way to find the definition of a function that is being called.

➤ To use the Find Function command:

1. **Highlight the text that corresponds to a function name.**
2. **Right–click, and select the Find Function command.**

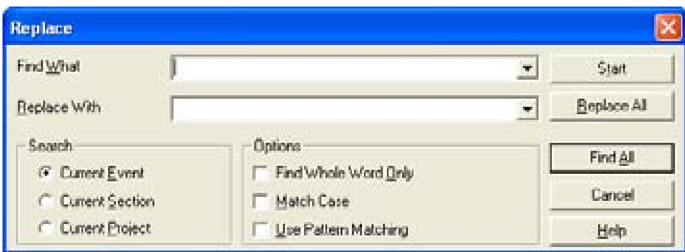
   Dashboard Architect prefixes the Find string with the word *function* and initiates a project–wide Find operation. The editing window displays the start of the function that is found, and the function is examined.

3. **After the examination is complete, return by clicking , to the launch point of the Find operation.**

The Replace Feature

The Replace feature is very similar to Find. In addition to finding text or patterns, it replaces the matched text with the text specified in the Replace With drop–down list box.

Access this feature by clicking Replace or choosing Edit > Replace. The Replace dialog box is a subset of the Find dialog box.



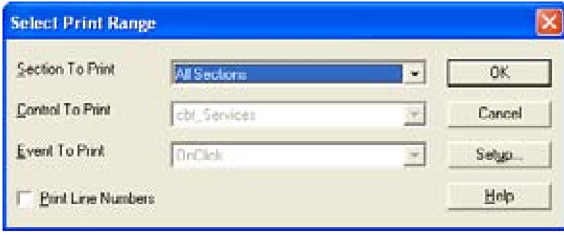
Using the Printing Command

There are several printing options available in the Print command of Dashboard Architect.

➤ To use the Print command:

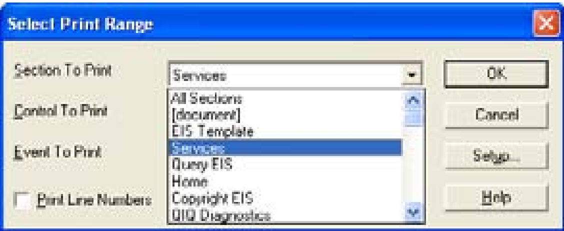
1 Select **File > Print** to launch the **Select Print Range** dialog box.

Figure 8 Selection to Print All JavaScript in the Project



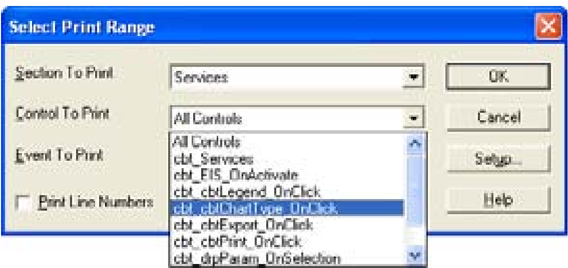
2 Select an option under **Section to Print** by using the drop–down arrow.

Figure 9 Selection to Print a Section



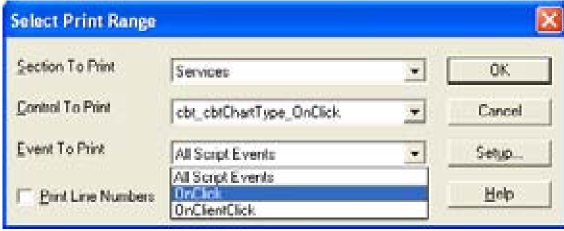
3 **Select an option under Control to Print by using the drop-down arrow.**

Figure 10 Selection to Print a Control



4 **Select an option under Event to Print by using the drop-down arrow.**

Figure 11 Selection to Print an Event



5 **Select Print Line Numbers, if this option is required.**

6 **After the selections are made, click OK.**

The Setup button controls the printer settings configuration.

Using the Match Brace Feature

Use the Match Brace feature to reduce the time spent in identifying sets of incomplete braces. A set of *braces* is one of these symbols:

- Braces {}
- Parentheses ()
- Brackets []

➤ To use the Match Brace feature:

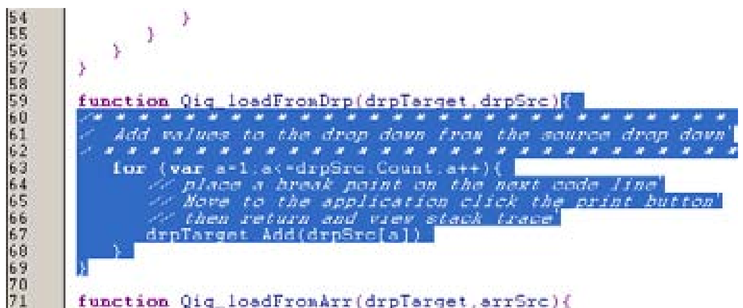
1 **Begin by placing the cursor after an opening brace, press Ctrl+B, or use the command in the Edit menu.**

Dashboard Architect moves forward to look for the opening brace from the current position. If no opening brace is found, the feature looks backward until one is found. When an opening brace is located, the feature searches until it finds a matching closing brace and it highlights the extent of the brace pair.

2 To repeat the operation, press Ctrl+B again.

If no closing brace is found, the operation ends without a highlight.

Figure 12 Matching Braces Found



The Auto–Code Feature

Dashboard Architect includes an Auto–Code feature that makes it faster and more convenient to type in source code. Auto–Code must be enabled by using the Options dialog box in the Tools menu for this feature to work. See "The Options Dialog Box" on page 33, for further information.

Auto–Code automatically adds closing quotation marks, brackets, braces, and parentheses.

When closing quotation marks, brackets, or parentheses are added, they are added immediately after the opening character. The cursor is left between the opening and closing characters waiting for you to type. Typing the closing character overwrites the automatically added character.

When closing braces are added, they are added two lines down with a blank line left between the opening and closing braces. The cursor is left on the blank line, indented one tab.

The Auto–Code feature also enables a special class of macros that are triggered by the spacebar. See "Macros" on page 54. Table 15 is a list and an expansion description of Auto–Code macros, that ship with Dashboard Architect. The caret position at the end of the expansion is shown as **I**.

Tip: To prevent Auto–Code macros from being expanded, press Ctrl+Space rather than space after the macro name. For example, `if` Ctrl+Space.

Table 15 Auto–Code Examples

Auto–Code	Expansion Description	Example
if	*if* expands to an if statement including braces, with the caret in parentheses	`if (I) {` `}`

else	*else* expands to an else statement including braces, with the caret on the blank line between the braces	`else {`
		`    I`
		`}`

while	*while* expands to a while loop including braces, with the caret in parentheses	`while (I) {` `}`
for	*for* expands to a for loop including braces, with the caret in parentheses at the loop index initialization position	`for (I;;) {` `}`
function (or fn)	*function* or *fn* expands to an empty function with the caret placed before the parentheses for the function, where the function name is placed	`function I() {` `}`

try	*try* expands to a try and catch block with the caret on the first line of the try block, indented one tab	`try {`
		`    I`
		`}`
		`catch(e) {`
		`}`

ad	*ad* expands to ActiveDocument and inserts the caret in the very next character position	`ActiveDocumentI`
cn	*cn* expands to a Console.Writeln statement with the caret between the quotation marks of the message to be printed	`Console.Writeln("I")`

Macros

Macros are another means of entering source code quickly. Some features of Auto–Code are implemented by using special macros.

You use macros to assign short character strings to commonly used snippets of code, so you can type the short character string and expand it into the code snippet.

Macros can be simple, single line text substitutions such as *ad* and *cn,* or they can expand into complex multi–line code snippets including replaceable parameters with default values.

Typical macros are invoked by typing the macro name and pressing Ctrl+P. The macro is expanded, only if the caret is immediately after the macro name, and the macro name is at the beginning of the line, or it contains at least one space before it.

About Defining a Macro

Macros are defined in a text file called *macros.txt* that is located in the config directory with this installation.

Each macro definition requires two lines. The first line contains the macro name. The second line contains the definition of the macro including parameters and caret control codes.

The macro name is the short character string typed into the editing window to invoke the macro. Each macro must have a unique name. A special class of macros used by the Auto–Code feature must have names that begin with *internal.*

Caution! The `macros.txt` file must end with a blank line.

Simple Macros

This topic defines simple macros that take no parameters, beginning with single–line macros and introducing caret control codes for macros whose output spans multiple lines.

The *ad* Macro

The *ad* macro is a very simple macro that expands from *ad* to *ActiveDocument* and leaves the caret at the end of the word.

The definition in the `macros.txt` file is:

```
ad
ActiveDocument
```

When you enter *ad* and press Ctrl+P, the characters *ad* are replaced by *ActiveDocument.* The caret is left where it is when the macro expansion is finished, which is at the end of the text entered by the macro expansion.

The *cn* Macro

The *cn* macro expands *cn* to *Console.Writeln("")* and demonstrates simple caret control codes by leaving the caret inside the quotation marks.

The definition in the `macros.txt` file is:

```
cn
Console.Writeln{(}""{)}{LEFT 2}
```

There are three places where characters are surrounded by braces { }.

Each parenthesis must be enclosed in braces, because each has a special meaning and the macro expansion feature must be warned that these characters are to be used as part of the expansion rather than take their special meaning.

Tip: When a character is enclosed in braces, to switch off its special meaning, is called *quoting* the character.

More interesting is the `{LEFT 2}` at the end of the macro definition. This expression means move the caret left two characters, as if you had typed the left–arrow twice. These caret control codes may be placed anywhere within the macro definition and are executed as they are found.

A complete list of caret control codes is provided later.

Multiple–Line Macros

This topic demonstrates how caret control codes can be used to define macros whose output spans multiple lines. The *for* macro is an example.

The *for* macro expands into a simple *for* loop including braces.

The definition in the `macros.txt` file is:

```
for
for {(};;{)} {{}{ENTER 2}{}}{UP 2}{END}{LEFT 5}
```

As in the *cn* macro, the parentheses are *quoted,* and so are the braces.

The control code `{LEFT 5}` at the end of the macro provides the same effect as in the *cn* macro, but other control codes are introduced.

`{ENTER 2}` means pressing the enter key twice, leaving a blank line between the two braces.

`{UP 2}` is very similar to `{LEFT}`, and acts as if the up arrow key was pressed twice.

`{END}` acts as if the end key was pressed, and takes the caret to the end of its current line.

Therefore, the overall effect of the previous macro can be seen by reading it from left to right, imagining the special keys being pressed where the caret control codes are found.

The macro expands to:

```
for (;;) {

}
```

with the caret between the opening parenthesis and the first semicolon; that is, where the loop index variable is initialized.

Macro Parameters

Macro definitions can include parameters. The text given along with the macro name is used in the macro expansion enabling the same macro to expand differently each time it is invoked.

This topic shows how to define macros that take parameters, and how to invoke macros that require parameters.

A *for* Macro with a Loop Index Parameter

Most *for* statements start with "for (var *x* = 0;…)," where *x* is some variable identifier that is used within the *for* loop. If *for* loops are nested, this variable identifier must be different for each loop inside another loop.

The *for* macro presented previously can be improved to provide the loop index variable initialization, and to enable you to use a different variable identifier in nested loops:

```
for
for {(}var ${1} = 0;;{)} {{}{ENTER 2}{}}{UP 2}{END}{LEFT 4}
```

In the preceding macro definition, the loop index variable identifier is given as `${1}`, which stands for the value of the first parameter supplied to the macro.

Macros can take up to nine parameters called *`${1}`* to *`${9}`*. Macro parameters can be inserted anywhere in a macro definition except inside other special codes enclosed in braces, such as caret control codes. Each macro parameter can be used as many times as necessary. For example, the macro definition can be modified to add the loop index variable increment code automatically by placing `${1}{+}{+}` after the second semicolon:

```
for
for {(}var ${1} = 0;; ${1}{+}{+}{)} {{}{ENTER 2}{}} {UP 2}{END}{LEFT 8}
```

The plus signs are *quoted* by using braces, because they have a special meaning within the macro definitions. Also the macro definitions must be on a single line.

Default Values *for* Parameters

The previous macro requires you to supply the value for the parameter every time the macro is invoked. It is necessary only to change the value of the parameter when *for* loops are nested. The macro definition can be improved further by giving the parameter a default value so it need not be specified unless the default value cannot be used (such as in nested *for* loops).

A macro parameter can be given a default value by specifying the value the first time the macro parameter is shown in a macro definition:

```
for
for {(}var ${1=a} = 0;; ${1}{+}{+}{)} {{}{ENTER 2}{}} {UP 2}{END}{LEFT 8}
```

The syntax `${1=a}` assigns *`a`* as a default value to the parameter `${1}`. If the macro is invoked with no value for `${1}`, it expands to

```
for (var a = 0;; a++) {

}
```

Invoking Macros

In general, macros are invoked by entering their name as a stand–alone word (that is, at the beginning of a line or with at least one space in front of the macro name), and pressing Ctrl+P.

Invoking Macros Without Giving Parameter Values

If a macro takes no parameters or all the parameters are given default values in the macro definition, a macro can be invoked as described in the previous paragraph.

The macro name is removed and replaced with the expanded macro definition.

Invoking Macros Giving Parameter Values

If a macro contains parameters that contain no default value, or the default value is not sufficient, you can give the value of each parameter by appending it to the macro name separated by colons:

`for:b`—sets the value of parameter 1 to *b*

`for:in_startVal`—sets the value of parameter 1 to *in_startVal*

When macros contain multiple parameters and some of those parameters contain default values, the parameters with default values need not be specified. For example, a macro *test* that takes three parameters, where the first and third parameter contain default values assigned in the definition, can be invoked in these ways:

`test:a:b:c`—assigns the value *a* to parameter 1, *b* to parameter 2, and *c* to parameter 3

`test:a:b`—assigns the value *a* to parameter 1, *b* to parameter 2, and parameter 3 takes its default value

`test::b:c`—parameter 1 takes its default value, *b* is assigned to parameter 2, and *c* to parameter 3

`test::b`—parameters 1 and 3 take their default values, and parameter 2 is assigned a value of *b*.

The parameters with default values that are followed by parameters *without* default values must still show a colon as a placeholder. Parameters at the end of the list, which contain default values, may be left off entirely.

Macro Control Codes

Macro control codes enable the use of non-printing keys, such as Enter or the arrow keys, in macro definitions.

The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses ( ) have special meanings in macro definitions. To specify one of these characters, enclose it with braces ({}). For example, to specify the plus sign, use {+}. Brackets ([ ]) have no special meaning but must also be *quoted* because of the routines that are used to implement macros.

To specify brace characters, use {{} and {}}.

To specify characters that are not displayed when you press a key, such as Enter or Tab, and keys that represent actions rather than characters, use the codes shown in Table 16.

Table 16 Macro Control Codes

Key	Code
BACKSPACE	`{BACKSPACE}, {BS}, or {BKSP}`
DEL or DELETE	`{DELETE} or {DEL}`
DOWN ARROW	`{DOWN}`
END	`{END}`
ENTER	`{ENTER} or ~`
HOME	`{HOME}`
INS or INSERT	`{INSERT} or {INS}`
LEFT ARROW	`{LEFT}`
RIGHT ARROW	`{RIGHT}`
TAB	`{TAB}`
UP ARROW	`{UP}`

To specify keys combined with a combination of the Shift, Ctrl, and Alt keys, precede the key code with one or more of the codes in Table 17.

Table 17 Combination Key Codes

Key	Code
SHIFT	+
CTRL	^
ALT	%

To specify a combination of Shift, Ctrl, and Alt to be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify a user should hold down Shift while *E* and *C* are pressed, use `"+(EC)"`. To specify a user should hold down Shift while *E* is pressed, followed by *C* without Shift, use `"+EC"`.

To specify repeating keys, use the form `{key number}`. You must insert a space between *key* and *number*. For example, `{LEFT 42}` means press the Left Arrow key 42 times; `{h 10}` means press *h* 10 times.

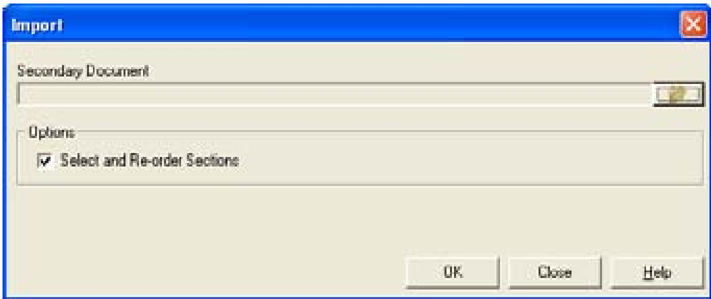
Importing Sections from Other Interactive Reporting Documents

The Dashboard Architect Import Utility enables other Interactive Reporting documents to be imported into the current document. It eliminates the recreation of identical information in different documents. One document can be imported into another document.

➤ To use the Dashboard Architect Import Utility:

1 Select **Project > Import.**

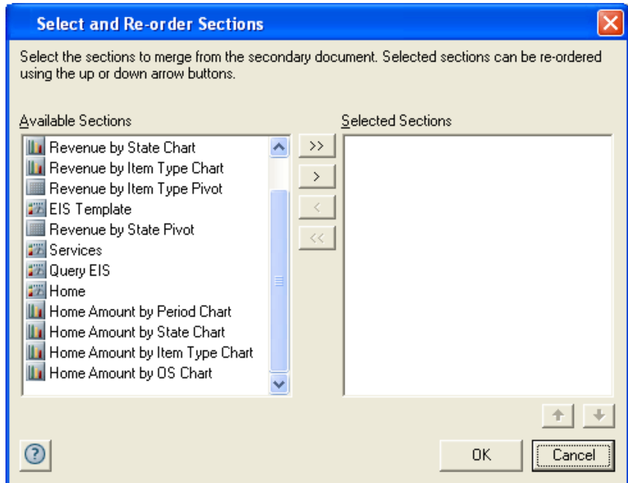
The Import window is displayed.



2 Click , next to the **Secondary Document** text box, and navigate to the location of the Interactive Reporting document from which one or more sections are to be imported.

3 Click **OK.**

If Select and Re-order Sections is selected, the Select and Re-order Sections Window is displayed.



4 From the **Available Sections** list box, select sections, and click [>] or [>>] to move them to the **Selected Sections** list box.

Sections can be selected individually or in groups, and moved from Available Sections to Selected Sections.

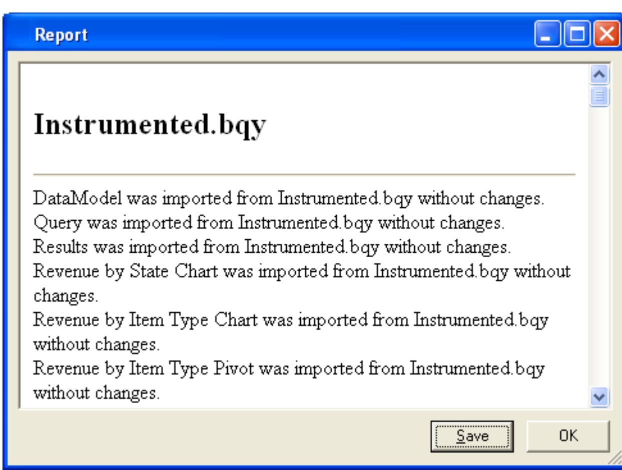
5 Select and reorder the sections to be imported into the original Interactive Reporting document, by clicking [↑] or [↓].

6 After the selections are finalized, click **OK.**

The sections are now imported into the open project in Dashboard Architect.

The following report shows which sections were imported, and if one or more of the sections from the secondary Interactive Reporting document required renaming.

Figure 13 Import Report



7 **Optional:** Save the report for future reference.

The imported sections are instrumented and become part of the project.

6

The Testing and Debugging Processes

Testing and debugging activities are closely associated with editing. Testing is often accompanied by correcting problems, and involves editing. Editing is not complete until the code being edited is tested and performs as required. See “Editing” on page 41.

In This Chapter

About Testing. 64
About Debugging. 66
Breakpoints. 66

About Testing

Testing is accomplished by placing Dashboard Architect in the background and working with the `instrumented.bqy`. The user clicks the buttons, processes queries, selects items from lists, and selects option and check boxes. Each of these clicks causes an event to be fired. The event handler that Interactive Reporting Studio calls, in turn calls the Debug Server and asks for the JavaScript. This is returned to it as a string, which the event handler executes, by using the JavaScript `eval()` function, and so `instrumented.bqy` behaves just like an ordinary Interactive Reporting document.

Testing Rule

A simple rule must be remembered when working with Dashboard Architect.

JavaScript code is sent to an event handler when, and only when, that handler calls the Debug Server and asks for it.

Procedural and Declarative JavaScript

It is important to be aware of JavaScript that is procedural in nature, and JavaScript that is declarative in nature. Declarative code acts to set up or declare shared or public resources such as global variables, properties, and functions that can be called later by using their public *handles* from procedural code. Procedural code acts immediately on shared or public, or local or private resources.

The rule operates equally to the sets of code, but declarative code is usually associated with `OnStartUp()` events or events in hidden objects that the user does not typically see.

Changes to procedural code require that the event they are declared in be fired before the change is observed. Changes to procedural code take place immediately in real time. The code associated with the `OnClick()` event of a button can be changed. In `instrumented.bqy`, click the button to run with the new code and observe the behavior.

Changes to declarative code also require the event they are declared in be fired before the change is observed. Causing an event that uses the results of a declaration is not the same as causing the event that creates the declaration. Consequently, changes to declarations require re-declaration before calls on the re-declarations can show the changes made in Dashboard Architect.

Consider the following situation:

MyEIS.Button_X	
Instrumented.bqy	Dashboard Architect
eval(<QIQDebug_call>)	ListBox1.RemoveAll() ListBox1.Add("select a Country") ListBox1.Add("Australia") ListBox1.Add("Britain") ListBox1.Add("France") ListBox1.Add("Germany") ListBox1.Add("United States")

When the user clicks `Button_X` on the MyEIS section, the Debug Server is called and returns the JavaScript held by Dashboard Architect for the `OnClick()` event of `Button_X`.

As soon as the code is changed, the button can be clicked again. The Debug Server is called and finds the new code. It returns the code to `instrumented.bqy`. The code is passed to the JavaScript `eval()` function. The new behavior is immediately observed.

This is illustrated in the dashboard section called *Query EIS* in the `Arch_tut.bqy` that is provided as a sample with Dashboard Architect.

Consider the following example:

OnStartUp	
Instrumented.bqy	Dashboard Architect
eval(<QIQDebug_call>)	var eis=ActiveDocument.Sections["MyEIS"] eis.Shapes["Button_X"].OnClick()
MyEIS.Button_X	
Instrumented.bqy	Dashboard Architect

eval(<QIQDebug_call>)	function fill_ListBox(lbx){
	lbx.RemoveAll()
	lbx.Add("select a Country")
	lbx.Add("Australia")
	lbx.Add("Britain")
	lbx.Add("France")
	lbx.Add("Germany")
	lbx.Add("United States")
	}
	ActiveDocument.fill_ListBox=fill_ListBox

MyEIS.Button_Y	
Instrumented.bqy	Dashboard Architect
eval(<QIQDebug_call>)	ActiveDocument.fill_ListBox(ListBox1)

1. The document calls the Debug Server for its code when it starts up. In this case, it causes the `OnClick()` event of `Button_X`. This creates a function called *fill_ListBox* available as a property of the `ActiveDocument`.
2. When a user clicks **Button_Y** on the MyEIS section, the `fill_ListBox` function is called with the same effect as the example discussed earlier.
3. Now make a change to the JavaScript in the `OnClick()` event of the `Button_X`.
4. Press **Button_Y.**
5. Unlike the previous example, this time the code is executed as if those changes were never made, despite the fact that changes were just made to the code under `Button_X`. This is because unless the `OnClick()` event of `Button_X` (or the `OnStartUp()` event of the document) is caused, the changes made to the code of `Button_X` are not sent into `instrumented.bqy.` Therefore `Button_X` must be clicked first and *then* `Button_Y` for the new code to take effect.

The Fire Current Event button is provided for that very purpose so the event associated with the JavaScript currently in view can be executed.

It is vital to remember this subtle but very important distinction between simple procedural code (statements that perform operations directly as in the first example) and declarative code (code that creates functions to be called and used later) as in the second example.

It is *highly recommended* to have the Console window open when is clicked so errors are seen and reported. If an error occurs, the cog associated with the JavaScript event handler that has failed, changes to red () in the Code Requests pane of the output window.

About Debugging

Debugging is closely associated with testing and editing. Debugging involves determining why a behavior is happening and what can be done to change it. Traditional approaches to debugging include one of the following to determine what is occurring:

1. `Console.Writeln()` statements are interspersed with regular code, or
2. Debug functions are interspersed with regular code.

In the case of (1), the code must be removed after the problem is found, in the case of (2), the code can stay, because debug functions can be turned on and off providing permanent instrumentation. While option (2) is distinctly better than (1), both suffer from the disadvantage that manual instrumentation is not flexible. It is of use only when identifying a very specific situation that was anticipated by the programmer when the code was originally being written.

Breakpoints

With Dashboard Architect, the convenience of being able to set a breakpoint makes debugging a great deal easier than the two traditional approaches.

Breakpoints are required when one or more areas of code come under suspicion. The cursor is placed on a strategic line and a breakpoint is inserted by clicking or F9.

Internally, a breakpoint is implemented as a set of additional dynamic instrumentation instructions alongside the regular JavaScript. As a breakpoint is extra JavaScript, it behaves just like other code changes. If it is placed inside declarative code such as a function, you must click so the breakpoint can become part of the declaration of the function.

When all breakpoints are set, move to `instrumented.bqy`, and activate the event (typically by clicking it, if it is a button). The code is executed and at some point it encounters the breakpoint. The `instrumented.bqy` calls the Debug Server and asks it to suspend execution and highlight the line associated with the breakpoint.

The Debug Server keeps Interactive Reporting Studio suspended and gives control to the user interface of Dashboard Architect, so the execution state of the application can be examined. At this point, the evaluate pane in the output window is activated, and valid JavaScript expressions can be entered for evaluation.

To see the value of a local variable, enter the name of the variable, and click . To see the value of a Interactive Reporting Studio property, type that into the evaluate pane. As a shortcut, highlight text in the editing window, right–click, and select if the highlighted text is appropriate to evaluate, or copy and paste into the evaluate pane and modify the expression as required. To clear the evaluate pane, press .

If incorrect syntax is entered, the JavaScript engine generates a syntax error and aborts the current thread of execution. Return to `instrumented.bqy` and recommence the test by clicking the control so as to reactivate the breakpoint.

There are limitations as to where breakpoints can be placed in code. Breakpoints are implemented, as JavaScript is inserted, so they must be syntactically valid. Dashboard Architect does not have direct access to the JavaScript engine and it is therefore not able to guarantee the correctness of the breakpoints. The following guidelines help with writing code that is straightforward to maintain, read, and set breakpoints upon.

Table 18 Breakpoints Working With JavaScript

JavaScript	Comment
Line 01 function f(param){	Break happens when the function is being parsed not when it is called
Line 02 var x	
Line 03 if (condition){	
Line 04 statement_1	
Line 05 }else{	
Line 06 statement_2	

Table 18 Breakpoints Working With JavaScript *(Continued)*

Line 07 }	
Line 08 switch (x){	
Line 09 case 1:	Breakpoint is not OK as nothing can come before *case*
Line 10 statement_3	
Line 11 break	
Line 12 case 2:	Breakpoint is not OK as nothing can come before *case*
Line 13 statement_4	
Line 14 break	
Line 15 default:	Breakpoint is not OK as nothing can come before *default*
Line 16 }	
Line 17 // comment	
Line 18 }	

As can be seen in the preceding table, there are a few, very distinct and recognizable places where a breakpoint *cannot* be placed. If the code is not written in this manner, the options for breakpoints are more restricted. Examples are provided in Table 19.

Table 19 Areas Where Breakpoints Do Not Work With JavaScript

JavaScript	Comment
Line 01 function f(param){	
Line 02 var x	
Line 03 if (condition)	
Line 04 {statement_1}	Breakpoint is not OK
Line 05 else	Breakpoint is not OK
Line 06 {statement_2}	Breakpoint is not OK
Line 07 switch (x){	
Line 08 case 1: statement_3;break	Breakpoint is not OK
Line 09 case 2: statement_3;break	Breakpoint is not OK
Line 10 default:	Breakpoint is not OK
Line 11 }	
Line 12 // comment	
Line 13 }	

7 Adding and Removing Objects

This chapter discusses the Dashboard Architect feature that facilitates the adding and removing of objects.

Dashboard Architect interacts with Interactive Reporting Studio through its COM programming interface. This interface defines the operations that Interactive Reporting Studio can perform on behalf of its clients.

The creation of objects (such as buttons, drop–down list boxes, list boxes, check boxes, and option buttons) and the addition of JavaScript into their event handlers are operations that are not currently available through programmatic means. You must perform these operations inside Interactive Reporting Studio.

Operations that can be performed in only one environment are problematic, because there are two sets of structures. Interactive Reporting Studio holds one set of structures in `instrumented.bqy`, and Dashboard Architect holds the other. Unless these two structures are in synch, correct operation cannot be provided.

Operations that must be performed through the graphical user interface of Interactive Reporting Studio must also be incorporated into the development environment of Dashboard Architect.

In This Chapter

Resynchronizing. 70
Adding Controls. 70
About Deleting Controls 72
About Renaming Controls 72
Adding and Duplicating Sections 72
About Renaming Sections 73
About Deleting Sections. 74

Resynchronizing

The simplest way for Dashboard Architect to incorporate objects is through the Resynchronize operation. This operation can be initiated selecting by Project > Resynchronize.

Resynchronize works in a similar way to the create project operation, see "Creating Projects" on page 36. It closes `instrumented.bqy`, and performs the following:

1. `Instrumented.bqy` is disassembled.
2. `Instrumented.bqy` is analyzed.
3. Objects that exist in `instrumented.bqy` but do not exist in Dashboard Architect are added to it. These are new objects created by using the Interactive Reporting Studio user interface. The JS files are extended to include references to these objects. JavaScript added by the user is transferred to the JS files and is replaced by instrumentation code.
4. Objects that exist in Dashboard Architect but do not exist in `instrumented.bqy` are removed from it. Dashboard Architect removes all references to the deleted objects and they cease to exist.
5. `Instrumented.bqy` is reassembled.
6. `Instrumented.bqy` is reopened in Interactive Reporting Studio.
   Resynchronization may take a few minutes to complete depending on the size of the project. It is the simplest operation from the viewpoint of the user, but it may not be the most convenient as `instrumented.bqy` must be closed, analyzed, and reopened.

If an object is renamed in Interactive Reporting Studio, the old object is deleted in Dashboard Architect (as it no longer exists in Interactive Reporting Studio), and a blank object is created. JavaScript associated with the old object is lost.

Adding Controls

Add a control by duplicating or creating a control.

Duplicating Controls

The quickest way to add a control is to duplicate one that exists and is instrumented.

➤ To duplicate a control:

1 **In Interactive Reporting Studio, and press Ctrl+D to enter Design mode.**

2 **Select a control and duplicate it.**

3 **Double-click the control.**

The Properties dialog box is displayed.

4. **Enter an object name, and click OK.**
5. **Press Ctrl+D to exit Design mode.**
6. **Click the control.**

   Dashboard Architect gets a call from the control for its code. Dashboard Architect fails to find a reference to this control so it creates an event handler on the fly and immediately synchronizes itself with the state of `instrumented.bqy`.

Creating Controls

Sometimes there is no control that can be duplicated, and a control without instrumented code must be created.

➤ To create a control:

1. **Select Project > Add Control.**

   The Add Control dialog box is displayed.

2. **Click Copy to Clipboard to copy the instrumentation code to the clipboard.**
3. **Click Cancel.**
4. **In Interactive Reporting Studio, and press Ctrl+D to enter Design mode.**
5. **Create the control and paste the instrumentation code into each of its event handlers.**
6. **Press Ctrl+D to exit Design mode.**
7. **Click the control.**

   Dashboard Architect gets a call from the control for its code. Dashboard Architect fails to find a reference to this control so it creates one, and immediately synchronizes itself with the state of `instrumented.bqy`.

About Deleting Controls

A control can be deleted in Interactive Reporting Studio. After the control is deleted, you must resynchronize immediately, or wait until the next time an Interactive Reporting document is made. Dashboard Architect performs a Resynchronize operation every time it makes an Interactive Reporting document.

About Renaming Controls

Renaming a control is a problematic operation. Dashboard Architect perceives that a control is deleted and a new control is added. JavaScript held for the control under its old name, must be manually cut and pasted into the event handler for the control with the new name.

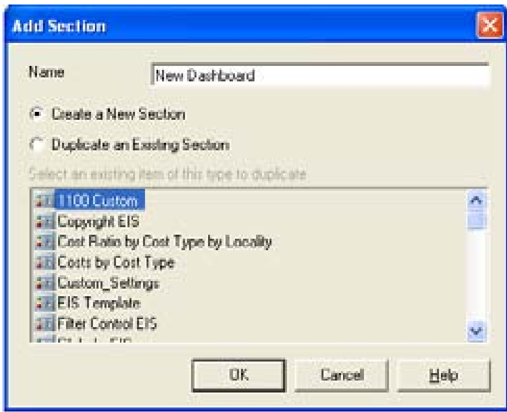
Adding and Duplicating Sections

This command relates only to dashboard sections. Other sections do not contain JavaScript and so are outside the domain of Dashboard Architect.

➤ To add a section:

1 **Select Project > Add Section.**

The Add Section dialog box is displayed.



To duplicate a section use the same dialog box, and select Duplicate an Existing Section. Or select a section node in the navigation panel, and right–click to open a popup menu and select the Duplicate command.

2 **In the Name text box, enter the name of the new or duplicated section, and click OK.**

The section is now added or duplicated.

Dashboard Architect provides for the creation of sections in the following way:

1. The new section is added programmatically (or copied from a section if duplicating).
2. `Instrumented.bqy` is closed.
3. `Instrumented.bqy` is disassembled.
4. Instrumented JavaScript is added to its `OnActivate()` and `OnDeactivate()` events.
5. `Instrumented.bqy` is reassembled.
6. `Instrumented.bqy` is reopened in Interactive Reporting Studio.

A matching section with all basic event handlers is added to Dashboard Architect. If the section is duplicated, a copy of all original event handlers is made and inserted into the node in the navigation panel. The operation is relatively fast.

About Renaming Sections

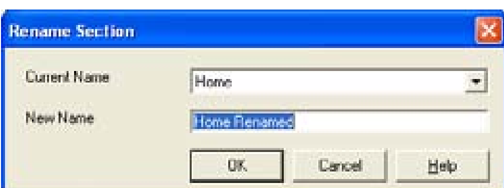
This command relates only to dashboard sections. Other sections do not contain JavaScript and so are outside the domain of Dashboard Architect.

➤ To rename a section:

1 **Select Project > Rename Section.**

Alternatively, a section node can be selected in the navigation panel.

The Rename Section dialog box is displayed.



2 **From the Current Name drop-down list, select the section to be renamed.**

3 **Enter a name in the New Name text box.**

4 **Click OK.**

The section is now renamed.

About Deleting Sections

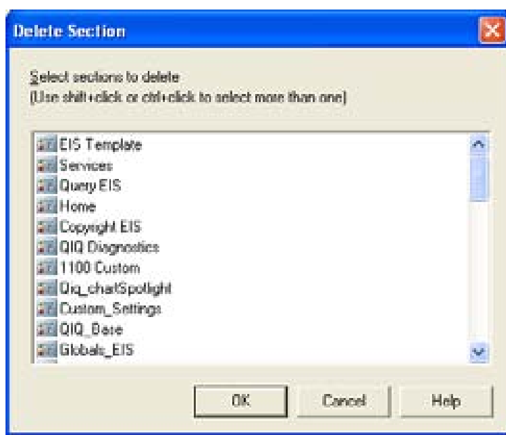
This command relates only to dashboard sections. Other sections do not contain JavaScript and so are outside the domain of Dashboard Architect.

➤ To delete a section:

1 **Select Project > Delete Section.**

Alternatively, a section node can be selected in the navigation panel.

The Delete Section dialog box is displayed.



2 **In the Delete Section dialog box, highlight the sections to be deleted.**

3 **Click OK.**

The sections are deleted.

8 Documentation

It is recognized that good documentation enhances reuse of code and makes code easier to maintain. This chapter discusses in detail, the features provided by Dashboard Architect to document reliable code.

In This Chapter

Documentation of Code 76
Generating Documentation 85

Documentation of Code

Dashboard Architect provides features that enable you to document code more easily and consistently, and to extract documentation from the code so it can be published in other forms.

Dashboard Architect provides features for documenting variables, functions, classes, and components. A special format of comment is used to recognize formal documentation, and a number of tags are supplied to define information such as the type of a variable (@type), parameter, or function, the formal parameters of a function (@param), and the visibility of a variable (@scope).

Documentation Comments

Comments to be published in the documentation must be entered in a special format. They are block comments that start with /** and end with */.

Documentation comments follow the general form of one or more paragraphs of descriptive text, the first of which is used when a summary is required, followed by special tags required.

Paragraphs within the descriptive text are marked by a blank line.

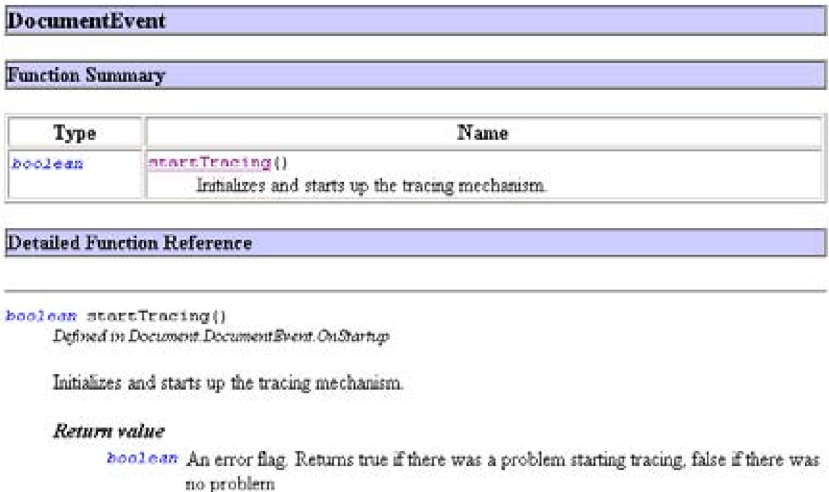
Documentation tags are placed after the general description and descriptive text after a tag must be in a single paragraph (that is, no blank lines).

A documentation comment to describe the function *multiply,* which takes two parameters, multiplies them together, and returns the result looks like this example:

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 */
function multiply(in_intA, in_intB) {...
```

This describes the function in two paragraphs, the first of which is a summary of what the function does. It also describes the names and suggested types of the parameters, and how the return value is to be interpreted.

When the documentation is generated, the documentation comment produces output similar to this example.



Documentation of Variables

To document a variable is a simple matter of describing what the variable is used for and how, and giving a type so readers know what type of data is acceptable for the variable.

The only documentation tags that are to be used in a variable comment are @type to specify the typical variable, and @scope to define the visibility of a variable. These tags are explained further in "Namespace Scope of Entities and the @scope Tag" on page 80. The tags are optional, but it is recommended to always use the @type tag. If the @scope tag is not used the documentation system assumes the variable is not visible outside the current event handler.

Variable documentation comments must be placed immediately before the variable declaration or immediately before the variable is used in an assignment statement, in which case the variable on the left side of the equals sign is assumed to be the variable being documented and the variable name may only have white space before it on the line.

A simple documentation comment for a variable.

```
/**
 * Points to the BQY Section that contains this shape.
 * objSection can be used to access other shapes within the
 * dashboard section.
 *
 * @type Section
 */
var objCurrentSection = this.Parent
```

Variable documentation is most useful when variables are exposed at a section level or as a property of a class, so they are used outside the event handler in which they are defined.

Documentation of Functions

Documentation of functions enable programmers to determine which functions are available, what their expected parameters are, and what they return.

In addition to the tags for variables mentioned previously, the @param tag can be used to declare the parameters of the function, their suggested type, and to describe them. When used in a function documentation comment, the @type tag declares the typical return type of the function.

A function documentation comment must be placed immediately before the function declaration.

An example of a documentation comment for a function is given in "Documentation Comments" on page 76.

Documentation of Classes

Classes are collections of related functions (member functions) that can share and expose a persistent state by means of properties (variables).

These member functions and properties can be documented by using the techniques described in the previous two topics.

A member function or variable of a class must be given an @scope tag with the name of the class as its value. This binds the function or variable to the class definition.

The function that defines the class is the class constructor. Class constructors are documented like other functions in the class, but in addition to their @scope tag showing they belong to the class, their @type tag shows they return an instance of the class.

The fact that the functions and variables are related to each other by membership to the class makes it necessary to add an extra level of documentation that binds them together and gives an overview of what the class is used for.

The *class* comment may be anywhere in the source code, but it is most useful when placed before the class constructor comment. The *class* comment provides an overview of the class and what it is used for.

A *class* comment is created by using an @class tag, with the class name as the first word after the tag. A *class* comment for a persistent property bag is given here, along with the constructor comment.

```
/**
 * The PropertyBag class is used to store, retrieve, and access
 * a set of persistent properties.
 *
 * Properties are stored in a Shape, which must be a text label
 * or text box.
 *
 * Properties are key/value pairs, where both the key and the
 * value must be Strings.
 *
 * @class PropertyBag
 */
```

```
/**
 * The PropertyBag constructor creates a new PropertyBag
 * instance.
 *
 *
 * If the in_objPersistenceShape parameter is not given the
 * PropertyBag is initialized to be empty.
 *
 * If the in_objPersistenceShape parameter is given, the
 * constructor attempts to load the property values from the
 * given shape.
 *
 * @param in_objPersistenceShape Shape an option parameter that
 * can be used to define the Shape to load the property values
 * from.
 * @type PropertyBag
 * @scope PropertyBag
 */
function PropertyBag(in_objPersistenceShape) {

   /**
    * Returns the value of the property identified by
    * in_strKey.
    *
    * If the PropertyBag does not contain a value for the
    * given key it will return null unless the optional
    * in_strDefaultValue parameter is supplied, in which
    * case that parameter value is returned.
    *
    * @param in_strKey String the key of the property
    * whose value is to be returned
    * @param in_strDefaultValue String an optional default
    * value to be returned if the PropertyBag does not
    * contain the property identified by in_strKey.
    * @type String
    * @scope PropertyBag
    */
   function getProperty(in_strKey, in_strDefaultValue) {
      ...
   }
}
```

In this example the @class tag in the first comment serves to mark it as the class overview comment; the @type and @scope tags in the function comment for the PropertyBag function reference back to the @class value of the class overview comment and mark the function as the class constructor. The comment for the getProperty function contains an @scope tag that links it to the PropertyBag class, but the @type tag shows it returns a String, so the documentation system knows it is a member function and not a constructor.

Note: Classes may expose member functions or properties that are not defined within the body of the class constructor using an assignment statement that references a function or variable declared outside the constructor. In this case, the @scope tag still works to bind the externally declared function or variable to the class.

Documentation of Dashboard Development Services Components

The documentation system of Dashboard Architect can provide extra information about Dashboard Development Services components, and list them in their own section of the documentation.

Each Dashboard Development Services component contains a number of text labels within its sections that contain information about the component and its runtime requirements. These text labels are `txlName`, `txlReqBrio`, `txlReqTemplate`, `txlDescription`, and `VersionInfo`. For more information see “Component Fingerprints” in the *Hyperion System 9 BI+ Dashboard Development Services Components Reference Guide.*

Dashboard Architect reads and formats the values of these text labels for use in the component documentation page. This provides a summary of the component at the top of the page.

Caution! For this feature to work, `instrumented.bqy` must be loaded in Interactive Reporting Studio. If the Interactive Reporting document is not loaded, Dashboard Development Services components are treated like a standard dashboard.

Namespace Scope of Entities and the @scope Tag

The visibility of a function or variable and how to describe it is a vital concept to understand when using the Dashboard Architect documentation system.

In JavaScript, variables (including variables that are pointers to functions) may be made visible in various scopes. In the context of JavaScript within Interactive Reporting Studio, the default scope for a variable is for it to be visible within the current event handler, anywhere after it is declared or first used.

It is common practice to expose a variable to other event handlers by attaching the variable to the parent section of the event handler or to the active document.

The @scope tag enables you to document the visibility of a variable, making it clear in the generated documentation how the variable can be accessed or how the function can be called. Previous examples have illustrated one use of the @scope tag, in associating functions and variables with classes.

For example, if a variable is defined and used in an event handler and not exposed, no other event handler can use it. If a variable is defined and attached to the parent section object, other event handlers can access that variable through the parent section. If a variable is defined and attached to the active document, other event handlers can access that variable by name alone because it is visible at the highest level. The following examples illustrate different levels of visibility and how to use the @scope tag to document them.

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 */
function multiply(in_intA, in_intB) {...
}
```

In the previous example, the function is not visible anywhere other than the current event handler. This means code in other event handlers cannot generally call this function. For this reason, no @scope tag is given and the documentation system considers this to be a *local* function and it is not included in the documentation unless you ask for control level documentation to be produced.

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 * @scope Section
 */
function multiply(in_intA, in_intB) {...}
this.Parent.multiply = multiply
```

The previous example shows that the function is exposed through the parent section of the correct shape, by the assignment after the closing brace of the function. This means code in other event handlers can easily call this function if they contain a reference to the section. This is made explicit in the documentation comment by giving an @scope tag with a scope of *Section.*

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 * @scope Document
 */
function multiply(in_intA, in_intB) {...}
ActiveDocument.multiply = multiply
```

Finally, the preceding example shows a function that was made available to an event handler in the current document simply by naming the function with no namespace qualifiers. This is made explicit in the documentation comment by giving an @scope tag with a scope of *Document.*

The @scope tag is necessary because JavaScript is type–less, and because the exposure of a function or variable through an assignment need not be done immediately after the function is declared. There is no reliable method for the Dashboard Architect documentation system to determine the visibility of a function or variable from the source code alone.

Documentation Grouping by Using the @scope Tag

The @scope tag may be given a second parameter, which is used to define different *groups* for parts of the documentation. For example, this feature can be used to differentiate the public and private parts of an API of an object—when the documentation is generated for use by in–house developers the private API calls can be included in the documentation. When external developers generate documentation they can exclude the private API calls to ensure they do not call private methods by mistake.

This feature imparts no impact on the code visibility—the notions of public and private presented here are a logical convenience. Other groupings are possible, simply by using different parameters.

Examples of the use of this parameter are:

```
/**
 * A private API function that should not be called from
 * outside the class it is defined in.
 *
 * @type void
 * @scope SomeClass private
 */
function privateAPICall() {...
}

/**
 * This function does something useful and can be called
 * by any code with a reference to an object of this
 * class.
 *
 * @type Integer
 * @scope SomeClass public
 */
function publicAPICall() {...
}
```

See "Generating Documentation" on page 85.

Documentation Comment Tag Reference

This topic shows what information can be provided to each documentation tag.

@class classname [documentation_group]

The @class tag must always be given a class name. This is the name that is used to bind the constructor, member functions, and properties back to the class.

The optional *documentation_group* is a single word used to group sets of classes, functions, and variables so they can be included or excluded from the generated documentation as a group. See "Generating Documentation" on page 85.

@param param_name type_name [description]

The @param tag must always be given the name of the parameter, and its expected type (such as number, section, object, shape, and so on). As JavaScript is type–less the parameter type can only ever be a suggestion but it enables callers to see what types of values are expected by the function.

The optional *description* may be given on multiple lines, but may not contain blank lines.

@type type_name [description]

The @type tag must be given the type of the variable or the return type of the function being documented (such as number, section, object, shape).

If documenting a function the optional *description* is given next to the return type of the function, so the caller can see how to interpret the return value.

@scope namespace [group]

The @scope tag may be given a namespace of *Document*, *Section*, *Control*, or the name of a class defined with an @class tag.

The optional *group* is a single word used to group sets of classes, functions, and variables so they can be included or excluded from the generated documentation as a group. See "Generating Documentation" on page 85.

Dashboard Development Services Component—Specific Features

The documentation feature can recognize Dashboard Development Services components and include additional information regarding those components.

Dashboard Development Services Component–Specific information is held in text labels on the component code section. All of this additional information is optional. The information in the table may be specified for a Dashboard Development Services component.

Table 20 Dashboard Development Services Component–Specific Information In Text Labels

txlName	A display name for the component. For example, dynamic headings rather than `QIQ_fmtHeading`.
txlDescription	One or more paragraphs of descriptive text about the component.
txlReqBrio	The release of Interactive Reporting Studio required to use the component. The format is: "VersionMajor=*a*\nVersionMinor=*b*\nVersionRelease=*c*\nVersionPatch=*d*" where *a, b, c,* and *d* are numbers, and \n represents a new line sequence. For example, VersionMajor=8 VersionMinor=3 VersionRelease=0 VersionPatch=647
txlReqTemplate	The release of the Dashboard Studio template required to use the component. The format is: "VersionMajor=*a*\nVersionMinor=*b*\nVersionRelease=*c*" where *a, b,* and *c* are numbers, and \n represents a new line sequence. For example, VersionMajor=8 VersionMinor=3 VersionRelease=45

Generating Documentation

To create the HTML documentation, select Project > Generate Documentation.

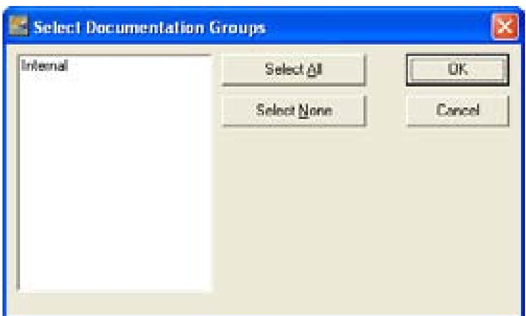
This topic refers to documentation generation features:

- Inclusion and Exclusion of Documentation Groups
- Inclusion and Exclusion of Unscoped Documentation
- Dashboard Development Services Component—Specific Features in HTML Documentation

Inclusion and Exclusion of Documentation Groups

If documentation groups are defined by using the second parameter of the @scope tag, the Select Documentation Groups dialog box is displayed. The dialog box enables you to select which documentation groups to include in the HTML output.

Figure 14 Select Documentation Groups Dialog Box



Selecting one or more of the documentation groups, and clicking OK includes the selected documentation groups in the HTML documentation. If no documentation groups are selected, or if Cancel is clicked, only documentation without a documentation group specifier (that is, an @scope tag with 1 parameter) is included in the HTML documentation.

Inclusion and Exclusion of Unscoped Documentation

Documentation blocks that do not include an @scope tag are not included in the generated HTML documentation unless the Project > Include Control Documentation menu command is selected. See "The Project Menu" on page 30.

If this command is selected, the unscoped documentation is included at the end of the section of the documentation, grouped by shape.

Dashboard Development Services Component–Specific Features in HTML Documentation

If you cannot find `instrumented.bqy` in the list of open Interactive Reporting documents, a warning dialog box is displayed.

The HTML documentation is still generated, but the documentation generator cannot distinguish Dashboard Development Services components from other dashboard sections, so Dashboard Development Services Component–Specific information is not included in the generated documentation.

9 Using the Dashboard Development Services Update Utility

This chapter discusses the Dashboard Development Services Update Utility. The utility provides a convenient method of updating JavaScript in Interactive Reporting document files, provided that the JavaScript is designed by using dashboard sections to create a layered architecture.

In This Chapter

About Dashboard Development Services Update Utility 88

Update Workflow 90

Creating New Sections Files 91

Updating New Sections Files 91

Configuring INI Files 92

Adding and Removing Sections 93

Updating Documents 94

About Dashboard Development Services Update Utility

The Dashboard Development Services Update Utility moves you closer towards a full solution for propagating changes. The principle of the Dashboard Development Services Update Utility only works for Interactive Reporting document files that are developed in a layered manner.

Layer 1 is the data and views (query, chart, pivot, and report sections). This layer is not touched by the Dashboard Development Services Update Utility.

Layer 2 is the dashboard sections that you interacts with – it has charts, pivots, and user interface controls (list boxes, buttons, drop–downs list boxes, and so on). The controls contain only very simple, one line calls to Library routines. This layer is not touched by the Dashboard Development Services Update Utility.

Layer 3 is the dashboard sections that contain reusable JavaScript functions, classes, and components. This layer is replaced by the Dashboard Development Services Update Utility.

The Dashboard Development Services Update Utility uses the `newsections.dat` file to *push* new Layer 3 sections into *old* dashboards, turning them into *new* dashboards.

The `newsections.dat` file is an Interactive Reporting document that contains all the latest Layer 3 dashboard sections.

The file to be updated is examined and compared against the `newsections.dat` file. Common sections are noted and deleted from the file to be updated, and their counterparts from the `newsections.dat` file are added. As of release 8.3.1, the release information found in a dashboard section is used to ensure that *earlier* sections do not replace *later* sections. Therefore, a section is only replaced if a more recent release is found in the `newsections.dat` file.

For example, in the following schematic, `Globals_EIS` and `QIQ_Limits` are replaced as part of the Update operation, but `Qiq_MetaData` is not.

Table 21 Update An Interactive Reporting Document

The Interactive Reporting Document to Update	New Sections.dat
Globals_EIS [8.2.27]	Globals_EIS [8.3.07]
Home	Qiq_Limits [8.3.07]
Qiq_Limits [8.2.27]	Qiq_MetaData [8.3.07]
Qiq_MetaData [9.0.0]	Qiq_CopyScale [8.3.07]
	Qiq_Cal [8.3.07]
	...
	...

The `Upgrade.ini` file provides some fine tuning for the operation of the process. This refinement includes:

1. **Option Preservation**
   To make reusable code work in a wide variety of situations, it needs to be configured, and the configuration options need to be stored.

   In practice these options and settings are stored in text labels, list boxes and drop–down list boxes in the Layer 3 sections found in the dashboard. While the Dashboard Development Services Update Utility replaces the *old* JavaScript, it needs to preserve the options, the `INI` file informs the Dashboard Development Services Update Utility about which objects need to be preserved in the dashboard.
2. **Unconditional Injection of Sections**
   At times, a feature may need to be internally reorganized and a new Layer 3 section, that did not exist, needs to be added as part of every Update operation. The `INI` file contains references to such sections.

Table 22 Example Of Upgrade. ini

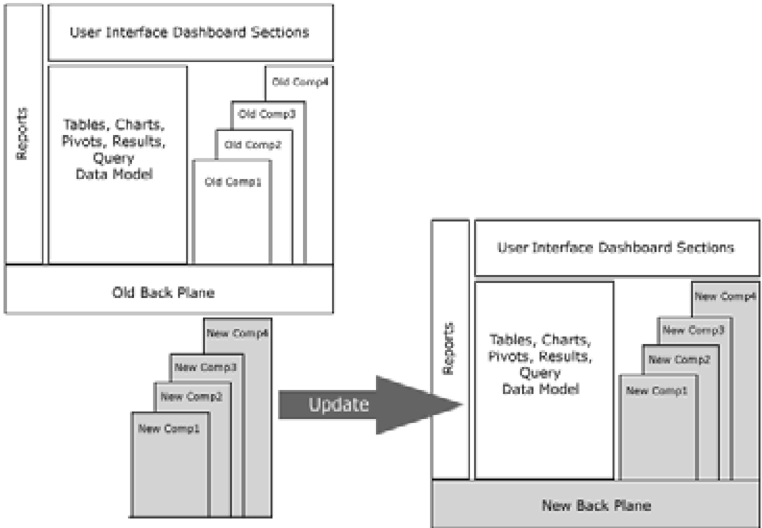
File content	Notes
`[ForceSectionTransfer]`	`INI` file section header
`ForceSection=Qiq_cal`	Force this section in even if one does not exist
`[SectionObjectsTransferValue]`	`INI` file section Header
`SectionName=Qiq_limits`	For the section called *`Qiq_limits`*
`ObjectName=gtxlProperties`	Preserve values in the document found in the shape called *`gtxlProperties`*

In summary, the Update operation rests on these assumptions:

- JavaScript is developed in layers.
- A layer is one or more Interactive Reporting document sections that together implement some discrete function or set of related functions.
- Document scripts are treated as if they were sections with `onStartUp`, `onShutDown`, `onPreprocess`, `onPostProcess` events.

Update Workflow

This diagram illustrates an Interactive Reporting document whose JavaScript code is implemented in layers. The user interface dashboard sections contain GUI controls (buttons, check boxes, charts, pivots, and so on) that use the services of JavaScript functions declared in the *invisible* infrastructure of the Interactive Reporting document. This is labeled *Back Plane* and *comp1* to *comp4*.



The process for updating dashboard sections is:

1. Create or update `newsections.dat`
2. Configure `INI` file
3. Add and remove sections
4. Update documents

Note: Steps 1 to 3 are optional. If you have not customized templates and do not want to, proceed directly to Step 4 to update documents.

Creating New Sections Files

The Dashboard Development Services Update Utility takes a file called *newsections.dat* as its input (an Interactive Reporting document with the suffix renamed). It contains the latest version of the infrastructure (the shared JavaScript function and object constructors). The utility opens each Interactive Reporting document in a nominated list and performs a compare operation which checks if section names from the list exist in the `newsections.dat` and the Interactive Reporting document to be updated. If a section does exist in both, the section in the Interactive Reporting document is removed and replaced by the section from the `newsections.dat`.

The creation of `newsections.dat` is typically carried out only once at the end of a development and test cycle. You can create this file by whatever means is most convenient to you. For example,

- By taking a test application and deleting all the sections that you do not want to be part of the infrastructure.
- By using the Dashboard Studio Merge Utility to create an Interactive Reporting document that includes a composite of all the fragments of infrastructure you may contain in a components library.

Note: Document Scripts are sourced from the primary document when using Dashboard Studio Merge Utility.

Updating New Sections Files

Follow these procedures to update the `newsections.dat` file.

➤ To update a `newsections.dat` file:

1. **Open a command window.**
2. **Move to the directory in which Dashboard Architect is installed.**
3. **Create a folder called *ns_back.***
4. **In the command window, type this code:**

   ```
   Update.exe/n:<File Path> /b:ns_back newsections.dat
   ```

   Where `<File Path>` is the location of the Interactive Reporting document that contains the latest release of the infrastructure. This command updates the file called *newsections.dat* in the current working folder by using the file specified by the /n: directive, and makes a backup to the ns_back folder.
5. **Close the command window.**
6. **From the installation folder, drag `newsections.dat`, into Interactive Reporting Studio to verify that it contains the latest sections.**

   A section is replaced if a section in the nominated document has the same name as a section in this `newsections.dat`.

Configuring INI Files

The Dashboard Development Services Update Utility enables you to move values from the contents of text labels, drop–down list boxes, and list boxes from the old dashboard sections that are about to be discarded, and copy them into the equivalent shapes of the new sections when it is inserted into the Interactive Reporting document.

The `Upgrade.ini` file, found in the Update Utility folder, is used to identify the text labels, drop–down list boxes, and list boxes in the dashboard sections whose values are to be transferred when updating a file. The values to be maintained must be specified in this file.

Note: `Upgrade.ini` need only be configured ***if*** there are sections or values that you want to keep. Otherwise, `Upgrade.ini` can be used as is.

➤ To configure the `INI` file:

1. **From the Dashboard Development Services Update Utility folder, open** `Upgrade.ini`**.**
2. **Type this line into the empty text file:**

   `[SectionObjectsTransferValue]`
3. **Under this line, type** `SectionName=[Name of Section]` **where** *`[Name of Section]`* **is replaced by the name of the section that contains the values to be maintained.**

   Specify the name of the text labels, drop–down list boxes, or list boxes to maintain the values in. The names must take this form:

   `ObjectName=[txlMyTextLabel]` where *`[txlMyTextLabel]`* is the name of the text label.

   `ObjectName=[drpMyDropDown]` where *`[drpMyDropDown]`* is the name of the drop–down list box.

   `ObjectName=[lbxMyListBox]` where *`[lbxMyListBox]`* is the name of the list box.

   List as many object names as necessary.
4. **To maintain values in another dashboard section, leave a line space and specify the** `SectionName` **and** `ObjectName`**. For example:**

```
[SectionObjectsTransferValue]
SectionName=MyDashboard
ObjectName=txlMyTextLabel
ObjectName=drpMyDropDown


SectionName=MyDashboard2
ObjectName=lbxMyListBox
ObjectName=txlTextLabel
ObjectName=drpMyDropDown
ObjectName=lbxListBox
```

Adding and Removing Sections

By default, the Dashboard Development Services Update Utility compares the available sections in the documents to update and the specified new sections file. If a section exists in the `newsections.dat` file and the Interactive Reporting document to be updated, the corresponding section from the new sections file replaces the section in the Interactive Reporting document to be updated.

The Dashboard Development Services Update Utility also enables you to specify sections that are to be added to the documents to update even if these sections did not previously exist. Similarly, you can specify sections to be removed from the documents to update.

➤ To add a section:

1 **Double-click the `Upgrade.ini` file that is installed with the Dashboard Development Services Update Utility.**

2 **Search for a section called *[ForceSectionTransfer].***

If this section does not exist, you can create it at the end of the document.

3 **Under this heading, type `ForceSection=[section name]`.**

For example, to add the section names *NewDashboard1* and *NewDashboard2*, add this text to the `Upgrade.ini` file.

```
[ForceSectionTransfer]
ForceSection=NewDashboard1
ForceSection=NewDashboard2
```

The section is added to the documents to update, provided that a section with this name exists in the specified `newsections.dat` file.

Note: To add a section to a Dashboard Studio template or an editable source master (ESM) only, and *not* to a runtime version, ensure that `DesignOnly=[section name]` is typed into the `Upgrade.ini` file under `[ForceSectionTransfer]`.

For example, to add the section name *NewDashboard3* so it is *not* included in a runtime deployed version, this text is added to the `Upgrade.ini` file.

```
[ForceSectionTransfer]
DesignOnly=NewDashboard3
```

➤ To delete a section:

1 **Double-click the `Upgrade.ini` file that is installed with the Dashboard Development Services Update Utility.**

2 **Search for a section called *[DeleteSections].***

If this section does not exist, you can create it at the end of the document.

3 **Under this heading, enter one or more `DeleteSection` specifications.**

For example:

```
[DeleteSections]
DeleteSection=OldDashboard1
DeleteSection=OldDashboard2
```

The specified sections are removed from the updated documents.

Updating Documents

The Dashboard Development Services Update Utility enables you to update documents in one of three ways:

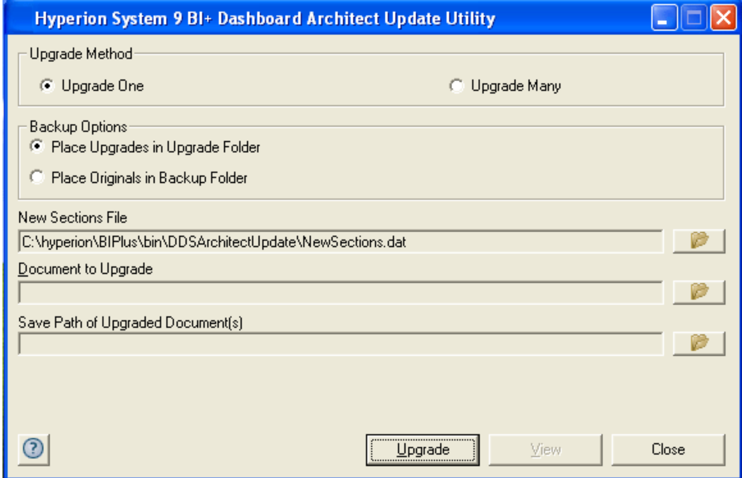
- Using the Upgrade One Method—enables you to update one Interactive Reporting document at a time by using a GUI.
- Using the Upgrade Many Method—enables you to update a folder of Interactive Reporting documents by using a GUI.
- Upgrade a list of documents by using Command Line Updates.

Using the Upgrade One Method

Update a single Interactive Reporting document at a time by using the Update Utility GUI.

➤ To update a single Interactive Reporting document:

1 **Open the Dashboard Development Services Update Utility window.**



2 **From the Upgrade Method section, select the Upgrade One command.**

3 **Select the preferred backup option.**

- If you select the Place Upgrades in the Upgrade Folder option, an Upgrade folder is created in the source path and the updated document is saved to this folder.
- If a document with an identical name exists in the Upgrade folder, a timestamp is added to the document you are currently updating. The original document in the Upgrade folder is not overwritten.
- If you select the Place Originals in Backup Folder option, a Backup folder is created in the source path. The original document is saved to this Backup folder with a timestamp added to the file name. The updated document is saved to the source path and takes the name of the original document.

When you select the option to create a backup of the document, the original document must not be set to read–only.

4 **Click , next to the New Sections File text box, and navigate to the location of the new sections file to use.**

A new sections file is provided by default in this text box, which contains the latest version of the infrastructure sections.

5 **Click , next to the Document to Upgrade text box, and navigate to the location of the document to update.**

The save path of the updated document is generated in the Save Path of Upgraded Document(s) text box.

6 **Click Upgrade.**

When the update process is complete, a report confirms a successful or unsuccessful update.

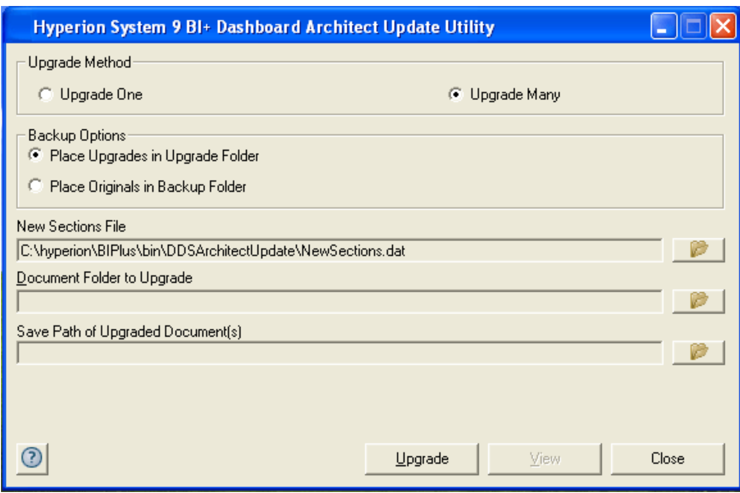
7 **Click View to view the updated document.**

Using the Upgrade Many Method

Update multiple Interactive Reporting documents by using the Update Utility GUI.

➤ To update a folder of Interactive Reporting documents:

1 **Open the Dashboard Development Services Update Utility window.**



2 **From the Upgrade Method section, select the Upgrade Many command.**

3 **Select the preferred backup option.**

- If you select the Place Upgrades in the Upgrade Folder option, an Upgrade folder is created in the source path and the updated documents are saved to this folder.
- If a document with an identical name exists in the Upgrade folder, a timestamp is added to the document you are currently updating. The original document in the Upgrade folder is not overwritten.
- If you select the Place Originals in Backup Folder option, a Backup folder is created in the source path. The original documents are saved to this Backup folder with a timestamp added to the file name. The updated documents are saved to the source path and take the name of the original documents.

When you select the option to create a backup of the documents, the original documents must not be set to read–only.

4 **Click , next to the New Sections File text box, and navigate to the location of the new sections file to use.**

A new sections file is provided by default in this text box, which contains the latest version of the infrastructure sections.

5 **Click , next to the Document Folder to Upgrade text box, and navigate to the location of the folder to update.**

The save path of the updated documents is generated in the Save Path of Upgraded Document(s) text box.

6 **Click Upgrade.**

When the update process is complete, a report confirms a successful or unsuccessful update.

7 **Click View to view the updated folder.**

Command Line Updates

The Dashboard Development Services Update Utility gives you the option of updating documents by using a command line. The major advantages of this method include:

- You can update multiple documents from different locations simultaneously.
- You can build a permanent list of documents to update.

The following command line flags are available when updating from a command line.

Table 23 Command Line Flags

Flag	Description	Example
`/r`	Replaces files with no backup created.	`Update.exe /r <file name>`
`/b`	The original document is copied to the specified backup directory first before being updated. The updated document takes the place of the original document. **Note:** Backup files overwrite files that may exist in the backup directory with an identical name. The backup directory must be a relative or absolute directory name.	`Update.exe /b:<backup file directory> <file name>`
`/l`	Enables you to specify a directory for the log file that is generated with the update process. Log files are saved to the default installation directory of Dashboard Development Services Update Utility.	`Update.exe /l:<log file directory> <file name>`

Note: The `/r` and `/b` flags are mutually exclusive.

The format for a command line update is: `update <file name>`

You must be in the directory where the Dashboard Development Services Update Utility is installed.

For example:

```
Update.exe "C:\Hyperion\BIPlus\bin\DDSArchitectUpdate\Test1.bqy"
```

Entering this example updates `Test1.bqy`. The updated document is saved as `Test1_updated.bqy` within the same directory. The original document remains the unchanged. A log file is also generated in the location where the Dashboard Development Services Update Utility is installed.

Other valid commands include:

`Update.exe /b:"C:\Program Files\Backup" /l:"C:\Program Files\Logs" Test1.bqy` (creates a backup of `Test1.bqy` and replaces the original Interactive Reporting document with the updated Interactive Reporting document. A log file is saved to `C:\Program Files\Logs` folder).

`Update.exe /r /l: "C:\Program Files\Logs" "C:\Hyperion\BIPlus\bin\DDSArchitectUpdate\Test1.bqy"` (replaces the original `Test1.bqy` with the updated Interactive Reporting document. No backup of the original document is created. A log file is saved to `C:\Program Files\Logs` folder).

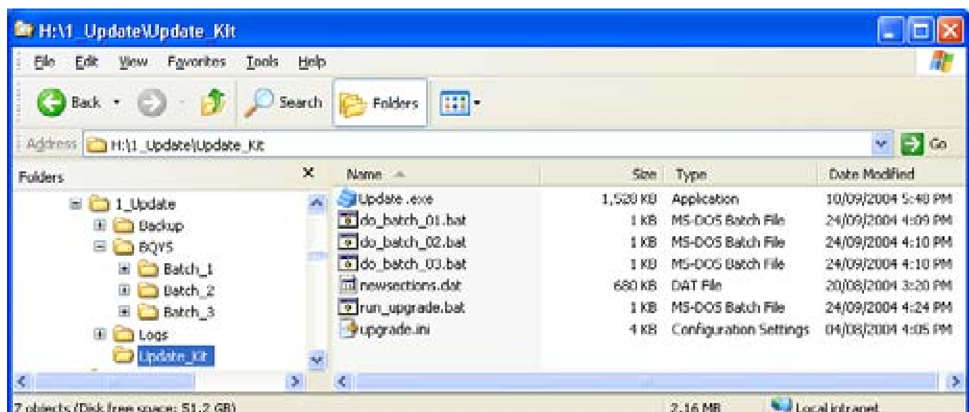
Note: The listed directories are examples.

You can also create batch files that can perform recursive folder searches for documents and use the list of documents returned to perform an update. The following example illustrates how to update a set of documents.

Updating Document Sets

Within a folder called *Update_Kit,* the file, `do_batch_01.bat`, contains instructions to update the first batch of Interactive Reporting documents called *Batch_1* in the BQYs folder.

Figure 15 Batched Interactive Reporting Document Files



Double–click `do_batch_01.bat` to begin the process of updating these files. These are the instructions within do_batch_01 to update Batch_1:

```
set BQYS="H:\1_Update\BQYS\Batch_1\Batch_1\"
set UPGR="H:\1_Update\Upgrade_Kit\run_upgrade.bat"
set NSD="H:\1_Update\Upgrade_Kit\newsections.dat"
set BACK="H:\1_Update\BQYS\Batch_1\Backup\Batch_1"
set LOGF="H:\1_Update\Upgrade_Kit\Logs"

call %UPGR% %NSD% %BQYS% %BACK% %LOGF%
echo done... %BQYS%
```

In detail, the process that has taken place is:

1. `call %UPGR%`. The `run_upgrade.bat` file is called and executed. It is this file that calls the Dashboard Development Services Update Utility executable file (`Update.exe`). `Run_upgrade.bat` contains these commands:

```
@echo off
rem
rem  This batch file updates the dashboard sections in BQYs using
rem  a newsections.dat file.
rem
rem  The first parameter is the path (including file name) of the
rem  newsections.dat file to be used to perform the update.
rem
rem  The second parameter is the root of a tree of files that have
rem  a BQY suffix that are to be updated using the newsections.dat file.
rem
rem  The third parameter is the folder in which to place backups of
rem  the BQYs as they were before the update.
rem
rem  The fourth parameter is the location of the log file.
rem
set UPGR="H:\1_Update\Update_Kit\Update.exe"
for /R %2 %%F in (*.bqy) do %UPGR% /n:%1 /b:%3 /l:%4 "%%F"
@echo on
```

2. `call %NSD%`. The `newsections.dat` file is called and implemented.
3. `call %BQYS%`. The Batch_1 file is updated in the BQYs folder.
4. `call %BACK%`. A backup is made of Batch_1 in the Backup folder.
5. `call %LOGF%`. This procedure is recorded in a log file.
6. `%BQYS%`. The Interactive Reporting document files in Batch_1 are updated.

This process is repeated by double–clicking the remaining `do_batch_`*`nn`*`.bat` files in the Update_Kit folder.

The `do_batch_02.bat` file contains directions to update Batch_2:

```
set BQYS="H:\1_Update\BQYS\Batch_2\Batch_2\"
set UPGR="H:\1_Update\Upgrade_Kit\run_upgrade.bat"
set NSD="H:\1_Update\Upgrade_Kit\newsections.dat"
set BACK="H:\1_Update\BQYS\Batch_2\Backup\Batch_2"
set LOGF="H:\1_Update\Upgrade_Kit\Logs"

call %UPGR% %NSD% %BQYS% %BACK% %LOGF%
echo done... %BQYS%
```

`Do_batch_03.bat` contains directions to update Batch_3:

```
set BQYS="H:\1_Update\BQYS\Batch_3\Batch_3\"
set UPGR="H:\1_Update\Upgrade_Kit\run_upgrade.bat"
set NSD="H:\1_Update\Upgrade_Kit\newsections.dat"
set BACK="H:\1_Update\BQYS\Batch_3\Backup\Batch_3"
set LOGF="H:\1_Update\Upgrade_Kit\Logs"

call %UPGR% %NSD% %BQYS% %BACK% %LOGF%
echo done... %BQYS%
```

Update Example: Interactive Reporting Documents in Multiple Locations, One DAT File and Backup Folder

This example displays how the Dashboard Development Services Update Utility can be run in batch mode to update a list of Interactive Reporting documents that reside in a number of locations by using the same `newsections.dat` and the same backup folder. The looping through a hierarchy of folders and subfolders is done by using Microsoft Windows scripting.

```
upgrade_a_List.bat

set P="C:\Hyperion\BIPlus\bin\DDSArchitectUpdate\Update.exe"
set N="Q:\dev\newsections.dat"
set B="Q:\dev\up_Backup"

set Q="Q:\dev\apps \accounts\sales_esm.bqy"
%P% /n:%N% /b:%B% %Q%

set Q="Q:\dev\apps\payroll\leave_esm.bqy"
%P% /n:%N% /b:%B% %Q%

set Q="Q:\dev\apps\mgmt\kpi_esm.bqy"
%P% /n:%N% /b:%B% %Q%

set Q="Q:\dev\apps\accounts\orders_esm.bqy"
%P% /n:%N% /b:%B% %Q%

pause
```

Update Example: Interactive Reporting Documents in One Location, Two BAT Files

This example illustrates how to use two .BAT files to update all Interactive Reporting documents found under a certain directory structure.

Call this by typing the following information:

```
upgrade.bat "C:\Hyperion\BIPlus\bin\DDSArchitectUpdate\newsections.dat"
"P:\My BQYs" "P:\bqy backups"
```

These are the three parameters:

a. the location of `newsections.dat`

b. the location of the ROOT of the list of Interactive Reporting documents — update every Interactive Reporting document found in the folder and its subfolders

c. where the backups are stored

The content of the .BAT file is windows .BAT file scripting.

Line 1 creates a variable called *P* and points it to the Dashboard Development Services Update Utility.

Note: In this release, it is called *`Update.exe`*.

Line 2 recurses through all folders under parameter #2 (denoted by %2) looking for Interactive Reporting document files. For each Interactive Reporting document file found it calls the Dashboard Development Services Update Utility which is pointed at by *P* passing it the following parameters:

```
/n:%1 (the location of newsections.dat — as passed in by the user)
/b:%3 (the location of the backup folder — as passed in by the user)
"%%F" (the location of the BQY as calculated by the windows iterator)
```

```
run_update_tree.bat

set P="C:\Hyperion\BIPlus\bin\DDSArchitectUpdate\Update.exe"
for /R %2 %%F in (*.bqy) do %P% /n:%1 /b:%3 "%%F"

pause
```

Glossary

BQY A structured document that describes to Interactive Reporting Studio viewer applications how to acquire, manipulate, and present information to users. The term derives from the .BQY extension, which all Interactive Reporting Studio files are assigned.

Breakpoint A line of JavaScript at which the programmer specifies that execution should stop. At a breakpoint, the status of properties and variables is examined or changed and execution is resumed. Breakpoints are used during debugging. One or more breakpoints can be set and execution stops at the first one encountered.

Dashboard A collection of metrics and indicators, which together provide an interactive overview or a snapshot of your business. Another name for a dashboard is a frame. Dashboard Architect is an application that enables the user create the JavaScript to control the interaction between a user and dashboards.

Dashboard Architect An application that consists of a JavaScript editor and debugger for programming dashboards to be run under Interactive Reporting Studio. Dashboard Architect is aimed at people who are developing the logic that manages the interactions that a dashboard section supports.

Dashboard Development Environment A development environment that enables developers or savvy business users to build custom analytic applications by using predefined components, services, or starter kits in a graphical environment that minimizes coding and facilitates rapid prototyping and deployment.
From Wayne W. Eckerson — TDWI Director of Education and Research.

Dashboard Architect and Dashboard Studio together form the Dashboard Development Services for Interactive Reporting Studio.

Dashboard Studio runtime A runtime dashboard is a slimmer version of an ESM dashboard. It contains the same features and properties as the ESM that is was created from, but without build time or development structures. It cannot be opened or modified by Dashboard Studio and is the version usually deployed to users. See *Editable Source Master (ESM).*

Debug Server See *Instrumented.bqy*

Debugger A tool that enables programmers to interact with the executing system during development so as to locate the causes of problems and make fixes. The debugger provided with Dashboard Architect sanctions the setting of breakpoints, the examination of variables at breakpoints, the execution of code, and the viewing of a stack trace.

Editable Source Master (ESM) This contains all the code and structures required by Dashboard Studio at build time and can be opened, modified, or enhanced by Dashboard Studio. An ESM is the source release of a dashboard and is where all development is done. An ESM is a standard, self–contained Interactive Reporting document that can run without being connected to Dashboard Studio. See *Dashboard Studio runtime.*

Event Handler A JavaScript directive that is executed when its associated event is invoked. For example, an `OnClick()` event handler is executed when a button is clicked or an `OnActivate()` event handler is executed when a dashboard section is activated in Interactive Reporting Studio.

Installation Diagnostics A Dashboard Architect feature that provides tools to repair or fix a broken or incomplete installation of Interactive Reporting Studio.

Instrumented.bqy A modified copy of a regular Interactive Reporting document in the Dashboard Architect project structure. It is used instead of the regular Interactive Reporting document while editing, testing, and debugging is done under the control of Dashboard Architect.

Dashboard Architect constructs an instrumented.bqy when the project is being created. The instrumented Interactive Reporting document looks and operates just like the regular Interactive Reporting document in the presence of the Debug Server and Dashboard Architect. JavaScript from the regular Interactive Reporting document is placed in text files outside of the Interactive Reporting document and debug code is added in its place (in instrumented.bqy). This debug code calls the Debug Server, which feeds code to instrumented.bqy in real time. The regular Interactive Reporting document is left unchanged. At the end of the development cycle, a regular Interactive Reporting document is created from instrumented.bqy and the JavaScript in the text files.

Interactive Reporting document See *BQY*

Interactive Reporting Studio An application that provides analytical capabilities to users. It provides the ability to query a variety of data sources in a transparent drag–and–drop manner, to create visualizations of the data that is returned, and to create graphical user interfaces (dashboard sections) for more powerful and friendly interactions with users.

Import Enables you to import non-Dashboard Architect sections (such as charts, pivots, or queries) from Interactive Reporting documents into Dashboard Studio templates to create ESMs. These ESMs can be connected to Dashboard Architect and turned into dashboards. Import performs a similar function to the Dashboard Studio Merge Utility, but it does not copy Dashboard Architect sections into the template.

JavaScript An interpreted programming language, which enables executable content to be included in Interactive Reporting documents. This scripting language contains a small vocabulary, and a simple but powerful programming model, which enables developers to create interactive and dynamic content by using syntax loops, conditions, and operations. While JavaScript is based on the syntax for Java, it is not Java.

Methods Actions that change the state of the object. Programmers write JavaScript to determine the actions that are executed when an event is fired. Dashboard Architect is concerned almost entirely with the scripting of these actions, which typically involve the execution of one or more methods. See *Objects* and *Properties.*

Objects An object is a software package that contains a collection of related characteristics and behaviors. These are also known as properties and methods. Methods are also called functions and event handlers. See *Event Handler, Properties,* and *Methods.*

Properties Characteristics that describe the objects. See *Objects* and *Methods.*

Index

A

add control, 70
add section, 72
adding objects, 69
architecture, 21
audience for this guide, viii
auto-code, 53

B

bqy, 103
breakpoints
- about, 66
- description, 103

buttons, 27

C

case matching, 46
code generation using the object browser, 43
commands, Help menu, x
consulting services, xi
create control, 71
create project, 36
creating a new project, 35
creating a new project from an Interactive Reporting document, 21

D

dashboard, 103
Dashboard Architect
- about, 20
- architecture, 21
- concepts, 18
- description, 103
- menus, shortcuts and buttons, 27
- options, 33
- user interface, 26

Dashboard Architect Import Utility
- description, 104
- importing sections from other Interactive Reporting documents, 60

dashboard development environment
- about, 19
- description, 103

Dashboard Development Services Component-Specific features, 84
Dashboard Development Services Component-Specific features, in HTML documentation, 86
Dashboard Development Services Update Utility
- add and remove sections, 93
- batch update, 98
- command line update, 97
- ini file, 92
- newsections.dat, 91
- overview, 88
- update a set of documents, 98
- update documents, 94
- update process, 90
- upgrade many method, 96
- upgrade one method, 94

dashboard section-level customization
- dashboard controls, 15
- dashboard graphics, 15

debugger, 103
debugging
- about, 66
- breakpoints, 66
- capability, 22

declarative javascript, 64

delete control, 72
delete section, 74
documentation
  comment tag reference, 83
  comments, 76
  Dashboard Development Services Component-Specific features, 84
  Dashboard Development Services Component-Specific features in HTML documentation, 86
  generating, 85
  grouping using the @scope tag, 82
  inclusion and exclusion of documentation groups, 85
  inclusion and exclusion of unscoped documentation, 85
  namespace scope of entities and the @scope tag, 80
documenting
  classes, 78
  Dashboard Development Services Components, 80
  functions, 78
  variables, 77
documents
  conventions used, x
  feedback, xii
  structure of, viii
documents, accessing
  Hyperion Download Center, ix
  Hyperion Solutions Web site, ix
  Information Map, ix
  online help, ix
duplicate control, 70
duplicating a project, 38

E

editable source master (ESM), 103
editing
  auto-code, 53
  code generation using the object browser, 43
  find, 45
  floating find menu, 50
  general notes, 42
  import sections, 60
  javascript, 22
  macros, 54
  match brace, 52
  navigation using the outliner, 43
editing *(continued)*
  printing, 51
  replace, 51
education services, xi
event handlers, 103

F

find
  all, 48
  Find dialog box, 45
  floating menu, 50
  function, 50
  match case, 46
  next, 48
  options, 45
  pattern matching, 46
  replace, 51
  search, 45
  whole words, 45
floating find menu, 50

G

generate code, object browser, 43
generating documentation, 85

H

Help menu commands, x
Hyperion Consulting Services, xi
Hyperion Download Center
  accessing documents, ix
Hyperion Education Services, xi
Hyperion product information, xi
Hyperion support, xi
Hyperion Technical Support, xii

I

import, 104
import sections from other Interactive Reporting documents, 60
installation diagnostics, 103
instrumented.bqy, 104
Interactive Reporting document
  description, 104
  import sections, 60

Interactive Reporting Studio
  description, 104
  synchronizing, 23

J

javascript
  concepts, 14
  dashboard section-level customization, 15
  declarative, 64
  definition of javascript, 14
  description, 104
  document-level customization, 14
  editing, 22
  Interactive Reporting Studio, 14
  procedural, 64

M

macros, 54
making a bqy, 39
making an Interactive Reporting document, 39
match brace, 52
match case, 46
menu commands, 27
menu commands, shortcuts and buttons, 27
methods
  description, 104
  examples, 18

N

navigation using the outliner, 43
new project, create, 36
newsections.dat, 91

O

object browser, code generation, 43
object oriented concepts, 16
objects
  adding and removing, 69
  description, 104
  examples, 16
objects, properties and methods, addressing, 18
Options dialog box, 33

P

pattern matching, 46
prerequisites for using this guide, viii
printing, 51
procedural javascript, 64
project
  create new, 36
  duplicate, 38
properties
  description, 104
  examples, 18

R

recreation of an Interactive Reporting document, 23
removing objects, 69
rename control, 72
rename section, 73
replace, 51

S

script outliner, navigation, 43
search, 45
shortcuts, 27
synchronizing with Interactive Reporting Studio, 23

T

technical support, xii
testing
  capability, 22
  rule, 64
testing and debugging, 63
tools, Options dialog box, 33

U

update
  command line, 97
  documents, 94
  many method, 96
  one method, 94
  set of documents, 98
user interface, 26