Making Sparx Fly

Sparx Application Platform Developer's Guide Volume I

Netspective

Palmer Business Park
4550 Forbes Blvd, Suite 320
Lanham, MD 20706

(301) 879-3321

http://www.netspective.com
info@netspective.com

Copyright

Disclaimer and limitation of liability

Warranty and License Agreement

Customer Support

Customer support is available through e-mail via support@netspective.com.

Contents

Evaluating Sparx 1
Making Sparx Fly Online 1
Making Sparx Fly on your PC 2
Pre-Requisites 2
Installing the Evaluation Kit................ 2
Starting the Application and Web Server (Resin) 2
Getting Started with Sparx 3
Browsing the Source Code 3

Sparx Directory Structure 5
Application Root Structure........... 5
🗀 APP_ROOT/resources 6
🗀 APP_ROOT/sparx 6
🗀 APP_ROOT/temp 6
APP_ROOT/WEB-INF Structure 6
🗀 WEB-INF/classes 7
🗀 WEB-INF/conf 7
🗀 WEB-INF/log.............................. 7
🗀 WEB-INF/lib 8
🗀 WEB-INF/schema 8
🗀 WEB-INF/security........................ 9
🗀 WEB-INF/sql 9
🗀 WEB-INF/tld 9
🗀 WEB-INF/ui 10
Application Configuration Files ... 10
🗎 WEB-INF/web.xml........................11
🗎 WEB-INF/conf/sparx.xml 11
🗎 WEB-INF/conf/app-config.xml 11

Hello World Tutorial 13
Functionality 13
Design...14
Implementation 14
Setting up the Application 15
Create the directory structure 15
Test the New Application in a Browser..16
Verify that ACE is working in the New Application...................................16
Creating the User Interface........ 16
Creating the XML Dialog 17
Unit Testing with ACE...................... 20
Embedding Dialogs into JSP Pages 24
Java Dialogs 26
Compiling the Application 28
Testing the Finished Dialog 29
Another option.............................. 31
Success 31

Sparx Tools 33
Value Sources......................... 33
Types of Value Sources 33
Value Source Declaration Format........ 34
Common Value Sources.................... 35
Available Value Source Classes........... 36
Ant Build Scripts 37
Build Script Prerequisites 37
Build targets................................. 38
Using the Build scripts 39
Upgrading to a New Release of Sparx ... 40
Application Components Explorer (ACE) 40
Application Menu Overview............... 41
Database Menu Overview.................. 41
Documents Menu Overview 42
Common Sections........................... 42
ACE Application Menu Items 43
Application Dialogs Page 43
Application Configuration Page 44
Application Servlet Context Page........ 45
Application Access Control Page 46
Application System Properties Page 46
Application Metrics Page 47
Application Factories Pages............... 48
Application Logs Pages 48
ACE Database Menu Items 48
Database SQL Statements Page 48
Database SQL Query Definitions Page ... 50
Database SchemaDoc Page................ 51
Database Generate SQL DDL Page 52
Database Generate Java DAL Page 52

Database Data Sources Page 53
Database Import DAL Data Page.......... 53
Database Reverse Engineer Page......... 54
ACE Documents Menu Items 54
Browse Source Code Online 54
Execution Environment 56
How to specify the execution environment 56
Datasources and Database Connectivity 57

Sparx Forms Management ... 59
Forms Processing Overview 59
Form Creation Flowchart........... 60
Form Beans: Dialog Context Beans (DCBs) 61
Dialog Data Commands 62
Standard Dialog Data Commands......... 63
How to use Dialog Commands............. 63
Dialog Execution (Sparx Dialogs as Web Services) 64
Field names in Sparx versus HTML 64
Auto Population of Fields Values using URL Request Parameters 64
Auto Execution for Web Services using URL Parameters 65
Auto Execution for Web Services using Java 65
Test Dialogs 66
Adding Test Dialogs to Your Application 66
The Test Dialogs in ACE.................... 66
Conclusion 73

Library Tutorial................ 75
Functionality 75
Design 75
Application Design 75
Database Design............................ 76
Creating the Schema (Data Layer) 76
Starting the Application 77
Setting up the Database 77
The Database Schema 78
Testing the Schema 82
Generating the Data Description Language (DDL) 83
Generating the Object-relational Map (Data Access Layer) 84
Creating the User Interface (Presentation Layer)................. 85
The Java Dialog............................. 87
Creating the SQL (Data Layer) 98
Introduction to Static SQL Libraries 99
Creating Query Definitions (Dynamic SQL) 104
Integrating the Data and Presentation Layers................ 110
Creating a Custom JSP Template Tag..111
Creating the JSP Files 114
Conclusion.......................... 115

Conclusion..................... 119
What's Next......................... 119
Sparx Reference Guide 119
Cura Sample Application and Tutorial .119

CHAPTER 1

Evaluating Sparx

Netspective allows you to evaluate its patent-pending Sparx Application Platform, an enterprise application framework, two different ways.

The first (and quickest) way to evaluate Sparx is to view all the documentation online and follow along with the online versions of the applications created in the tutorials. The main advantages of this method are simplicity and speed. Since all you need is a web browser, you can start learning about Sparx within minutes. You will also be introduced to Sparx's built-in remote development and team programming capabilities. The disadvantage is that you will not have the ability to dabble in or change the application code yourself.

The second (and recommended) way is by downloading a free 30 day evaluation kit or obtaining an evaluation CD direct from Netspective. This kit contains everything you need to start getting familiar with Sparx and learning how to develop applications powered by it. The main advantage of using this method is that you will get a good hands-on look at Sparx and, by the time you are through with the documentation and tutorials, you will be well on your way to developing your own applications using Sparx. The only disadvantage is that you'll need to download and install a Java Developer's Kit (JDK) and the Sparx kit itself on your own server.

Making Sparx Fly Online

An online evaluation is only different from a regular evaluation in the amount of interaction you have with Sparx. While the regular evaluation will enable you to re-create all the demonstration applications that come with the evaluation kit, the online version will allow you to only see the source code and the final products (you will not be able to modify any code).

The only thing you will need to keep in mind while following the tutorials is that all URLs listed in the documentation that are supposed to be pointing to the applications developed in the tutorials will be different. The general rule is that for an application named appName, the URL for its online version should look like the following URL: http://developer.netspective.com/samples/appName.

Therefore, if you are following the development of the Hello World application, instead of using your browser to access the URL mentioned in the documentation (`http://localhost:8080/hello`), you should point your web browser to the following URL: http://developer.netspective.com/samples/hello. Similarly, when following the development of the Sparx Library sample application, you should point your web browser to http://developer.netspective.com/samples/library. Finally, when perusing the tutorials regarding the development of Cura (the project management application), you should go to http://developer.netspective.com/samples/cura.

Making Sparx Fly on your PC

Pre-Requisites

Since Sparx is an application framework for J2EE application servers, a fundamental requirement to develop applications with it is a Java SDK (the full SDK is required, the JRE will not be enough). You can obtain Sun's official Java SDK from its Java web site at http://java.sun.com/j2se/1.3/download.html. This is a link to the Java 1.3. SDK but Java 1.2 and 1.4 will also work.

You can also optionally install a Java integrated development environment (IDE) for developing Sparx applications using the evaluation kit. Sparx does not require any particular Java IDE and if you prefer to use simple text editors like `vi`, `emacs`, or `TextPad` those will do just as well. We recommend Eclipse[1] or JetBrain's IDEA[2] for those who are not already familiar with other IDEs.

Installing the Evaluation Kit

Navigate to the Sparx Support page on Netspective Corp's web site and download the evaluation kit. The evaluation kit comes in the form of a Windows `.zip` or Unix `tar.gz` file. Uncompress the contents of the file into a directory of your choice – however, please realize that this document assumes you're using the default `C:\Netspective` directory.

Starting the Application and Web Server (Resin)

Sparx includes a free development version of an excellent Java application server that doubles as a web server. Resin is a separate product and support for it is available from http://www.caucho.com. Sparx fully supports WebLogic, WebSphere, Tomcat, and many other application servers as well but for evaluation purposes we provide Resin because it's easy to use and fast.

Before attempting to access any Sparx sample application, you need to ensure that Resin is running. There are two methods of starting Resin. The first is as a foreground application that you manually start and leave running while you are working with Sparx. The second is as a Windows service or a UNIX daemon.

You can start Resin in the foreground by opening up My Computer under Windows and navigating to the evaluation kit's installation directory (`c:\netspective` by default). Navigate to the `resin-x.y.z/bin` directory under this installation directory

[1] This is a free, open-source, project available at http://www.eclipse.org.

[2] This is a commercial, but inexpensive, product available at http://www.jetbrains.com.

and double-click on the executable named `httpd.exe`. This should start Resin in foreground mode running the web server and application servers on port `8080`.

If you'd like to use a different port you can modify `resin-x.y.z/conf/resin.conf`. For complete instructions on how to start Resin in the foreground or by deploying it as a NT service/daemon, please visit http://www.caucho.com/resin/ref/httpd.xtp.

Getting Started with Sparx

You can get an idea of the kind of applications that can be developed with Sparx by taking a look at the sample applications that come with the evaluation kit.

You should be able to access the first application – a very simple Hello World application – by opening a web browser to the following URL: `http://localhost:8080/hello`. If you opted for an online evaluation, you can see the Hello World application by pointing your web browser to the following URL: http://developer.netspective.com/samples/hello.

The second application, the Sparx Library, is a more extensive one and is a model of a library of books from which you can add, edit or delete books. This can be accessed at the URL `http://localhost:8080/library`. For online evaluators, the Sparx Library can also be found at the following URL: http://developer.netspective.com/samples/library.

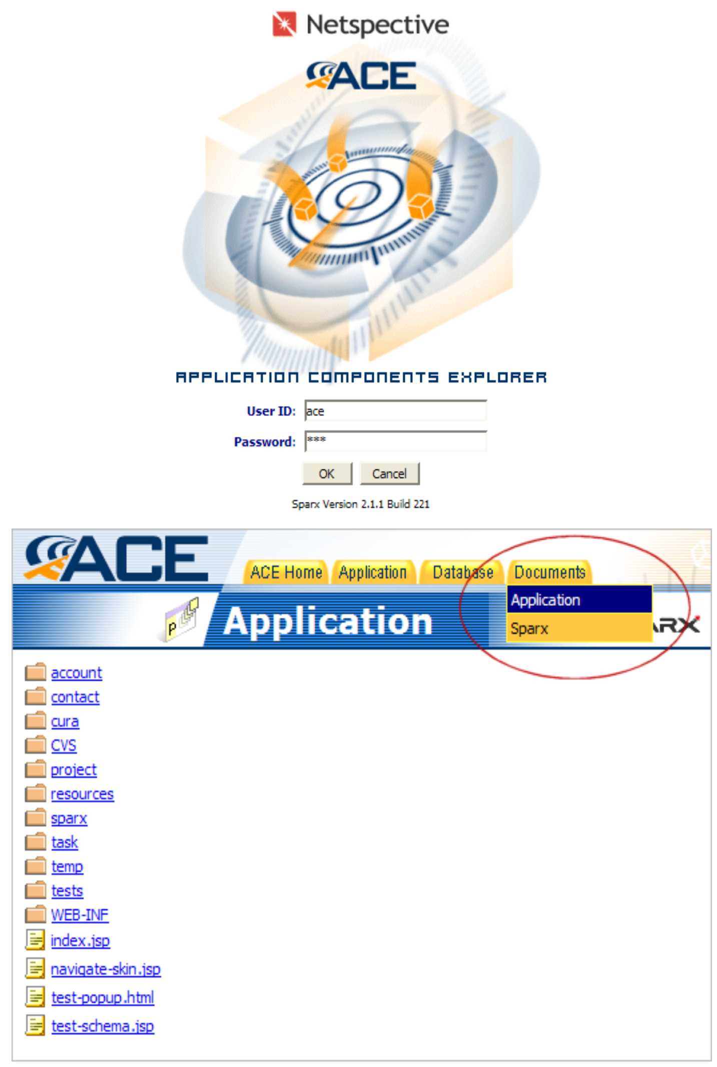
Browsing the Source Code

Sparx's administration console, called the Application Components Explorer (ACE), provides a complete file browser with color syntax highlighting for XML, JSP, Java, JavaScript, and SQL files – this file browser allows you to review source code for sample applications at http://developer.netspective.com or locally if you downloaded the evaluation kit.

You can review the source code for each application by going to the application's ACE and choosing *Application* from the *Documents* menu. You can run ACE by using the "`ace`" suffix in any Sparx application's primary URL. These are the ACE URLs for the Hello World, Library, and Cura applications running online at `developer.netspective.com`:

- http://developer.netspective.com/samples/hello/ace
- http://developer.netspective.com/samples/library/ace
- http://developer.netspective.com/samples/cura/ace

The first time you enter the application documentation section, you should see a list of all the directories that exist in the application's root directory. You can click on any entry to navigate to it and view the list of files and sub-directories inside it. If you click on any XML, JSP or Java source file, you should also be able to see the source for those files directly from the browser.



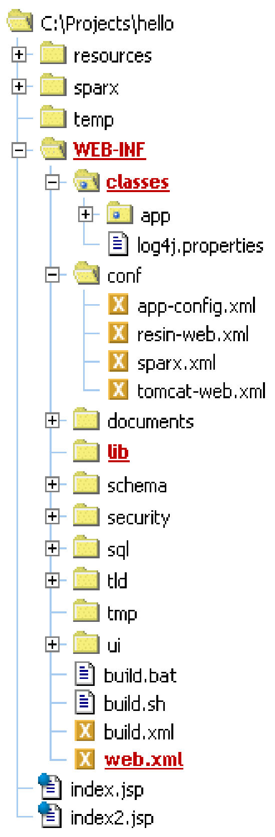
User ID: ace
Password: ***

Sparx Version 2.1.1 Build 221

account
contact
cura
CVS
project
resources
sparx
task
temp
tests
WEB-INF
index.jsp
navigate-skin.jsp
test-popup.html
test-schema.jsp

Figure 1: The default login screen for ACE (user name is `ace`, password is `ace`)

Sparx Directory Structure



```
C:\Projects\hello
  resources
  sparx
  temp
  WEB-INF
    classes
      app
      log4j.properties
    conf
      app-config.xml
      resin-web.xml
      sparx.xml
      tomcat-web.xml
    documents
    lib
    schema
    security
    sql
    tld
    tmp
    ui
    build.bat
    build.sh
    build.xml
    web.xml
  index.jsp
  index2.jsp
```

Every application that is created using the Sparx Application Platform needs to conform to the standard J2EE Servlet application directory structure[3]. This way, all the different components of the application are stored in predictable locations. Additionally, all application servers conformant to the Java Servlet standard will be able to find and run the application components without reconfiguration.

Before we delve into the structure of a Sparx application, it needs to be noted that starting a Sparx application from scratch is made considerably easier with the build mechanism bundled with Sparx. Sparx relies on the (now commonplace) Apache Ant[4] build tool to help not only compile your application's source code into binary Sparx applications but also to take care of numerous housekeeping chores. The supplied `build` script creates a skeleton directory structure suitable for starting a new application. You can then rename and modify this directory structure with your own application data. We will go over this process in detail in the next chapter.

The screenshot shows a tree view for the Sparx *Hello World* application. Once you have some experience with Sparx you can change any of the default locations by modifying `WEB-INF/conf/app-config.xml`. **Please realize that directories formatted like this (bolded and underlined in red) are required by the J2EE Servlet specification so you should not try to alter those locations.**

Application Root Structure

The `APP_ROOT` directory contains all the browser-accessible files for the application and in the screen shot is called `C:\Projects\hello`. This is commonly referred to as the *Document Root* for a website because it is the root directory visible to web browsers. It also contains a private directory, called `WEB-INF`, for the application to store Sparx and Java servlet related files. More on this directory further in these pages. All files in the application's root directory are accessible through a web browser. All subdirectories in the application root other than `WEB-INF` will also be directly accessible through a browser. Therefore, if you put an `index.jsp` file in this directory, you should be able to access it using a URL of the form http://host:port/appName/index.jsp.

[3] You can get a copy of the Java Servlet Specification directly from Sun Microsystems' Web site at the following URL: http://java.sun.com/products/servlet/download.html

[4] You can learn more about Apache Ant at its official web site: http://jakarta.apache.org/ant/index.html

APP_ROOT/resources

This directory contains your application's images, javascript files, CSS stylesheets, or any other "resources" that need to be served to a web browser. This directory is a recommended best-practice, not a Sparx requirement.

APP_ROOT/sparx

This directory contains all of the Sparx shared files. Web browser resources such as style sheets, JavaScript sources, images, and ACE files are placed here. If you modify files in this directory, please realize that when you run the `build upgrade-sparx` target you may lose your changes (please see *Ant Build Scripts* on page 37 for more information about the build scripts).

APP_ROOT/temp

This directory contains temporary files generated by Sparx (or your own application) that need to be served to end-users through a web browser. For example, if a Sparx report is generated that needs to be downloaded by an end user, the downloadable file will be placed here. This directory should be cleaned up periodically. If you need to create temporary files that should *not* be accessible by an end user's browser you should store those in `WEB-INF/temp` because files in `WEB-INF` are not available to end users of your application.

APP_ROOT/WEB-INF Structure

The `WEB-INF` directory is required by the J2EE Servlet Specification. It contains all files private to the application, meaning none of the files in this directory will be accessible to an end-user's web-browser (unless you turn on ACE, which optionally allows browsing of source files in `WEB-INF`).

Sparx uses the `WEB-INF` directory to its external resources. Each separate category of items has its own subdirectory (but you can choose to place everything into a single directory if you wish – the actual locations are controlled by `WEB-INF/conf/app-config.xml`). Each of these categories and corresponding sub-directories is listed here.

Four very important files in `WEB-INF` include `build.bat`, `build.sh`, `build.xml`, and `web.xml`. The build files (`build.bat` for Windows, `build.sh` for UNIX, and `build.xml`for Ant) manage compilation and deployment of your application (please see *Ant Build Scripts* on page 37 for more information about the build scripts).

The `web.xml` file configures your application for your J2EE Servlet container and you should refer to your application server's documentation for how to configure the contents of that file. For Resin (the application server included in the Sparx Evaluation Kit) you can refer to the documentation available at http://www.caucho.com.

🗁 WEB-INF/classes

This directory, which is a part of the J2EE Servlet Specification, holds all the custom Java source code written for the application. After the application is built, each Java source file in this directory contains a corresponding compiled version in the same location as the source.

As noted in the section describing the `UI` directory, the automatically generated Java dialogs and dialog context beans (DCBs or form beans) for each dialog are stored in the `WEB-INF/ui/classes` directory. The generated object-relational map classes (Data Access Layer, or DAL) are also stored elsewhere (in `WEB-INF/schema/classes`). This is done mainly to avoid cluttering up the `classes` directory with auto generated source and binary Java files. So, you can be sure that any classes stored in `WEB-INF/classes` may be safely modified.

All Java classes in `WEB-INF/classes` are automatically included in the `classpath` of the application. Therefore if you have declared a dialog (in the `dialogs.xml` file) to have a corresponding Java version for complete or partial dialog processing, the Java source and compiled versions should be located somewhere in this directory structure. Any auxiliary Java classes that you might need should also be placed here.

By default, you should place all of your Java classes in the directory `WEB-INF/classes/app` because certain application servers will not work with Java classes that are not in a package.

🗁 WEB-INF/conf

This directory, specific to Sparx, contains the main Sparx configuration files called `WEB-INF/conf/sparx.xml` and `WEB-INF/conf/app-config.xml` . It also contains sample `web.xml` configuration files for application servers like Resin and Tomcat.

🗁 WEB-INF/log

This directory, which is **not** created by default but is highly recommended, contains all logs generated by Sparx and your application. These logs are generated using the Log4J[5] library mentioned earlier. The six different log files will contain the following information. Unless you change the `WEB-INF/classes/log4j.properties` file to point the log files to the `WEB-INF/log` directory, the following files will end up in the application server's directory.

FILE	PURPOSE
`page-debug.log`, `sql-debug.log`, `security-debug.log`	These files contain extended debug information regarding each page, SQL statement and security check that is processed by Sparx. It will contain variable information, depending which part of the page reports debug information.
`page-monitor.log`, `sql-monitor.log`, `security-monitor.log`	These files contain statistics regarding each page rendered, SQL statement executed and security check examined in a Sparx application. This information tends to be brief.

[5] For more information on Log4J, please visit http://jakarta.apache.org/log4j.

Table 1: Sparx Log Files

WEB-INF/lib

This directory, which is a part of the J2EE Servlet Specification, holds all the Java Archive (JAR) files needed by your application. These include not only JAR files needed for Sparx but also extra JAR files needed by your own Java classes. At a minimum, this directory will hold the following files.

FILE	DESCRIPTION
README	This file is a text file that provides the version numbers and sources for all the libraries contained in the WEB-INF/lib directory.
`sparx.jar`	This is the entire Sparx library compiled as a JAR file. This file is needed for all Sparx applications
`oro.jar`	The Jakarta-ORO Java classes are a set of text-processing Java classes that provide Perl5 compatible regular expressions, AWK-like regular expressions, glob expressions, and utility classes for performing substitutions, splits, filtering filenames, etc.
`ant.jar`	The Jakarta Ant Java library is a build management tool.
`xalan.jar`	Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath). Its primary use is by the ACE component of Sparx and, if excluded, will break ACE among other things.
`xerces.jar` `xml-apis.jar`	The Xerces Java Parser supports the XML 1.0 recommendation and contains advanced parser functionality, such as support for the W3C's XML Schema recommendation version 1.0, DOM Level 2 version 1.0, and SAX Version 2, in addition to supporting the industry-standard DOM Level 1 and SAX version 1 APIs.
`log4j.jar`	This is a free library that allows Java programs to use a very powerful logging feature

Table 2: Sparx JAR Files

If you would like to, you can place ORO, Ant, Xalan, and Xerces in the application server's common lib directory where they will be available to all applications running under that server. Sparx.jar and Log4J.jar, however are required in each application's `WEB-INF/lib` directory.

WEB-INF/schema

This directory, specific to Sparx, contains an XML representation of the database schema used by an application. It contains multiple subdirectories, each containing variously processed forms of the same database schema. This directory contains all the files required by the eXtensible Information Framework (XIF) component of Sparx.

The `WEB-INF/schema` directory contains the actual XML definition in the file `schema.xml`. This file can optionally include other XML files, allowing you to modularly create a complex database schema.

The `WEB-INF/schema/ddl` directory contains the SQL generated after Sparx processes the `schema.xml` file (DDL is an acronym for *Data Definition Language*). This SQL can be executed on the appropriate database server to instantly create regular tables, lookup tables and an assortment of other structures as defined by the XML schema.

The `WEB-INF/schema/java` directory, which is created automatically after an application is built, contains the complete Object-Relational (O-R) map for the database table structure given in the `schema.xml` file. This O-R map, called the Data Access Layer (DAL), is a pure-Java interface to every table, column, and enumeration stored in the database. The actual DAL classes are automatically compressed into a single JAR file and stored in `WEB-INF/lib` by the application's `build` script.

Whereas the first type of processed schema (DDL) is meant for use in the database server, this second type of processed schema (DAL) is meant for use by an application developer in her programs to speed up development and isolate her code from database-specific behavior.

🗁 WEB-INF/security

This directory, which is specific to Sparx, contains one file: `access-control.xml`. This is an XML definition of role-based security for the entire application. This file contains definitions of hierarchical roles and permissions that can be assigned to users. This allows developers to compartmentalize information as finely or as coarsely as they desire.

🗁 WEB-INF/sql

This directory, which is specific to Sparx, contains one file named `statements.xml`. It may also contain other files, all of which (like in the case of `dialogs.xml`) may be ultimately included into the one `statements.xml` file. `Statements.xml` contains XML definitions for all pre-defined SQL statements used in the application. These files allow the creation of complex SQL statements with variable (SQL bind) parameters as well as dynamic user-defined queries.

🗁 WEB-INF/tld

Here you can store definitions for custom JSP tags that can be used in JSP pages throughout your application. At the very least, you should have the `sparx.tld` and `page.tld` files here which provide custom JSP tags specific to Sparx and XAF. Without these, embedding any Sparx components in JSP files will not be possible.

JSP custom tags[6] are basically XML-style tags that have user-defined names and user-defined actions. A very common use of JSP custom tags is to create a tag for different elements of a standard corporate web template that will allow the final JSP files to look clean. However, before sending the final JSP file to the browser, your application

[6] To learn more about JSP Custom Tags, try the following URL: http://java.sun.com/webservices/docs/ea1/tutorial/doc/JSPTags.html.

server will process the custom JSP tags using the Java classes you provide and convert every tag into the corresponding HTML.

Each tag's user-defined action is implemented in the form of a Java class with a specific structure. The Java classes needed to handle each tag should be located under the `WEB-INF/classes` directory described earlier; that way the classes will always be in the application's `classpath`.

🗁 WEB-INF/ui

This directory, which is specific to Sparx, contains one file named `dialogs.xml`. It may also contain other files, all of which may be ultimately included into the one `dialogs.xml` file. `Dialogs.xml` contains XML definitions for all dialogs used in the application. These files allow the design of multiple dialogs complete with many different types of fields, field validation and basic actions taken in reaction to specific input to a dialog. These dialogs may optionally have two sub-components stored elsewhere in the directory structure.

COMPONENT	DESCRIPTION
Java handler for the Dialog	The first sub-component is a Java handler for the dialog. Before we give an overview of how the two are related, you should know that all Sparx dialogs are transparently and automatically translated from XML to a default Java class for execution. This default Java class, named `com.netspective.sparx.xaf.form.Dialog`, is able to render all the components specified in the XML and is also able to execute all the tasks defined in the XML. However, it is possible to subclass the default Java class and change the behavior of the dialog according to need. In either case (default or custom Java dialog), these Java dialogs are stored in binary form inside the `classes` directory under `UI`. This classes directory will appear after an application has been compiled using the Sparx build tool.
Dialog Context Bean (the "form bean")	The second sub-component is a Java data bean (called a *Dialog Context Bean* or `DCB`) that is used to access data from all fields of the dialog. Just like the Java dialog, this data bean can either be the default data bean (which is automatically generated by Sparx after scanning each dialog's XML) or a custom bean written for the purpose. In either case, these Java dialog context beans are stored in source and binary form inside the `WEB-INF/ui/classes` and compiled automatically into a single JAR file that is stored in the `WEB-INF/lib` directory.

Table 3: Sub-components related to Dialog XML Files

Application Configuration Files

Each Sparx application must contain three essential configuration files. The first, `WEB-INF/web.xml`, is needed by the application server under which the application will run (WebLogic, WebSphere, Resin, Tomcat, etc). The other two are needed by Sparx to store configuration information related to the application.

🗎 WEB-INF/web.xml

This file, which is a requirement of the Java Servlet Specification[3], contains information specific to the application server under which the application is running. As an example, it might contain a mapping of URLs to specific servlets that should handle those URLs. It might also contain references to data sources (i.e. database handles) referenced in the main application server's configuration. For application servers that run according to a standards compliant servlet specification, this file is necessary.

🗎 WEB-INF/conf/sparx.xml

This file, which is a requirement for all Sparx applications, contains a list of properties necessary for a Sparx application to find its resources. A default `WEB-INF/conf/sparx.xml` will contain information about all the paths and URLs required for Sparx to function with any specific application.

If you modify the contents of this file, please be aware that future upgrades of Sparx may overwrite your changes. Any application-specific properties should reside in `WEB-INF/conf/app-config.xml` described below.

🗎 WEB-INF/conf/app-config.xml

This file contains a list of properties and preferences that point Sparx to all of an application's directories and resources. You are free to add as many properties that may be necessary for your own applications.

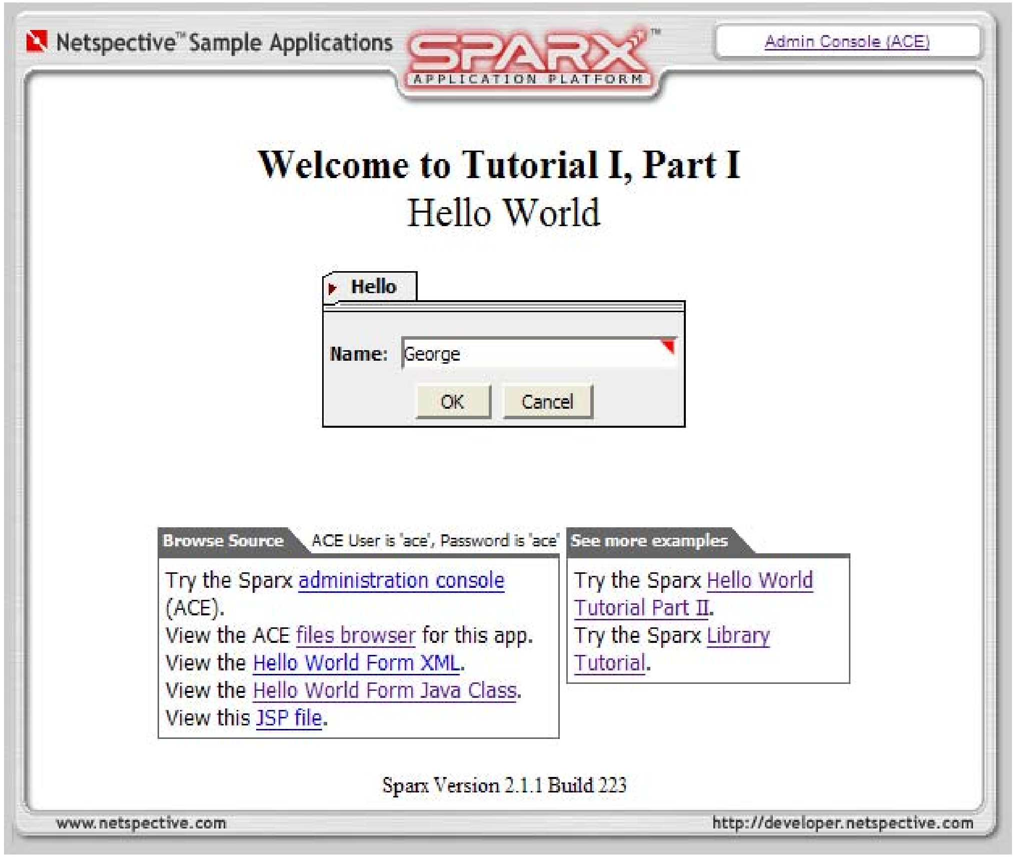
C H A P T E R 3

Hello World Tutorial

The "Hello World" example is a classic in nearly all texts dealing with programming. It is the simplest way to demonstrate the syntax of a language or tool, how to compile a program written in the language and how to run the resulting binary. Sparx is no different. Our first example is the Hello World example with a slight twist.

Functionality

The output of most Hello World programs is a simple output to the screen of the phrase "Hello, World". The Sparx version, however, will have a more interactive version of this example: it will want to get some information about you and then give you a Hello World greeting tailored to the information you provide.


Netspective™ Sample Applications
SPARX™ APPLICATION PLATFORM
Admin Console (ACE)
Hello
OK
Cancel
Browse Source
See more examples
www.netspective.com
http://developer.netspective.com

Welcome to Tutorial I, Part I
Hello World

Name: George

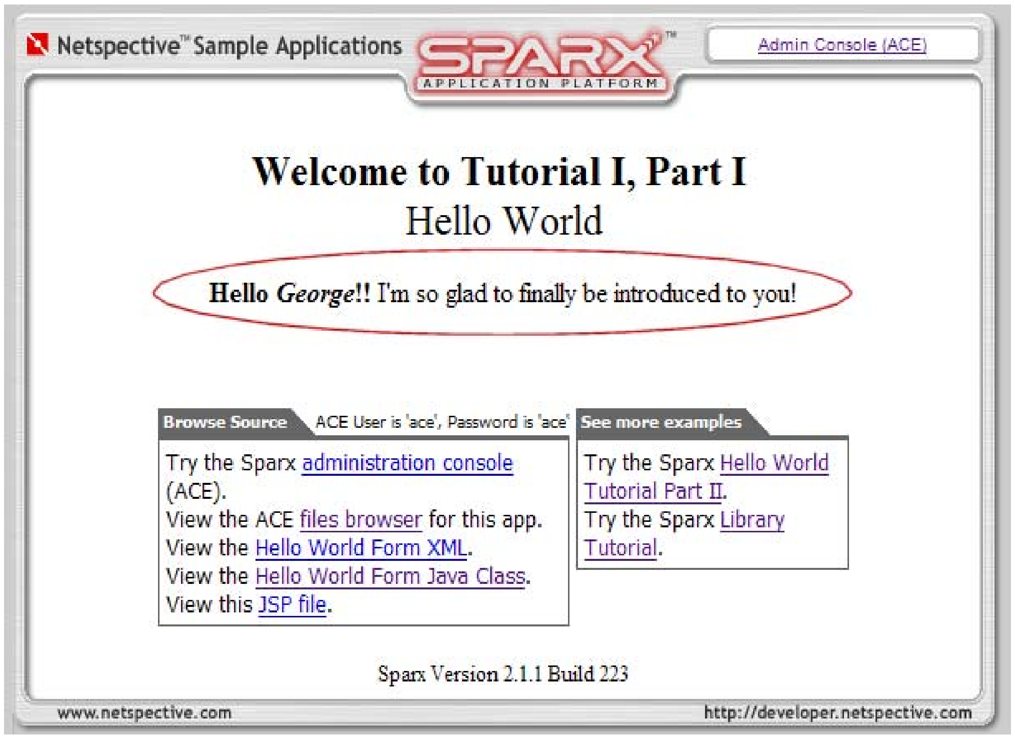
ACE User is 'ace', Password is 'ace'

Try the Sparx administration console (ACE).
View the ACE files browser for this app.
View the Hello World Form XML.
View the Hello World Form Java Class.
View this JSP file.

Try the Sparx Hello World Tutorial Part II.
Try the Sparx Library Tutorial.

Sparx Version 2.1.1 Build 223

Figure 3: The sample Hello World application available at http://developer.netspective.com/samples/hello


Netspective™ Sample Applications
SPARX APPLICATION PLATFORM
Admin Console (ACE)
Browse Source
See more examples
www.netspective.com
http://developer.netspective.com

Welcome to Tutorial I, Part I

Hello World

Hello *George*!! I'm so glad to finally be introduced to you!

ACE User is 'ace', Password is 'ace'

Try the Sparx administration console (ACE).
View the ACE files browser for this app.
View the Hello World Form XML.
View the Hello World Form Java Class.
View this JSP file.

Try the Sparx Hello World Tutorial Part II.
Try the Sparx Library Tutorial.

Sparx Version 2.1.1 Build 223

Figure 4: Execution results of the sample Hello World application Part I

Design

The core components of this example will be the user input screens and the processors that will generate the greeting on the web page. The user input screens will be created as dialogs defined in the `WEB-INF/ui/dialogs.xml` file while the processors will be created by overriding the default Java dialogs with our own custom Java dialogs located in `WEB-INF/classes/app/form`. These dialog handlers will process all input from the dialogs and output the result on the web page.

Since we do not have any database access in this application, the `WEB-INF/sql/statements.xml` file does not come into play at all. Furthermore, in the interests of simplicity, we will not be creating or using any custom JSP tags for easier web page templates. These, and other advanced features, will be introduced in a later example.

Implementation

We will cover the implementation of the application step by step from the initial creation of the application directory structure all the way to the final testing to ensure it's working properly.

For the purpose of this tutorial we will be assuming you installed the Sparx evaluation kit in the default locations. We will also assume you are using the default port `8080` for your web server. If you chose different values for the installation path and the port number, you should substitute the paths and URLs in our example with your values as needed. The defaults that we will use are listed below.

PROPERTY	DEFAULT
Installation Path	`c:\Netspective`
SPARX_HOME	`c:\Netspective\sparx-x.y.z`
RESIN_HOME	`c:\Netspective\resin-x.y.z`
WEB_APPS_HOME	`c:\Netspective\resin-x.y.z\webapps`
Hello World Install Path	`c:\Netspective\resin-x.y.z\webapps\hello`
Resin Web Server Port	8080

Table 4: Default Installation Paths for Sparx Evaluation Kit's Hello World Sample Application

Additionally, although multiple developers can work in a single development server, we will assume you are developing applications and following this tutorial on the same computer as your installation of the Sparx evaluation kit. Therefore, when you want to test out an application through a web browser, you will need to go to a URL of the form http://host:port/applicationName. Assuming the defaults, the word `host` will be replaced by the word `localhost` to indicate that the server is on the same machine as the web browser. The `port` will be replaced by `8080` and the application name will be replaced by the application you are testing. As an example, if you want to test the Hello World application, you would use the following URL: http://localhost:8080/hello. In the rest of this text, when you encounter a URL of the form http://host:port/applicationName, please take care to replace all variables according to your evaluation kit setup.

If you chose to evaluate Sparx online without going through an installation of the Sparx evaluation kit on your own server, you can access all the applications developed in this and all other documents on Netspective's developer website at http://developer.netspective.com/samples. You should find all examples at a URL of the form http://developer.netspective.com/samples/applicationName.

Setting up the Application

The procedure for setting up any Sparx application we will follow is detailed below; however, none of these steps is required to be performed for the Hello World application because the hello sample is pre-configured for you. To minimize the startup time, we will use the Sparx build scripts (please see *Ant Build Scripts* on page 37 for detailed information). A single script that launches an Ant build file will automatically create all appropriate directories for us along with blank files where they are needed.

Create the directory structure

Go to the `SPARX_HOME/tools` directory and run the `new-sparx-app` script with a parameter that specifies the new directory you would like to create and fill with a "starter" Sparx application. We will assume you're creating

`RESIN_HOME\webapps\hello` for this example. **Again, these steps are only examples and you don't need to actually perform the steps since the Hello World application is supplied as an example in the Sparx Kit.** However, these steps will come in handy once you're ready to start your own Sparx application.

```
cd SPARX_HOME\tools

new-sparx-app RESIN_HOME\webapps\hello[7]
```

Test the New Application in a Browser

Use a web browser to access the root of the application we just created. Therefore, in a web browser, we can go to the following URL: http://localhost:8080/hello[8]. If everything worked as it should, you will see the Hello World welcome page.

Verify that ACE is working in the New Application

Use a web browser to access the Application Component Explorer (ACE) for the application we just created. This will ensure not just the proper configuration of the application but also its proper configuration in relation to Sparx. In a web browser, then, we can go to the following URL: http://localhost:8080/hello/ace[9]. If everything is working, you will see the Sparx ACE login screen. Congratulations! You now have a new empty application upon which you can build. At this time, feel free to log in to your application's ACE. The default login and password of for all applications ACE is the same one word: `ace`. Please refer to *Application Components Explorer (ACE)* on page 40 for a complete ACE reference.

Creating the User Interface

The general steps involved in creation of the user interface (UI) are described below.

STEP	PROCESS
1	Create the XML definition of the interface in the `WEB-INF/ui/dialogs.xml` file. This definition will be used by Sparx to generate a nicely formatted input form on the web page. On its own this XML definition will be able to take user input and store it in internal variables for use by Java Servlets that need the information.
2	Unit test the XML definition in ACE to ensure that the functionality works before integrating it into your application.
3	Add the dialog as a custom JSP tag to all needed JSP files. This allows the dialog to be used in your application. This is not for testing purposes, since the Sparx ACE helps us test every aspect of the dialog without having to resort to creating a stub to test with, but instead is for integrating the dialog into your application's look and feel.

[7] By default, this is `C:\Netspective\resin-x.y.z\webapps\hello`.

[8] If you are evaluating Sparx online, you can access the Hello World application at http://developer.netspective.com/samples/hello.

[9] If you are evaluating Sparx online, you can access the Hello World ACE at http://developer.netspective.com/samples/hello/ace.

STEP	PROCESS
4	Create the Java dialog handler or JSP handler corresponding to this dialog. This Java handler will be able to get the input grabbed by the dialog, process it and output a result to the screen.

Table 5: Steps Required in Creating the Hello World UI

Creating the XML Dialog

To create the XML definition of the dialog, you will have to open up the application's main `WEB-INF/ui/dialogs.xml` file in a text editor such as notepad or your favorite XML editor. In a newly created application, this file will contain nothing more than an XML header followed by opening and closing `xaf` tags. To create dialogs in this XML notation, we will need to create what is known as a *dialog package*. A dialog package is a part of the hierarchy that allows multiple similar dialogs to be grouped under one naming scheme. You can have multiple dialog packages in the same `dialogs.xml` file. You can also separate each dialog package into its own file and use the `include` directive to include these separate files into the main `dialogs.xml` file. Our dialog will go in between the dialog package tags and will therefore be a part of the dialog package section if we look at the `dialogs.xml` file as a hierarchy.

The XML definition of the first version of the dialog is shown below and needs to be inserted in between the xaf tags in the dialogs.xml file. This version, as noted, just asks for the user's name and, in response, greets him or her with a personal greeting.

```
<dialogs package="tutorial">
 <dialog name="hello_first" heading="Hello">
      <field.text name="personName" caption="Name" required="yes"/>
 </dialog>
</dialogs>
```

Step by Step Explanation

```
<dialogs package=”tutorial”>
```

This line, along with its corresponding closing tag (`</dialogs>`) is what creates a dialog package. The name of the package is `tutorial`. This name will also be the prefix for the name of any dialog declared within this package as will be illustrated next.

```
 <dialog name=”hello_first” heading=”Hello”>
```

This line, along with its corresponding closing tag (`</dialog>`) is what creates a dialog. The name of the dialog is `hello_first`. Combined with the name of the package it's declared in, the fully qualified name for this dialog is `tutorial.hello_first`. The heading parameter of this dialog definition is what will appear in the "title bar" of the dialog on the web page.

```
 <field.text name=”personName” caption=”Name” required=”yes”/>
```

This line declares the sole field present in this dialog: a text field called personName. If you are familiar with XML structure, you will notice that this field.text tag does not need an ending tag since it is a one-liner and ends with a "/>" instead of a ">". The value of the name parameter in this field is what the Java dialog handler will need to look for as input data. The value of the caption parameter is what is displayed as a caption to the left of this field on the web page. The value of the required parameter (either a 'yes' or a 'no') determines whether this field is required or not. If it is required, not only will there be server-side checking done on the value of the field but there will also be client-side checking done to ensure a user does not forget to enter a value here. It also means that unless this field is filled, the Java dialog handler will not get to process this dialog's contents. This helps developers focus the Java dialog handler code only on values that are validated by the XML definition of the dialog: a very large speed boost in the development cycle in itself.

Other field types

To ease the work required of developers, Sparx comes bundled with many additional field types. Each field type has numerous parameters that will allow developers to customize its behavior according to the requirements of the application. Some of the field types include such field types as `field.integer`, `field.float`, `field.currency`, `field.date`, `field.grid`, `field.composite` and many more[10]. To get detailed descriptions of what each of these fields does, please visit http://developer.netspective.com/xaf/dialogs/xml.html.

Attributes common to all `<field.xxx>` fields

The following table describes the set of attributes that are allowed by default for all fields. Each field type (like a `<field.integer>`) might define additional attributes for special features. For a complete reference for all dialog controls and their usage, please visit http://developer.netspective.com/xaf/dialogs/xml.html.

NAME	DESCRIPTION
name	The name of the field. If this field is a child of a composite or grid field, the name provided is automatically appended to the parent's name to create a unique name. The actual name of the control when the HTML is generated is usually _dc.*field-name*.
caption	The caption or label that describes the usage of the field to the end user. If this field is a member of a composite or grid field thn the caption will only show if the parent field's show-child-caption is set to yes.
default	The single value source (or list value source if appropriate) that specifies the default value of the field. The default value is the value that is pre-filled into a dialog field when it is initially displayed.
hint	The text that will be shown to a user to provide a hint as to the usage of the field. The exact behavior of this attribute depends upon the skin being used, but typically the contents of the hint attribute are shown right under the field control.
required	Specifies whether the field is required or not. If the field is specified as required, code is automatically generated that will enforce this validation rule.

[10] To learn more about the different types of fields, validations, data management, and other features available via XML please visit http://developer.netspective.com.

NAME	DESCRIPTION
read-only	Specifies whether the field is read-only or not. If the value is set to `yes`, then the field's value becomes a static text string (will not generate a real HTML control). If the value is set to `browser` then the appropriate HTML control is still created but the control is marked read-only so the browser will not allow the value to be changed.
hidden	Specifies whether the field is hidden or not. As a hidden field, the value of the field is still available to the programmer, but there will no field caption/label or input control available to the user.
visible	Specifies whether the field is visible or not. As an invisible field, there is no value available to the programmer nor is there a caption/label, or control available to the user. Setting a field to visible=no is almost like commenting out the field because most dialog skins will not process invisible fields.
create-adjacent-area	Specifies whether to create a <span> element in the HTML adjacent to this field. If set to yes, then a <span> is created with the complete name of this field plus the word _adjacent. For example, if the field name is `customer_id` then the adjacent area will have the id `customer_id_adjacent`. This attribute is very useful when defined in conjunction with popup windows.
col-break	Specifies whether a column break should be included before or after this field. The actual behavior of this attribute is determined by a skin, but typically if this field is a primary (top-level) field, the column break will create a dialog with multiple columns. If this field is a secondary field (child of composite), then a simple line break will be inserted between the composite siblings.
identifier	Specifies whether or not to treat the contents of this field as an identifier. An identifier is a field whose values may only contain uppercase letters, numbers, and an underscore.
initial-focus	Specifies whether to set the initial focus of the dialog to ths field. The last field to have this value set to yes (in creation order) will have the focus when the dialog is first displayed.
persist	Specifies whether or not to automatically remember the last contents of this field (as a browser cookie) for the next time the user loads the dialog. This can be used in the place of the default attribute when the user's last input value should be used as the default for the field.
size	The size (usually the number of characters) of text that the control should display at any given time.
max-length	The maximum length of data that the control should allow.
uppercase	Specifies whether the input should be uppercased or not.
lowercase	Specifies whether the input should be lowercased or not.
trim	Specifies whether the input should be trimmed (all leading and trailing white space removed).
mask-entry	Specifies whether the input should be masked when entered. This means that data entry will be allowed but the control will not show the input. This is useful when getting passwords and other private data.
validate-pattern	Specifies a Perl5 regular expression that should be matched against the input data. This regular expression should be of the form [m]/pattern/[i][m][s][x]. Please see the Jakarta ORO API for more information about regular expressions.
validate-msg	Specifies the error message to display if a validate-pattern is provided but the pattern does not match the input.

NAME	DESCRIPTION
format-pattern	Specifies a Perl5 regular expression that should be used to format the input for display on the screen. The pattern should be of the form s/pattern/replacement/[g][i][m][o][s][x]. Please see the Jakarta ORO API for more information about regular expressions.
<client-js>	Client-side JavaScript specification
<conditional>	Conditional processing declaration
<popup>	Popup window and automatic fill-in of data specification

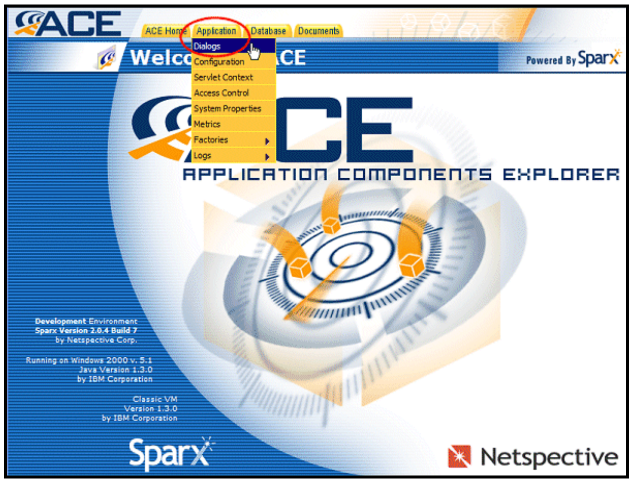
Table 6: Attributes common to all `<field.xxx>` tags

Unit Testing with ACE

Now that you have finished adding the XML definition of the dialog to the `WEB-INF/ui/dialogs.xml` file, save the file so we can get to unit testing this dialog. Unit testing, for those who are not familiar with the term, refers to testing individual components of an application thoroughly before they are integrated with the rest of the application. This allows a developer to run the component through many series of tests to ensure it performs as expected and desired before integrating it into a finished application.

Unit testing for Sparx application components is accomplished very easily with the Sparx ACE. In a web browser, go to the URL http://host:port/hello/ace[11] and you will be presented with the ACE login page. The default username for ACE is `ace` and the default password is `ace`.

Once you log into ACE, you will see an opening screen with pull-down menus near the top of the page. Move your mouse over the *Application* menu item and choose *Dialogs* from the menu that drops down. This will bring up a list of all the dialogs available to the application, the current options set for dialogs as well as a list of all the XML source files that were scanned to build the list of dialogs.



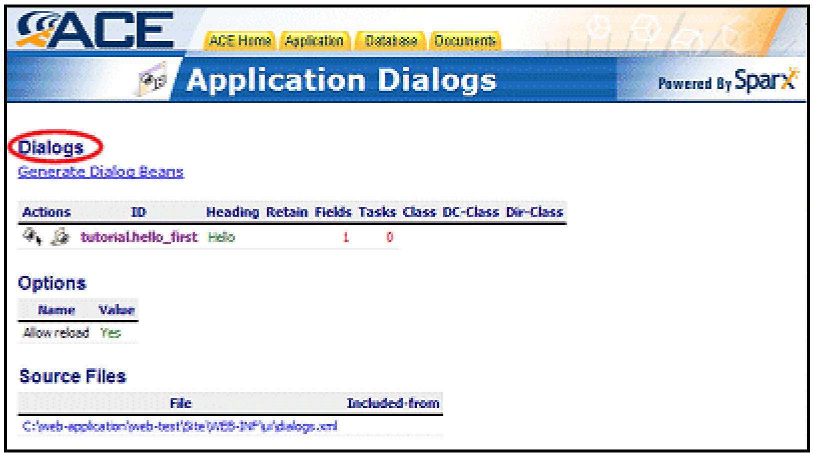
Development Environment
Sparx Version 2.0.4 Build 7
by Netspective Corp.

Running on Windows 2000 v. 5.1
Java Version 1.3.0
by IBM Corporation

Classic VM
Version 1.3.0
by IBM Corporation

[11] If you are evaluating Sparx online, you can access the Hello World ACE at http://developer.netspective.com/samples/hello/ace.

Figure 5: Getting To The Application Dialog Screen



Dialogs

Generate Dialog Beans

Actions	ID	Heading	Retain	Fields	Tasks	Class	DC-Class	Dir-Class
	tutorial.hello_first	Hello		1	0			

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\web-application\web-test\Site\WEB-INF\ui\dialogs.xml	

Figure 6: List Of Application Dialogs.

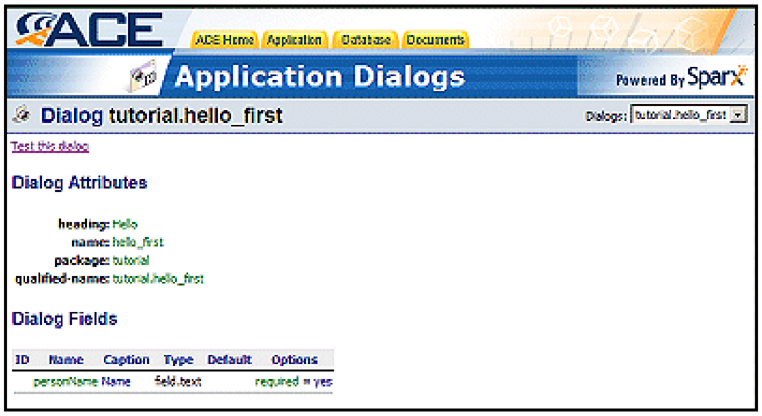
For each dialog, the Application Dialogs screen of ACE will give the following pieces of information. Please see the ACE *Application Dialogs Page* explanation on page 43 for more details.

ENTRY	DESCRIPTION
Actions	The first icon (with the pointer) will allow you to unit test the dialog/form. The second icon allows you to review the functional specifications of the form.
ID	The complete package and name of the form. Clicking this link will show the functional specification of the form.
Heading	Displays the heading of the dialog.
Retain	Allows you to specify which request parameters, if any, are carried through multiple invocations of a form's data management process. A Sparx form automatically manages all of its data (from initial entry through validation through execution). Because Sparx is managing the data, the same form may be displayed multiple times (in case the user enters invalid data). The *retain* option allows URL parameters to be carried through the entire process even through multiple reloads of the same page.
Fields	The number of fields present in the dialog.
Tasks	The number of XML-based tasks that the dialog performs (useful for simple data entry and database management).
Class	The Java class that is bound to the XML representation of the dialog. Since the XML is merely a resource, the class option allows you to completely take control of one or more aspects of a dialog's functionality.
DC-Class	The Java class that manages the form's data. This is a tightly typed Java representation of each of the dialog's XML fields. This class is called a Dialog Context Bean (DCB) and is described further in *Form Beans: Dialog Context Beans (DCBs)* on page 59.
Dir-Class	The Java class that manages the flow of the dialog. The director is responsible for helping the user submit data and navigate to another page.

Table 7: Columns in ACE Dialogs Functional Specifications

Unit Testing

Having introduced ACE, we can now look at how to test our newly created dialog. This dialog should be listed in the ACE Application Dialogs page as `tutorial.hello_first`. Click on the dialog name to bring up complete details about the dialog including a complete list of all the fields that comprise it.



Dialog tutorial.hello_first

Test this dialog

Dialog Attributes

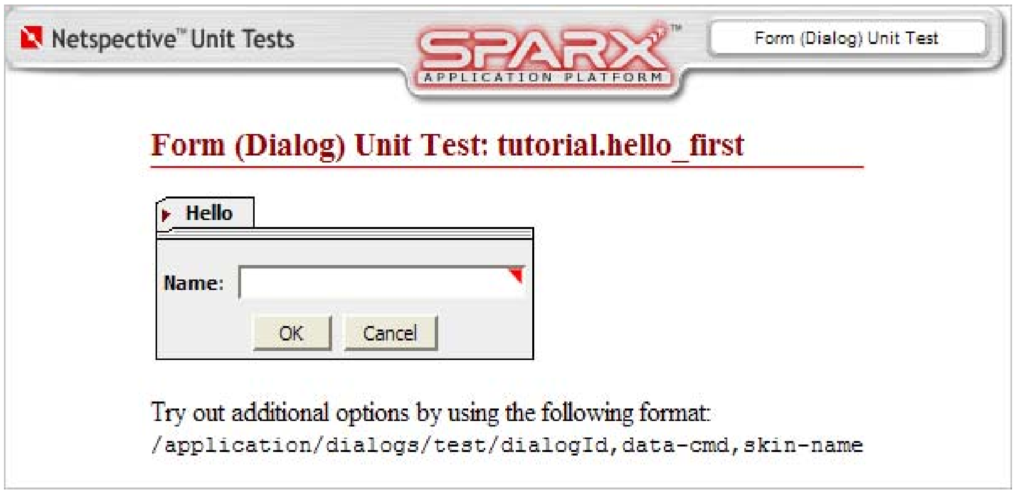
heading: Hello
name: hello_first
package: tutorial
qualified-name: tutorial.hello_first

Dialog Fields

ID	Name	Caption	Type	Default	Options
	personName	Name	field.text		required = yes

Figure 7: `tutorial.hello_first` Application Dialog.

Near the top of this page, you should see a link that says "Test this dialog". Click on that link and a new browser window will open with our `tutorial.hello_first` dialog already loaded and rendered. If everything goes smoothly, the rendered dialog should look something like the screenshot shown.



Form (Dialog) Unit Test: tutorial.hello_first

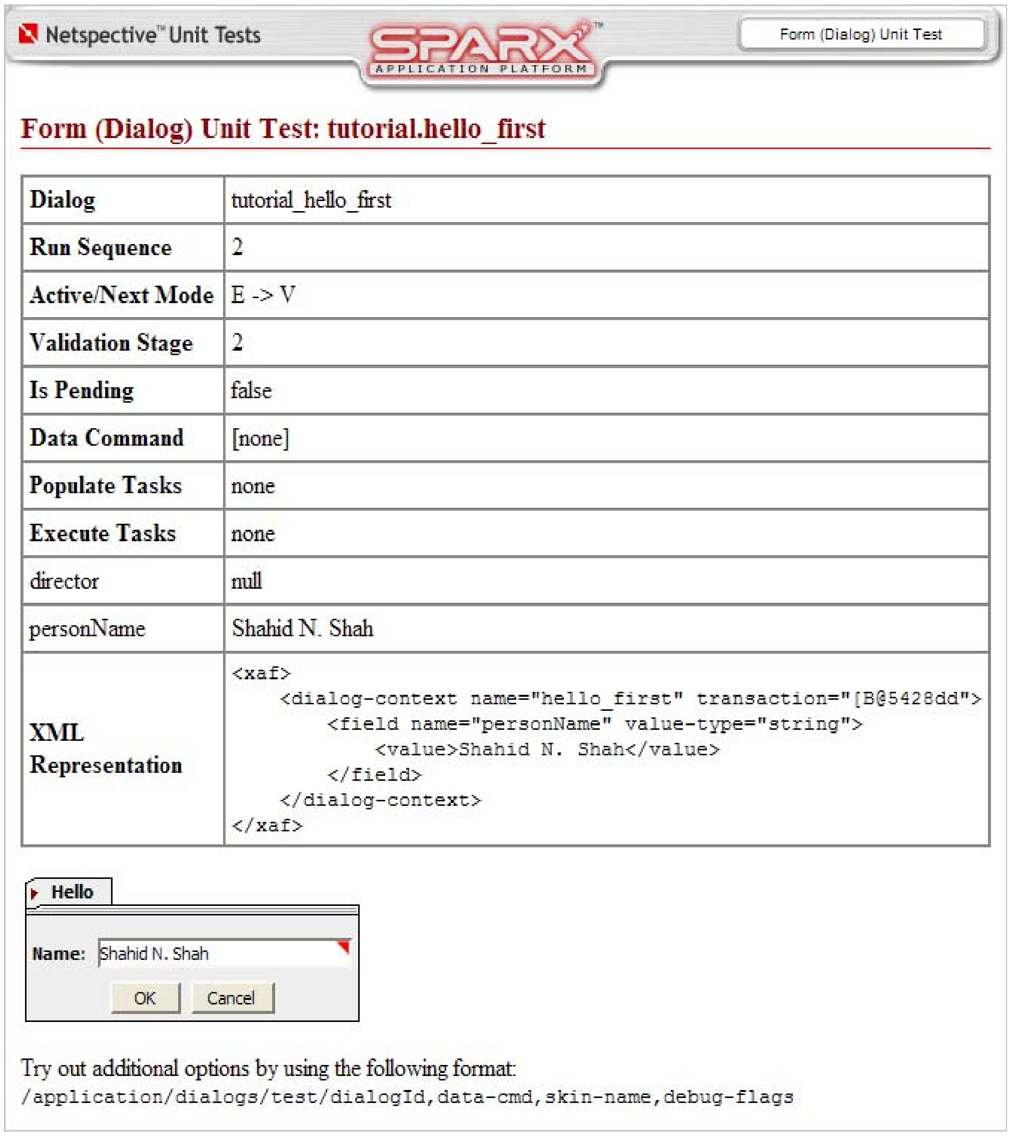
Try out additional options by using the following format:
`/application/dialogs/test/dialogId,data-cmd,skin-name`

Figure 8: Testing The tutorial.hello_first Dialog

As you can see, the rendered result of just a few lines of XML is quite impressive as it stands. What makes it even more powerful, however, is that developers can control the look and feel of a dialog using custom skins.

In the default skin, the little red flag rendered in the text field shows that this field is a required field. This means that the Name field has, as described previously, not only server-side validation but also client-side validation of data. To get an idea of how this works, ensure that the field is empty and try to submit the dialog as it stands by pressing the OK button. You should see a browser window pop up at once to inform you that the Name is a required field.

However, if you enter a value for the Name field and press OK, you will see the result of the default processing built into any dialog. The default processor dumps some debugging information about the dialog that includes, among other things, a list of all the field names in the dialog and their corresponding values (as of the time when you pressed OK). If you enter the word "test" (without quotes) in the Name field, you will see that in the result, `personName` has a value of `test`.



Form (Dialog) Unit Test: tutorial.hello_first

Dialog	tutorial_hello_first
Run Sequence	2
Active/Next Mode	E -> V
Validation Stage	2
Is Pending	false
Data Command	[none]
Populate Tasks	none
Execute Tasks	none
director	null
personName	Shahid N. Shah
XML Representation	`<xaf> <dialog-context name="hello_first" transaction="[B@5428dd"> <field name="personName" value-type="string"> <value>Shahid N. Shah</value> </field> </dialog-context> </xaf>`

```
<xaf>
    <dialog-context name="hello_first" transaction="[B@5428dd">
        <field name="personName" value-type="string">
            <value>Shahid N. Shah</value>
        </field>
    </dialog-context>
</xaf>
```

Try out additional options by using the following format:
`/application/dialogs/test/dialogId,data-cmd,skin-name,debug-flags`

Figure 9: Showing a Run of the First Dialog

At the end of the output of the dialog you will see the same dialog repeated in case you want to test it some more. This behavior is called looping and is available for use in your own dialogs outside of ACE. The appropriate parameter to look up is the `loop` attribute of the `<dialog>` tag.

Once our dialog is working according to our specification, it is time to take the next few steps. The first is to embed it in a real web page so we can see how it will look on the site. The second is to make it do something other than producing debugging information. To the first end, we have to create a JSP file as shown in the next section and use some custom JSP tags that are a part of the Sparx library. To the second end, we have to create a Java dialog handler class for our XML dialog. Once that is done, we have to change the XML definition of our dialog to let Sparx know which class to send the dialog's data to for processing. Another option, which is not being explored here, is that dialogs may be embedded into Servlets as well.

Embedding Dialogs into JSP Pages

Embedding an XML dialog into a JSP page is a process whose simplicity belies the power we are harnessing with just a few custom JSP tags. Let us first see an example of our XML dialog embedded in a JSP file and then go over the salient aspects of the JSP source. The JSP source shown below can be stored as the index.jsp in your `APP_ROOT` directory.

It should be pointed out that in the source shown below the most important line is the one which starts with the `<sparx:dialog>` tag. This one line tag is the bare minimum needed to embed a dialog in a JSP page. The rest of the source is part of the HTML design of this page.

```
<jsp:directive.include file="/resources/include/site-header.jsp"/>

    <center>
        <br>
        <font size="5">
            <b>Welcome to Tutorial I, Part I</b>
            <br>Hello World
        </font>
        <p>

        <!-- create the dialog state machine and form HTML -->
        <sparx:dialog name="tutorial.hello_first"/>
    </center>

<jsp:directive.include file="/resources/include/site-footer.jsp"/>
```

Step by Step Explanation

```
<jsp:directive.include file="/resources/include/site-header.jsp"/>
```

The first line in the file is a JSP command to include the Sparx Tag Library Descriptor using a standard include file so the application server knows what to do when it encounters custom JSP tags that are specific to Sparx. It is essential to include this line at the top of all JSP files when you are using any Sparx elements.

```
<sparx:dialog name="tutorial.hello_first"/>
```

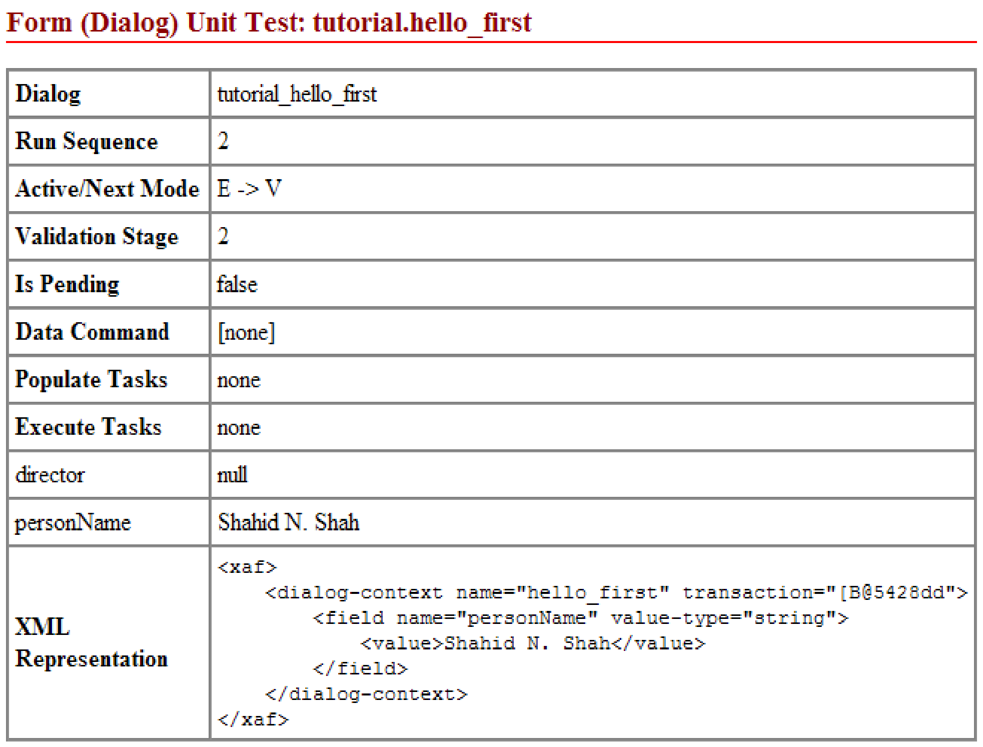
The second salient line in the above JSP source is the Sparx XAF custom tag that actually embeds the dialog into the file. As you can see, the `sparx:dialog` tag is a one-liner that takes one argument which is the fully qualified name of the dialog that needs to be embedded here. Sparx takes care of everything from here on.

As far as the rest of the JSP source is concerned, this example just shows plain HTML. However, using custom JSP tags for templating and for embedding other Sparx controls, we can accomplish a whole lot more. For now, though, this will suffice to demonstrate the ease of Sparx.

Testing Embedded Dialogs

Now that you have a dialog embedded in a JSP page, you can test it simply by pointing your web browser to the URL http://host:port/hello/filename.jsp (where filename.jsp is the name of the file you stored the above JSP source in). If you stored it in index.jsp, you can access your embedded dialog using the URL http://host:port/hello[12].

Try testing out the dialog using empty input and then with some input in the Name field. Since the dialog still does not have a handler to process the data, it will output the same debugging information we saw when unit testing it in ACE. The one difference you will notice, however, is that the dialog is not repeated after the debugging information; that was a facility ACE provided us for testing purposes. However, as mentioned earlier, it is possible to replicate that looping behavior by using the `loop` parameter to the `dialog` tag in `dialogs.xml`.



Form (Dialog) Unit Test: tutorial.hello_first

Dialog	tutorial_hello_first
Run Sequence	2
Active/Next Mode	E -> V
Validation Stage	2
Is Pending	false
Data Command	[none]
Populate Tasks	none
Execute Tasks	none
director	null
personName	Shahid N. Shah
XML Representation	`<xaf> <dialog-context name="hello_first" transaction="[B@5428dd"> <field name="personName" value-type="string"> <value>Shahid N. Shah</value> </field> </dialog-context> </xaf>`

```
<xaf>
    <dialog-context name="hello_first" transaction="[B@5428dd">
        <field name="personName" value-type="string">
            <value>Shahid N. Shah</value>
        </field>
    </dialog-context>
</xaf>
```

[12] If you are evaluating Sparx online, you can access the Hello World application at http://developer.netspective.com/samples/hello

Figure 10: Testing the Embedded Dialog

An observation worth making is that the output of the dialog occupies the exact same place that the rendered dialog used to occupy. Keeping this in mind, it should be a little easier to visualize and design the output of a dialog or any other Sparx component on a JSP page.

Java Dialogs

Creating a custom Java dialog class for an XML dialog requires a developer to keep in mind a few important points. It is necessary to know what Sparx class to use as a starting point and what capabilities will be available to the dialog handler as well as the directory that class needs to go into, how to compile it, and so on. Without much further ado, then, let's present the complete source for the first custom Java dialog.

```
package app.form;

import com.netspective.sparx.xaf.form.DialogContext;

…

public class HelloFirstDialog extends
 com.netspective.sparx.xaf.form.Dialog
{
    /**
     * This dialog greets the user once the user enters a valid
 name.  The method used to process and
     * respond to the dialog is called execute.
     */
    public void execute(Writer writer, DialogContext dc)
    {
        // if you call super.execute(dc) then you would execute
 the <execute-tasks> in the XML; leave it out
        // to override
        // super.execute(dc);
        String personName = "";
        String returnValue = "";

        personName = dc.getValue("personName");

        returnValue = "<b>Hello <i>" + personName + "</i>!!</b>
 I'm so glad to finally be introduced to you!";

        try {
            writer.write(returnValue);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

As noted in the previous chapter, all Java classes that are required to support XML dialogs or other Java classes need to be placed somewhere in the application's `classpath`; the most convenient location is in the `WEB-INF\classes` directory which is automatically in the application's `classpath`. In addition, you will notice that the name of the class created in this bit of source code is HelloFirstDialog and it is part of the package named "tutorial". Therefore, this file should be stored in the `WEB-INF\classes\app\form\HelloFirstDialog.java` file for compilation and integration with the application.

Step by Step Explanation

```
package app.form;
```

This line declares this class and all classes after this line to be a part of the app.form package. By putting a class into a particular package, we are not only giving them a prefix for their fully qualified names and putting them in their own place in the directory hierarchy, but we are also classifying like classes together so there are no namespace conflicts with other similarly purposed classes in the future. The most visible result of this line is, however, that this file is stored in the WEB-INF/classes/app/form directory.

```
import com.netspective.sparx.xaf.form.Dialog;
import com.netspective.sparx.xaf.form.DialogContext;

import java.io.IOException;
import java.io.Writer;
```

These lines make available to our class a few necessary bits of functionality that we are using.

```
public class HelloFirstDialog extends Dialog
```

This line gives us two bits of information. The first is that the core Sparx class that handles all dialogs is known as com.netspective.sparx.xaf.form.Dialog. Armed with this information, you can look at the Sparx API documentation and discover the various methods of this class that can be overridden to change the behavior of your applications. The second piece of information it gives us, in addition to the package declaration above, is the fully qualified name of the class that needs to be added to the XML definition of our hello_first dialog. We will go into an explanation of the syntax used to accomplish this in the next section.

```
        public void execute(Writer writer, DialogContext dc)
```

This line is the beginning of the method declaration within the HelloFirstDialog class. The execute method, which is being overridden from the parent com.netspective.sparx.xaf.form.Dialog class, is responsible for examining all the tasks listed in the XML definition of a dialog and executing them depending on the current state of the dialog. Once overridden, however, we wrest control of this execution from the Sparx engine. Whatever code goes into this method will determine how our dialog will react to data that is input using it. This will become apparent when we discuss the body of this method in the next few paragraphs.

The parameters passed into this method are a Writer and, more importantly, the Sparx DialogContext for the dialog that this class will handle. The dialog context is an object that is a representation, in real-time, of the complete state of the dialog including it's mode of execution and the data it holds in each field. This is probably the most important object you will need to appropriately process the data entered into a dialog by a user.

```
 String personName = "";
 String returnValue = "";

 personName = dc.getValue("personName");
```

```
returnValue = "<b>Hello <i>" + personName + "</i>!!</b> I'm so glad
      to finally be introduced to you!";
```

This half of the overridden execute method is responsible for the main processing of information input into the dialog. After declaring the variables that we will be using, we then proceed to use the `DialogContext` object for the current dialog to get the value of the sole field in the dialog. With this information in hand, we can determine what the output of this dialog will be. We store this output into the variable returnValue.

```
try {
 writer.write(returnValue);
} catch (IOException e) {
 e.printStackTrace();
}
```

In this last part of the execute method we want to output the result of processing the data. Since the `Writer` object's write method throws an `IOException` upon failure, we must enclose our final line of code in a try-catch block so we can ensure our dialog's successful execution. In case the `Writer` fails to write the text to the web page as desired, it will print out debugging information regarding the exception that was thrown. Both outputs are useful: one being the desired output and the other being output that would help us reach our desired goal.

Binding Java to XML

Now that the Java dialog handler is written, it is time to finish the job by binding this dialog handler to the XML dialog whose data it will process. Open up the `WEB-INF/ui/dialogs.xml` file in an editor and look for the line that declares the tutorial.hello_first dialog. Change it to the following:

```
      <dialog name="hello_first" heading="Hello"
      class="app.form.HelloFirstDialog">
```

This new line tells Sparx that the `hello_first` dialog will be handled by the Java class `app.form.HelloFirstDialog`. Now, whenever the `tutorial.hello_first` dialog is used, all its data will be passed to the `HelloFirstDialog` class to process. This is true of any invocation of the `tutorial.hello_first` dialog, whether this be from within ACE or an application. With this final phase of binding the Java dialog handler to the XML dialog definition, the Java dialog handler is complete. When a dialog class is specified in this manner (using the `class` attribute) that class has the capability to take over the entire processing activity of the dialog including the reading of the XML, performing validations, executing code, etc.

Compiling the Application

With the Java dialog handler complete, only one thing stands between us and a final test of the dialog: recompiling the application. Depending on your application server, this might not even be necessary. For example, if you are using Resin (the default application server that ships with Sparx) you will not need to recompile your application. In case your application server does not support automatic compilation, the build file that we used to create the directory structure for our application can be used to recompile our application. In order to do this, you will need to go to the command line prompt and navigate to your application's `WEB-INF` directory using the command

```
cd WEB_APPS_HOME\hello\WEB-INF
```
[13]

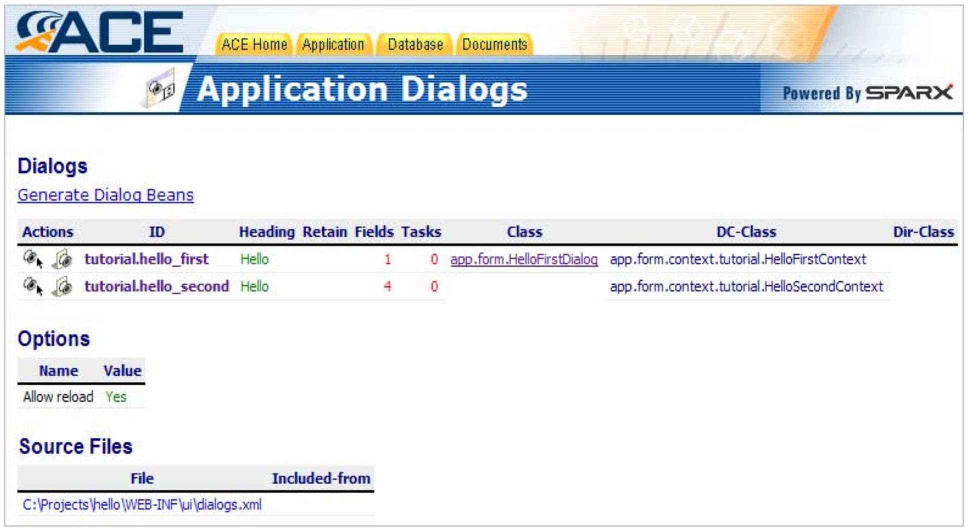
Then, to recompile the application, issue the following command

```
build
```

You should see quite a bit of output and a final message that tells you your build was successful. A successful build implies not only that your custom Java classes have been compiled but that any and all changes in your XML files have resulted in new Dialog Context classes and/or new Data Access Layer classes being created. Therefore it is recommended that you run build after any change to your application and before you test those changes out. It will ensure your application has all the generated and custom components it needs. Now you are ready to test the finished dialog.

Testing the Finished Dialog

All three components of our dialog are in place and we can test it out completely. First let us see how this dialog looks in ACE. Point your web browser to the URL http://host:port/hello/ace[14].



Dialogs

Generate Dialog Beans

Actions	ID	Heading	Retain	Fields	Tasks	Class	DC-Class	Dir-Class
	tutorial.hello_first	Hello		1	0	app.form.HelloFirstDialog	app.form.context.tutorial.HelloFirstContext	
	tutorial.hello_second	Hello		4	0		app.form.context.tutorial.HelloSecondContext	

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\Projects\hello\WEB-INF\ui\dialogs.xml	

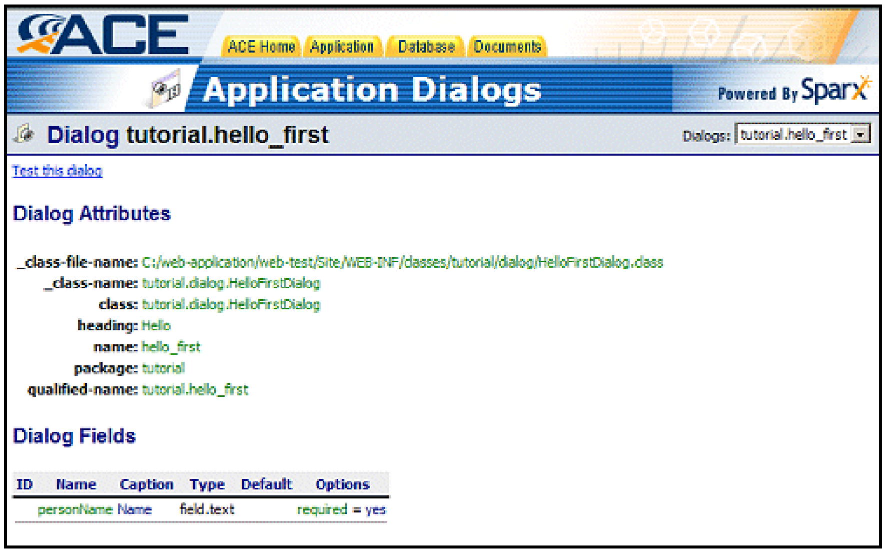
Figure 11: The Dialog List with the Added class.

Move your mouse over the *Application* menu item and choose *Dialogs* from the menu that drops down. If everything went smoothly you should see a change in the table entry for our `tutorial.hello_first` dialog: it now has an entry under the Class column which is `app.form.HelloFirstDialog`. We know now, therefore, that the

[13] By default, this is C:\Netspective\resin-x.y.z\webapps\hello\WEB-INF

[14] If you are evaluating Sparx online, you can access the Hello World ACE at http://developer.netspective.com/samples/hello/ace

binding was successful and that our Java dialog handler will indeed handle all data input using the `hello_first` dialog.



Dialog tutorial.hello_first

Test this dialog

Dialog Attributes

```
_class-file-name: C:/web-application/web-test/Site/WEB-INF/classes/tutorial/dialog/HelloFirstDialog.class
     _class-name: tutorial.dialog.HelloFirstDialog
           class: tutorial.dialog.HelloFirstDialog
         heading: Hello
            name: hello_first
         package: tutorial
  qualified-name: tutorial.hello_first
```

Dialog Fields

ID	Name	Caption	Type	Default	Options
	personName	Name	field.text		required = yes

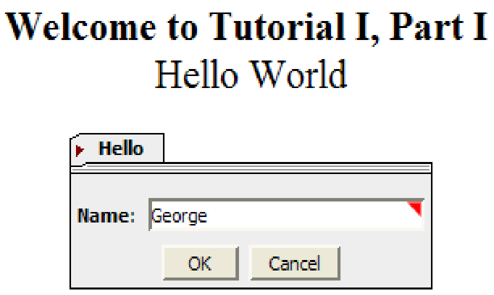
Figure 12: Field List with the Added Class

Click on the name of our dialog to see more details about it. You will notice that now there is additional information about the Java dialog handler class listed in this view. Go ahead and click on the "Test this dialog" link and it should open a new window with the dialog ready to be tested in it.

You may verify that client side field validation is still working by leaving the Name field empty and clicking on the OK button. The real test, however, is when you enter your name in the dialog and click on the OK button. Instead of giving you the debugging information like last time, you should get a personalized greeting (as specified in the Java dialog handler class) and the same dialog rendered directly underneath that greeting. That finally verifies that not only is our Java dialog handler successfully binding to the XML dialog, but that it is also able to access the data input to the dialog using its dialog context and it is able to output a result back to us.

Now it is finally time for the field test. Point your browser to the URL http://host:port/hello/filename.jsp[15] (where filename.jsp is the name under which you saved the JSP file we embedded our dialog in). You should see the familiar Hello World banner on top of the page and the rendered dialog beneath it just like when we tested the embedding of a dialog into a JSP. You may verify the client side validation of the field if you wish. Go ahead and enter your name in the Name field and click OK.

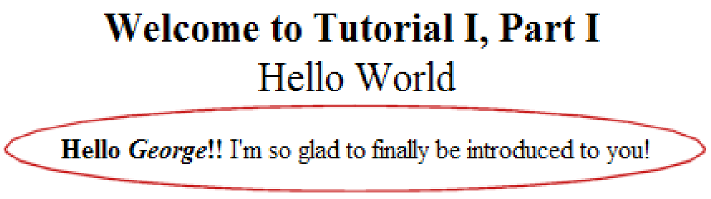
[15] If you are evaluating Sparx online, you can access the Hello World application at http://developer.netspective.com/samples/hello


Welcome to Tutorial I, Part I

Hello World

Figure 13: Sample of Hello World Tutorial I Part I Dialog

Congratulations! You should see your application greeting you by name; exactly what we sought to achieve.


Welcome to Tutorial I, Part I

Hello World

Hello *George*!! I'm so glad to finally be introduced to you!

Figure 14: Sample of Hello World Tutorial I Part I Dialog Execution

Another option

Please note that you can also place JSP commands and other HTML inside the `<sparx:dialog>` tag and that code will be executed once the dialog data has been submitted. Thus, another way of accomplishing the same output without any extra Java classes would be to use the following 3 lines of JSP.

```
<sparx:dialog name="tutorial.hello_first">
       Hello <%= dialogContext.getValue("personName") %>
</sparx:dialog>
```

Success

With a total of 9 lines to define an XML dialog, 22 lines to create a Java dialog handler class (which could be eliminated for simple cases by using JSP as shown above) and a handful of command-line issued commands, we have created an application that is interactive, has complete client-side and server-side validation of input and can process data input to it.

CHAPTER 4

Sparx Tools

This chapter serves as a reference to the major tools that Sparx provides to accelerate the constructions, operation, and maintenance phases of your application development process.

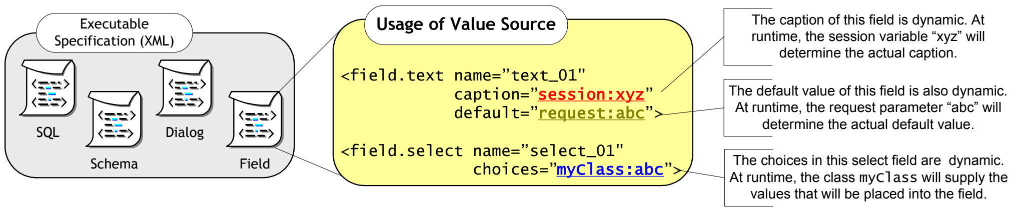
Value Sources

As you learned in Chapter 2, value sources are a simple Java interface along with numerous implementation classes that allows dynamic data to be included in XML without creating a programming language inside XML. As you have already seen (and will learn more about in subsequent chapters), there are many executable specifications locations where value sources are used. Some of them include:

- Configuration variables
- Forms/dialogs
- Form fields
- Form conditionals
- SQL statements
- SQL bind parameters

A complete reference to all value sources may be found at http://developer.netspective.com/xaf/value-sources.html.

Types of Value Sources



```
<field.text name="text_01"
            caption="session:xyz"
            default="request:abc">

<field.select name="select_01"
              choices="myClass:abc">
```

The caption of this field is dynamic. At runtime, the session variable “xyz” will determine the actual caption.

The default value of this field is also dynamic. At runtime, the request parameter “abc” will determine the actual default value.

The choices in this select field are dynamic. At runtime, the class `myClass` will supply the values that will be placed into the field.

Figure 15: Sample values sources in UI

Single Value Source (SVS)

In the figure shown above, the value sources used for the `caption` and `default` attributes are *single value sources*. A single value source (SVS) is an object that returns a single value from a particular source like a request parameter, text field, or session attribute. Many single value sources can double as list value sources (depending upon the context).

List Value Source (LVS)

In the figure shown above, the value sources used for the `choices` attribute is a *list value source*. A list value source (LVS) is an object that returns a list of values from a particular source (like an EJB or SQL query). Many list value sources can double as single value sources (depending upon the context).

Value Source Declaration Format

In general, value sources may appear in almost location of an executable specifications XML file. However, most value sources appear inside XML *attributes* as opposed to *tags*. For example, most value sources appear like this: `<some-tag attribute="value-source"/>`.

Static Values

The simplest value source is known as the *static* value source – which is simply a wrapper for the Java `String` object. So, if an attribute accepts a value source, you can always supply a static string and it will be wrapped inside a `StaticValue` class from the `*.sparx.util.value` package. For example, the following examples all produce the same results:

```
<field.text caption="ABC"/>
<field.text caption="static:ABC"/>
<field.text caption="string:ABC"/>
<field.text
       caption="com.netspective.sparx.util.value.StaticValue:ABC"/>
```

How Sparx Parses the name:params Declaration

The "name" portion of the value source refers to either a value source *identifier* (like "session" or "request") or the full name of a *class* that exists in your `classpath`. You can escape a value source by using the \ character in front of the `:` token. For example, if you'd like to create a string called "name:params" and not have it treated as a value source, use `name\:params`.

`name` : `params`

Given a string of the format `name:params`, Sparx performs the following steps:

	STEP
1	Using the "name" portion of the declaration, check the `ValueSourceFactory` class in the `com.netspective.sparx.utils.value` package to see if it's a built-in value source or has been registered separately by an application (all value sources are "first class" in the sense that it doesn't matter if it's a built-in source or one that you create and register yourself).
2	If the value source identifier is found in the `ValueSourceFactory` value sources dictionary (map), get the associated class name (an instance of `SingleValueSource` or `ListValueSource`). We will refer to this as the *Value Source Class*.
3	If the value source identifier is not found in `ValueSourceFactory`, check to see if "name" is a class name by using the Class.forName("name") method. Assuming a class is found, we will refer to this as the *Value Source Class*.

	STEP
4	Using the Value Source Class found from either step 2 or step 3, check to see if there is already an instance created for the class. If there is an existing instance, use that instance. If there is no existing instance, instantiate a new object by constructing the Value Source Class and calling its `initializeSource()` method and passing in the "params" portion of the value source declaration and cache the object for future use.

Common Value Sources

For a more exhaustive list, please visit http://developer.netspective.com/xaf/value-sources.html.

NAME	DESCRIPTION
config	Provides access to configuration variables in the default configuration file (`WEB-INF/conf/sparx.xml`).
create-app-url	Generates a URL based on the current application by automatically prepending the default servlet URL to the expression provided.
create-data-cmd-heading	Returns the current dialog `data_cmd` identifier plus the text provided that would be suitable for use as the heading of a multi-purpose dialog (a dialog that can be used for adding, updating, and deleting). For example, if `Person` is the text, and the current dialog's data_cmd is `add` then this SVS would return `Add Person`.
custom-sql	This SVS is used in SQL DML tasks and custom tags when, instead of a java value or expression, you want the DML processing to take an actual SQL expression that should be evaluated in the database. For example, if `sysdate` is the sql-expr, then this SVS would return `sysdate` without evaluating in Java or treating it as a Java string. If this value source is used in any context other than a DML (SQL insert or update or delete) then it returns just the sql-expr itself.
filesystem-entries	Provides list of files contained in a directory (either all files or by filter).
form	Provides access to a specific field of a dialog.
form-or-request	Provides access to a specific field of a dialog. If the field-name refers to a dialog field whose value is null, then this value source will return the value of a request parameter named field-name.
form-or-request-attr	Provides access to a specific field of a dialog. If the field-name refers to a dialog field whose value is null, then this value source will return the value of a request attribute named field-name.
generate-id	Returns a unique value each time the value source is called. The unique value is computed as a MD5 message digest hash based on the md5-seed provided with current date/time appended.
query	Executes a query and returns the results of the query as rows.
query-cols	Executes a query and returns the results of the query as columns.
request	Provides access to HTTP Servlet request parameters.
request-attr	Provides access to ServletRequest attributes; intelligently handles object of types String, String[], List, and Map.

NAME	DESCRIPTION
request-param	Provides access to HTTP Servlet request parameters. All parameter values are returned as String objects.
session	Provides access to HTTP session attributes. Intelligently handles object of types String, String[], and List.
simple-expr	Allows a string to be treated as a simple expression containing *other* value sources.
system-property	Provides access to the system property indicated by the specified key.

Available Value Source Classes

All registered value sources are shown in ACE's *Application* Menu, *Factories* submenu, *Value Sources* item. The following table is a list of most of the value source classes provided with Sparx – they serve as good examples when you want to create your own.

NAME	CLASS
config	`com.netspective.sparx.util.value.ConfigurationValue`
create-app-url	`com.netspective.sparx.util.value.ServletContextUriValue`
create-data-cmd-heading	`com.netspective.sparx.util.value.DialogDataCmdExprValue`
data-source-entries	`com.netspective.sparx.util.value.DataSourceEntriesListValue`
dialog-field-types	`com.netspective.sparx.util.value.DialogFieldFactoryListValue`
dialogs	`com.netspective.sparx.util.value.DialogsListValue`
filesystem-entries	`com.netspective.sparx.util.value.FilesystemEntriesListValue`
form	`com.netspective.sparx.util.value.DialogFieldValue`
formOrRequest	`com.netspective.sparx.util.value.DialogFieldOrRequestParameterValue`
formOrRequestAttr	`com.netspective.sparx.util.value.DialogFieldOrRequestAttributeValue`
generate-id	`com.netspective.sparx.util.value.GenerateIdValue`
query	`com.netspective.sparx.util.value.QueryResultsListValue`
query-cols	`com.netspective.sparx.util.value.QueryColumnsListValue`
query-defn-fields	`com.netspective.sparx.util.value.QueryDefnFieldsListValue`
query-defn-selects	`com.netspective.sparx.util.value.QueryDefnSelectsListValue`
request	`com.netspective.sparx.util.value.RequestParameterValue`
request-attr	`com.netspective.sparx.util.value.RequestAttributeValue`
request-param	`com.netspective.sparx.util.value.RequestParameterValue`
schema-enum	`com.netspective.sparx.util.value.SchemaDocEnumDataListValue`
schema-tables	`com.netspective.sparx.util.value.SchemaDocTablesListValue`
servlet-context-init-param	`com.netspective.sparx.util.value.ServletContextInitParamValue`
servlet-context-path	`com.netspective.sparx.util.value.ServletContextPathValue`
session	`com.netspective.sparx.util.value.SessionAttributeValue`
simple-expr	`com.netspective.sparx.util.value.ConfigurationExprValue`
static	`com.netspective.sparx.util.value.StaticValue`

NAME	CLASS
string	`com.netspective.sparx.util.value.StaticValue`
strings	`com.netspective.sparx.util.value.StringsListValue`
system-property	`com.netspective.sparx.util.value.SystemPropertyValue`

Ant Build Scripts

Sparx provides a series of scripts that, among other functions, allow you to compile your application, generate code, and upgrade Sparx files. The build scripts automatically read the exact same `WEB-INF/conf/sparx.xml` and `WEB-INF/conf/app-config.xml` files to create a simulated Servlet environment at the command line for compilation and code generation purposes.

FILE	PURPOSE
`WEB-INF/build.xml`	This is the primary Ant build script that actually performs the work outlined in the following section. This file is identical to `SPARX_HOME/tools/app-build.xml` and in fact is a copy of that file if you use the `new-sparx-app` script to create the Sparx application skeleton.
`WEB-INF/build.bat`	This script is the launcher for the `build.xml` file on Windows (using the MS-DOS prompt). This file is identical to `SPARX_HOME/tools/app-build.bat` and in fact is a copy of that file if you use the `new-sparx-app` script to create the Sparx application skeleton.
`WEB-INF/build.sh`	This script is the launcher for the `build.xml` file on UNIX or Linux. This file is identical to `SPARX_HOME/tools/app-build.sh` and in fact is a copy of that file if you use the `new-sparx-app` script to create the Sparx application skeleton.
`SPARX_HOME/tools/new-sparx-app`	This script is used to create a "starter" Sparx application. It takes a directory name as a parameter and copies bootstrap files (like the sparx.jar and XML libraries) to the new directory and then runs the build script with the `start-sparx-app` target.

Table 8: Sparx Build Scripts

Build Script Prerequisites

The Sparx build scripts (`build.bat` for Windows and `build.sh` for UNIX/Linux) assume that the following environment variables are already set before launching the scripts.

VARIABLE	DEFAULT VALUE
`JAVA_HOME`	The location in which you installed your Java Developer Kit (JDK). For example, if you installed the JDK in C:\JDK1.3.1_04 then you need to set your environment variable using a command like "`set JAVA_HOME=C:\JDK1.3.1_04`" before launching the build scripts. Although it's not required, we recommend that you add the `JAVA_HOME\bin` directory into your `PATH` as well.

VARIABLE	DEFAULT VALUE
SPARX_HOME	The location in which you installed the Sparx Developer Kit (SDK). For example, if you're using the evaluation kit this would need to be specified using a command like "`set SPARX_HOME=C:\Netspective\sparx-x.y.z`" where x.y.z is the appropriate version of Sparx you'd like to point to.

Table 9: Sparx Build Scripts Required Environment Variables

Build targets

The Sparx build file contains multiple *targets*. A *target* is a set of tasks you want to be executed. When starting the build script, you can select which target(s) you want to have executed. When no target is given, the project's default target of `all` is used. The following table lists all the targets and the tasks they perform.

TARGET	PURPOSE
`all`	Run all of the targets in this order: `clean`, `dal`, `dcb`, `identifiers`, `compile`, `dal-doc`, `dcb-doc`, `identifiers-doc`.
`all-but-docs`	Run all of the targets that are run for '`all`' except for the JavaDoc generators. This target may be almost twice as fast as the '`all`' target.
`clean`	Clean directories of derived files like `.class` and the log directory.
`clean-log`	Clean only the log directory (removes all entries and re-creates the directory).
`start-sparx-app`	In an empty directory, start a new sparx app (runs `setup-sparx-structure`, `copy-sparx-libs`, `copy-sparx-resources`).
`upgrade-sparx`	Upgrades Sparx libraries and resources in this application to the latest version available in SPARX_HOME (basically runs these two targets: `copy-sparx-libs` and `copy-sparx-resources`).
`compile`	Compile all the classes in the `WEB-INF/classes` directory.
`dal`	Generate the Data Access Layer (DAL) from the SchemaDoc. This target generates the classes and compiles them into `WEB-INF/`*`app-name`*`-dal.jar`.
`dal-doc`	Generate the Data Access Layer (DAL) API Documentation. This target also automatically runs the `dal` target.
`dcb`	Generate the DCBs (Dialog Context Beans) from Dialogs XML resources (`WEB-INF/ui/dialogs.xml`). Please see *Form Beans: Dialog Context Beans (DCBs)* on page 61 for more information. This target generates the classes and compiles them into `WEB-INF/`*`app-name`*`-dcb.jar`.
`dcb-doc`	Generate the Dialog Context Beans API Documentation. This target also automatically runs the `dcb` target.
`dialogs-ids`	Generate classes that represent Dialog/Form ids as Java constants.
`statements-ids`	Generate classes that represent the SQL Statement ids as Java constants.
`acl-ids`	Generate classes that represent the Access Control List permissions and roles as Java constants.

TARGET	PURPOSE
`identifiers`	Generate all the Identifiers Classes that provide Java constants for XML identifiers like Dialog Names, Statement IDs, and config value names. This is equivalent to running the targets `dialogs-ids`, `statements-ids`, and `acl-ids`. This target generates the classes and compiles them into `WEB-INF/`*`app-name-`*`ids.jar`.
`identifiers-doc`	Generate the Identifiers API Documentation. This target also automatically runs the `identifiers` target.
`copy-sparx-libs`	Copy (or upgrade) all the applicable Sparx libraries (JAR files) into the application's `WEB-INF/lib` directory.
`copy-sparx-resources`	Copy all the web-based resources needed by Sparx (`SPARX_HOME/web-shared` to `APP_ROOT/sparx`) plus any files (like `WEB-INF/tld/sparx.tld`) that are maintained only in Sparx distribution.
`copy-sparx-templates`	Copy all starter files for a Sparx application (if you run this on an existing application YOUR files will be overwritten by the original Sparx templates from `SPARX_HOME`).

Table 10: Sparx Build Targets

Using the Build scripts

As you have seen above, the Sparx build files allow a variety of actions but some of the more common uses are described in this section.

Performing a "complete" build of your Sparx-based application

The first type of common build is the "complete" build of the application which will clean (remove) any existing `.class` files in `WEB-INF/classes` and proceed to generate the Dialog Context Beans (DCBs), generate the Data Access Layer (DAL), Java Identifiers classes, compile all the classes in `WEB-INF/classes` and then generate the JavaDoc API documentation for the DCBs and DAL.

```
cd APP_ROOT/WEB-INF

build
```

The example above shows how to run the "all" target. Of course, APP_ROOT would be replaced name of the root directory of your application.

Compiling only your application's Java files

```
cd APP_ROOT/WEB-INF

build compile
```

This example above shows how to run only the "compile" target. This would just recompile all the classes in the `WEB-INF/classes` directory without removing any files or generating additional Sparx code.

Creating a "starter" Sparx application

To create a new, empty, Sparx application go to the `SPARX_HOME/tools` directory and run the `new-sparx-app` script with a parameter that specifies the new directory you would like to create and fill with a "starter" Sparx application.

```
cd SPARX_HOME\tools

new-sparx-app C:\your-new-app-directory-name
```

Upgrading to a New Release of Sparx

As Sparx is periodically updated (when new versions appear), upgrading an application's usage of a particular Sparx version is quite easy. There's a special build target called `upgrade-sparx` that facilitates an automatic upgrade.

```
cd APP_ROOT/WEB-INF

build upgrade-sparx
```

Please pay special attention to the output of the build – depending upon what's been upgraded in Sparx you may get additional messages that require you to perform other steps.

When `SPARX_HOME/tools/app-build.bat` or `app-build.sh` change

If either `SPARX_HOME/tools/app-build.bat` or `app-build.sh` is updated, you will need to manually copy them to your `WEB-INF/build.bat` or `WEB-INF/build.sh` files (the `upgrade-sparx` target will alert you to any changes in either file).

When `SPARX_HOME/app-build.xml` changes

If the `app-build.xml` file changes, you will need to run the `upgrade-sparx` target **twice**. The first time you run the build, it will actually copy the new `app-build.xml` to `WEB-INF/build.xml`. The second time you run the `build`, it will actually perform any upgrades necessary.

Application Components Explorer (ACE)

As you have already learned, the Sparx Application Components Explorer (ACE) is a Servlet that provides a browser-based administrative interface to all of the dynamic components and objects that Sparx generates. ACE is automatically available to all Sparx-based applications during development and can be optionally available in production.

Application Menu Overview

MENU OPTION	DESCRIPTION
Dialogs	Displays a list of all dialogs defined in external dialogs resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/ui/dialogs.xml` (and any files included by that file).
Configuration	Displays a list of all of the configuration properties present in `WEB-INF/conf/sparx.xml` and `WEB-INF/conf/app-config.xml`.
Servlet Context	Displays the execution environment (development, testing, or production), the `classpath` and classloader, the JAXP and TRaX libraries in use, and Servlet `init parameters`.
Access Control	Displays the list of permissions and roles available to the application. Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/security/access-control.xml` (and any files included by that file).
System Properties	Displays a simple list of all system properties (available from `System.getProperties()` method).
Metrics	Displays overall metrics for the entire application. This includes the total packages and dialogs for the UI, the number of SQL statements, code files, application files, and numerous other statistics.
Factories	Displays a list of submenus that provide the ability to review all the Sparx object factories, their contents, and classes they represent.
Logs	Displays a list of submenus that provide the ability to review all the log files managed by Sparx.

Table 11: Sparx ACE Application Menu Overview

Database Menu Overview

MENU OPTION	DESCRIPTION
SQL Statements	Displays a list of all the static SQL statements defined in external SQL resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/sql/statements.xml` (and any files included by that file).
SQL Query Definitions	Displays a list of all the dynamic SQL query definitions defined in external SQL resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/sql/statements.xml` (and any files included by that file).
SchemaDoc (XML)	Displays the database documentation represented by an external SchemaDoc (XML resource file). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/schema/schema.xml` and all files included by that file.
Generate SQL DDL	Provides a form that allows you to use an existing SchemaDoc and generate database-specific SQL data definition language (DDL) for the schema.
Generate Java DAL	Provides a form that allows you to use an existing SchemaDoc and generate a database-independent Java Data Access Layer (DAL or object-relational map). This ACE menu option provides the same capability as the `dal` target of the Sparx build script (please see *Ant Build Scripts* on page 37 for more information on that target).

MENU OPTION	DESCRIPTION
Data Sources	Displays the list of data sources defined by your application. This is a good way to unit test whether or not a given datasource is being seen by Sparx and if Sparx can connect to the database defined by a data source.
Import DAL Data	Provides a form that allows you to specify an XML data file that can be used to import data into a database that has a Java Data Access Layer (DAL).
Reverse Engineer	Provides a form that allows you to generate a SchemaDoc file based on an existing database that is accessible through a JDBC driver.

Table 12: Sparx ACE Database Menu Overview

Documents Menu Overview

MENU OPTION	DESCRIPTION
Application	Provides a file browser that allows you to make all of your application's data and source code available through a browser.
Sparx	A link to http://developer.netspective.com.

Table 13: Sparx ACE Documents Menu Overview

Common Sections

There are two sections that appear on a number of pages in ACE: Options and Source Files and described here once to avoid repetition in the remainder of the document.

Options section

The options section displays a variety of options depending upon your application. The most important option here is usually the "Allow reload" option. The `Allow reload` option is usually `true` in a development environment, but `false` in testing or production environments (please see *Execution Environment* on page 56 for more information). If `Allow reload` is `true`, then the XML resource files are automatically reloaded if any changes are made. If it's `false`, the XML resource files are only loaded at the startup of the application (and the only way to reload changed files is to restart either the application or the app server).

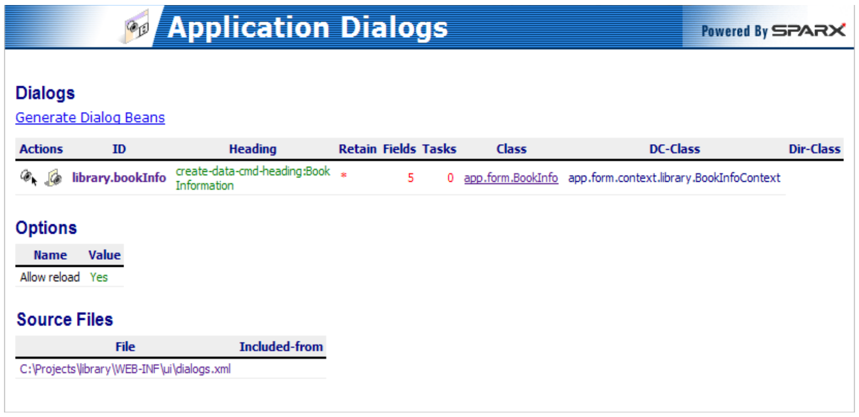
Source Files section

This section lists all of the files that were read (as well as included files). Clicking on the name of a file in this list will show the contents of the file with syntax-highlighting.

ACE Application Menu Items

Application Dialogs Page

The Application Dialogs page (which is displayed when you choose the *Dialogs* menu item from the *Application* menu) contains the three sections: Dialogs, Options, and Source Files.



Dialogs

Generate Dialog Beans

Actions	ID	Heading	Retain	Fields	Tasks	Class	DC-Class	Dir-Class
	library.bookInfo	create-data-cmd-heading:Book Information	*	5	0	app.form.BookInfo	app.form.context.library.BookInfoContext	

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\Projects\library\WEB-INF\ui\dialogs.xml	

Figure 16: Sparx ACE Application Dialogs Page

Dialogs section

This section is a list of all dialogs defined in external dialogs resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/ui/dialogs.xml` (and any files included by that file). The table contains the following columns.

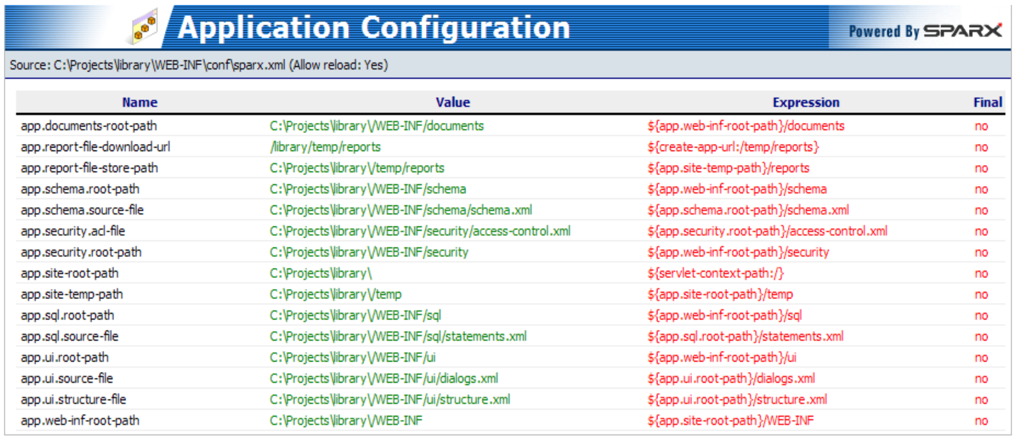
ENTRY	DESCRIPTION
Actions	The first icon (with the pointer) will allow you to *unit test* the dialog/form. This testing mode allows a developer to see how the dialog will render as well as to ensure that all dialog parameters are working as intended. This is a very good way to unit test a dialog and is the method we will be using to test the dialog we just created.
ID	The complete package and name of the form. Clicking this link will show the functional specification of the form (including a complete field listing with per field options)
Heading	The heading of the dialog as it will appear on the "title bar" of the dialog when displayed on a web page.
Retain	The status of the "retain" option. This option determines how any URL encoded parameters to a dialog are treated. These URL encoded parameters, which appear as strings such as ?param1=one&param2=two at the end of a URL, can be interpreted by a Java dialog to further fine tune a dialog's handling of the data that is input. When the retain option is set to "yes" for a dialog, it implies that Sparx will keep these URL encoded parameters at all stages of that dialog's execution. This would enable the Java dialog to read them and take appropriate action if necessary.
Fields	The number of fields that are a part of the dialog.

ENTRY	DESCRIPTION
Tasks	The number of tasks that are a part of the dialog. Tasks are XML definitions of common processing that a dialog can do based on the data input by the user. Tasks include XML defined database insertions, deletions and updates as well. When you use tasks in XML Dialogs, they eliminate the need for custom Java processing for common tasks like selecting data from a database and inserting/updating data in a database.
Class	The custom Java dialog class that will handle the dialog's XML definition. The default Java dialog that interprets, renders and processes all XML dialogs is the `com.netspective.sparx.xaf.form.Dialog` class that is a part of the Sparx binary. However, to process data input through anything more than the simplest of XML dialogs, developers have to create custom Java dialog classes (that inherit from the default Sparx dialog class) to interpret, render and process that data. If such a custom class exists for the dialog, it will be shown here. If, however, this dialog uses the default Sparx dialog class, then this space will be empty. When you hover over the class name with your mouse, a small window will appear showing the complete path of the class and where it is located. Clicking on the name of the class will show the source code for the class.
DC-Class	The Java class that manages the form's data. This is a tightly typed Java representation of each of the dialog's XML fields (called a Dialog Context Bean or DCB) that allows seamless programmatic access to the dialog data and options. Similar to the default Java dialog, there is a default Java dialog context (`com.netspective.sparx.xaf.form.DialogContext`) which is created after Sparx interprets the XML definition of the dialog. However, developers might be interested in creating a custom Java dialog context which, if it exists, is specified here. If the dialog is not associated with a custom dialog context, this space will be empty. When you hover over the class name with your mouse, a small window will appear showing the complete path of the class and where it is located.
Dir-Class	The custom Java dialog director class. Sparx comes with a default Java dialog director class which provides the `OK` and `Cancel` buttons at the bottom of each dialog that is rendered using this class. If an application requires these buttons to change in functionality, appearance or any other aspect, the developer will need to create a custom Java dialog director class. If such a custom Java dialog director class is associated with the dialog, it will be listed here. Otherwise, this space will be empty.

Table 14: Sparx ACE Application Dialogs Page Details

Application Configuration Page

The Application Configuration page (which is displayed when you choose the *Configuration* menu item from the *Application* menu) contains a list of all of the configuration properties present in `WEB-INF/conf/sparx.xml` and `WEB-INF/conf/app-config.xml`. It contains four separate columns that are shown and described below.


Application Configuration

Powered By SPARX

Source: C:\Projects\library\WEB-INF\conf\sparx.xml (Allow reload: Yes)

Name	Value	Expression	Final
app.documents-root-path	C:\Projects\library\WEB-INF/documents	${app.web-inf-root-path}/documents	no
app.report-file-download-url	/library/temp/reports	${create-app-url:/temp/reports}	no
app.report-file-store-path	C:\Projects\library\/temp/reports	${app.site-temp-path}/reports	no
app.schema.root-path	C:\Projects\library\/WEB-INF/schema	${app.web-inf-root-path}/schema	no
app.schema.source-file	C:\Projects\library\/WEB-INF/schema/schema.xml	${app.schema.root-path}/schema.xml	no
app.security.acl-file	C:\Projects\library\/WEB-INF/security/access-control.xml	${app.security.root-path}/access-control.xml	no
app.security.root-path	C:\Projects\library\/WEB-INF/security	${app.web-inf-root-path}/security	no
app.site-root-path	C:\Projects\library\	${servlet-context-path:/}	no
app.site-temp-path	C:\Projects\library\/temp	${app.site-root-path}/temp	no
app.sql.root-path	C:\Projects\library\/WEB-INF/sql	${app.web-inf-root-path}/sql	no
app.sql.source-file	C:\Projects\library\/WEB-INF/sql/statements.xml	${app.sql.root-path}/statements.xml	no
app.ui.root-path	C:\Projects\library\/WEB-INF/ui	${app.web-inf-root-path}/ui	no
app.ui.source-file	C:\Projects\library\/WEB-INF/ui/dialogs.xml	${app.ui.root-path}/dialogs.xml	no
app.ui.structure-file	C:\Projects\library\/WEB-INF/ui/structure.xml	${app.ui.root-path}/structure.xml	no
app.web-inf-root-path	C:\Projects\library\/WEB-INF	${app.site-root-path}/WEB-INF	no

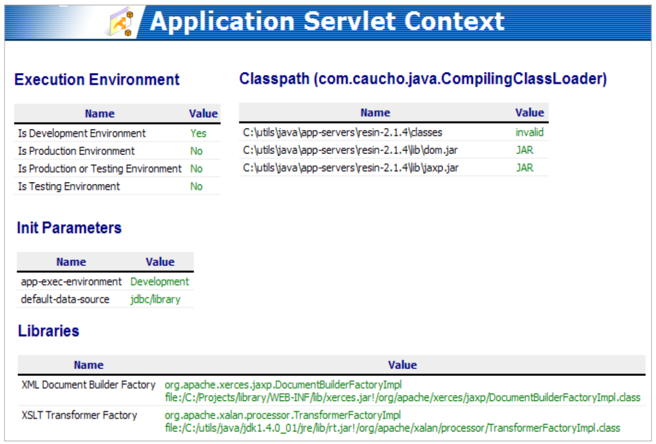
Figure 17: Sparx ACE Application Configuration Page

ENTRY	DESCRIPTION
Name	The name of the property.
Value	The unit test of the value of the property. This is the *evaluated* value of the property.
Expression	The actual specification of the property in the original file (before it is evaluated).
Final	If this column is blank (or "yes") then it means that the value of the property is evaluated once and cached (which means it doesn't change through the execution of the program). If the value of a property is dependent upon runtime variables (like request parameters, Servlet context paths, etc) then *final* will be set to "no" to indicate that the value *may* change each time it is evaluated.

Table 15: Sparx ACE Application Configuration Page Details

Application Servlet Context Page

The Application Servlet Context page (which is displayed when you choose the *Servlet Context* menu item from the *Application* menu) displays the execution environment (please see *Execution Environment* on page 56 for more information), the `classpath` and classloader, the JAXP and TRaX libraries in use, and Servlet `init parameters`.


Application Servlet Context

Execution Environment

Name	Value
Is Development Environment	Yes
Is Production Environment	No
Is Production or Testing Environment	No
Is Testing Environment	No

Classpath (com.caucho.java.CompilingClassLoader)

Name	Value
C:\utils\java\app-servers\resin-2.1.4\classes	invalid
C:\utils\java\app-servers\resin-2.1.4\lib\dom.jar	JAR
C:\utils\java\app-servers\resin-2.1.4\lib\jaxp.jar	JAR

Init Parameters

Name	Value
app-exec-environment	Development
default-data-source	jdbc/library

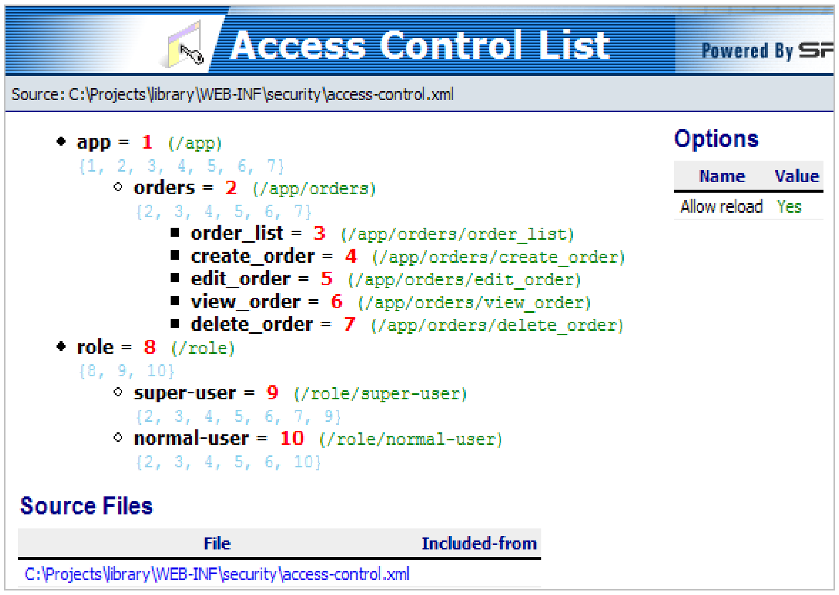
Libraries

Name	Value
XML Document Builder Factory	org.apache.xerces.jaxp.DocumentBuilderFactoryImpl file:/C:/Projects/library/WEB-INF/lib/xerces.jar!/org/apache/xerces/jaxp/DocumentBuilderFactoryImpl.class
XSLT Transformer Factory	org.apache.xalan.processor.TransformerFactoryImpl file:/C:/utils/java/jdk1.4.0_01/jre/lib/rt.jar!/org/apache/xalan/processor/TransformerFactoryImpl.class

Figure 18: Sparx ACE Application Servlet Context Page

Application Access Control Page

The Application Access Control page (which is displayed when you choose the *Access Control* menu item from the *Application* menu) displays the list of permissions and roles available to the application. Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/security/access-control.xml` (and any files included by that file). This page contains three sections: the permissions, Options, and Source Files.



Source: C:\Projects\library\WEB-INF\security\access-control.xml

```
◆ app = 1 (/app)
  {1, 2, 3, 4, 5, 6, 7}
    ◇ orders = 2 (/app/orders)
      {2, 3, 4, 5, 6, 7}
        ■ order_list = 3 (/app/orders/order_list)
        ■ create_order = 4 (/app/orders/create_order)
        ■ edit_order = 5 (/app/orders/edit_order)
        ■ view_order = 6 (/app/orders/view_order)
        ■ delete_order = 7 (/app/orders/delete_order)
◆ role = 8 (/role)
  {8, 9, 10}
    ◇ super-user = 9 (/role/super-user)
      {2, 3, 4, 5, 6, 7, 9}
    ◇ normal-user = 10 (/role/normal-user)
      {2, 3, 4, 5, 6, 10}
```

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\Projects\library\WEB-INF\security\access-control.xml	

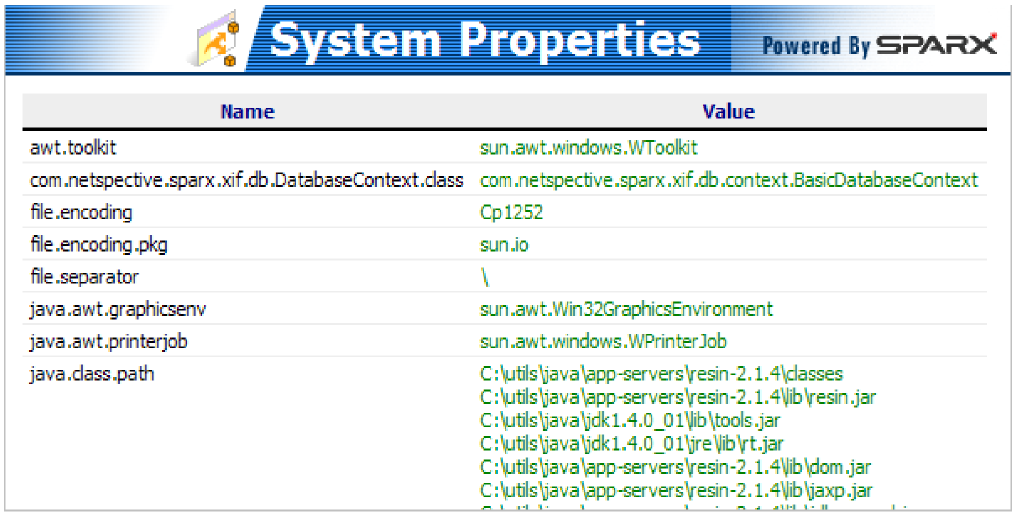
Figure 19: Sparx ACE Application Access Control List Page

Permissions section

This section shows a hierarchical list of permissions and roles defined by the application the `WEB-INF/security/access-control.xml` file. It doubles as both a unit test to ensure that all the persmissions were read and defined properly and as documentation for the full permission paths (like `/app/order/order_list`). Application security in general and permissions and roles in particular are described in detail on http://developer.netspective.com.

Application System Properties Page

The Application System Properties page (which is displayed when you choose the *System Properties* menu item from the *Application* menu) displays a simple list of all system properties (available from `System.getProperties()` method).

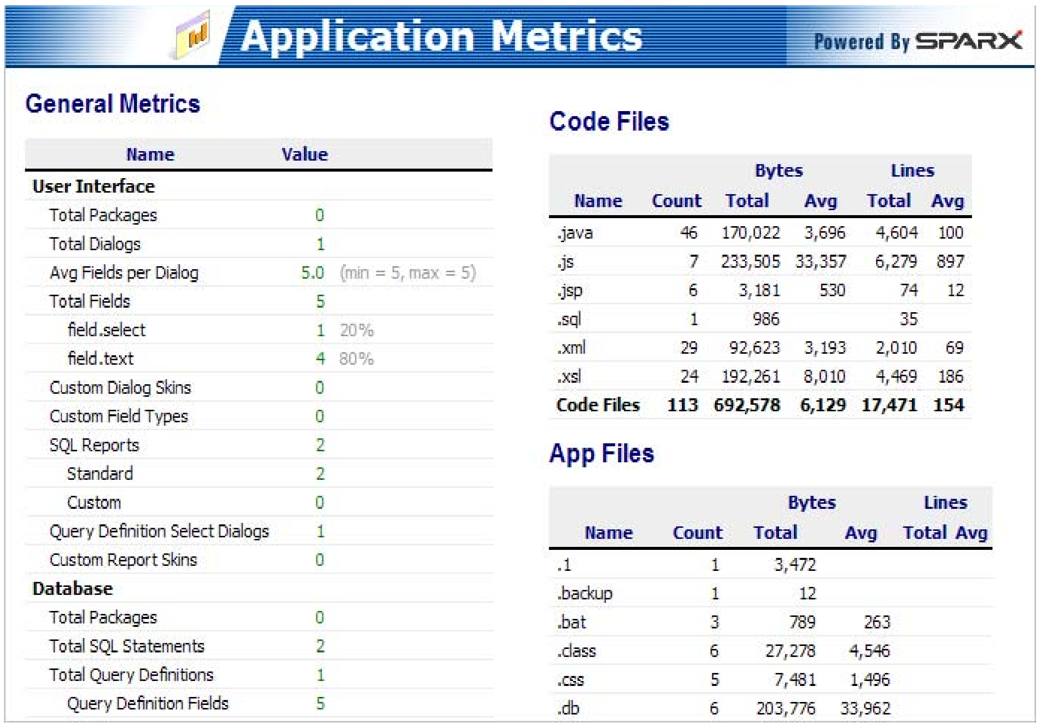


Name	Value
awt.toolkit	sun.awt.windows.WToolkit
com.netspective.sparx.xif.db.DatabaseContext.class	com.netspective.sparx.xif.db.context.BasicDatabaseContext
file.encoding	Cp1252
file.encoding.pkg	sun.io
file.separator	\
java.awt.graphicsenv	sun.awt.Win32GraphicsEnvironment
java.awt.printerjob	sun.awt.windows.WPrinterJob
java.class.path	C:\utils\java\app-servers\resin-2.1.4\classes C:\utils\java\app-servers\resin-2.1.4\lib\resin.jar C:\utils\java\jdk1.4.0_01\lib\tools.jar C:\utils\java\jdk1.4.0_01\jre\lib\rt.jar C:\utils\java\app-servers\resin-2.1.4\lib\dom.jar C:\utils\java\app-servers\resin-2.1.4\lib\jaxp.jar

Figure 20: Sparx ACE Application System Properties Page

Application Metrics Page

The Application Metrics page (which is displayed when you choose the *Metrics* menu item from the *Application* menu) displays overall metrics for the application. This includes the total packages and dialogs for the UI, the number of SQL statements, code files, application files, and numerous other statistics.



General Metrics

Name	Value	
User Interface		
Total Packages	0	
Total Dialogs	1	
Avg Fields per Dialog	5.0	(min = 5, max = 5)
Total Fields	5	
field.select	1	20%
field.text	4	80%
Custom Dialog Skins	0	
Custom Field Types	0	
SQL Reports	2	
Standard	2	
Custom	0	
Query Definition Select Dialogs	1	
Custom Report Skins	0	
Database		
Total Packages	0	
Total SQL Statements	2	
Total Query Definitions	1	
Query Definition Fields	5	

Code Files

		Bytes		Lines	
Name	Count	Total	Avg	Total	Avg
.java	46	170,022	3,696	4,604	100
.js	7	233,505	33,357	6,279	897
.jsp	6	3,181	530	74	12
.sql	1	986		35	
.xml	29	92,623	3,193	2,010	69
.xsl	24	192,261	8,010	4,469	186
Code Files	**113**	**692,578**	**6,129**	**17,471**	**154**

App Files

		Bytes		Lines	
Name	Count	Total	Avg	Total	Avg
.1	1	3,472			
.backup	1	12			
.bat	3	789	263		
.class	6	27,278	4,546		
.css	5	7,481	1,496		
.db	6	203,776	33,962		

Figure 21: Sparx ACE Application Metrics Page

Application Factories Pages

The Application Factories pages (which are displayed when you choose one of the *Factory* submenu item from the *Application* menu) provide the ability to review all the Sparx object factories, their contents, and classes they represent. The following menu items are available from the *Factory* menu.

MENU ITEM	FACTORY
Value Sources	The names, usage, and descriptions of all single and list value sources available to applications. This includes any custom value sources that may be registered by an application.
Dialog Fields	The names and associated class names of all dialog fields (`field.xxx` tags) and field conditionals. This includes any custom field types and conditionals that may be registered by an application.
Report Components	The names and classes of report column types, column formats, and column data calculators.
Tasks	The names and associated classes of tasks.
Skins	The names and associated classes of report and dialog skins.
SQL Comparisons	The identifiers and associated classes for each of the SQL Query Definition comparison types.

Table 16: Sparx ACE Application Factories Pages

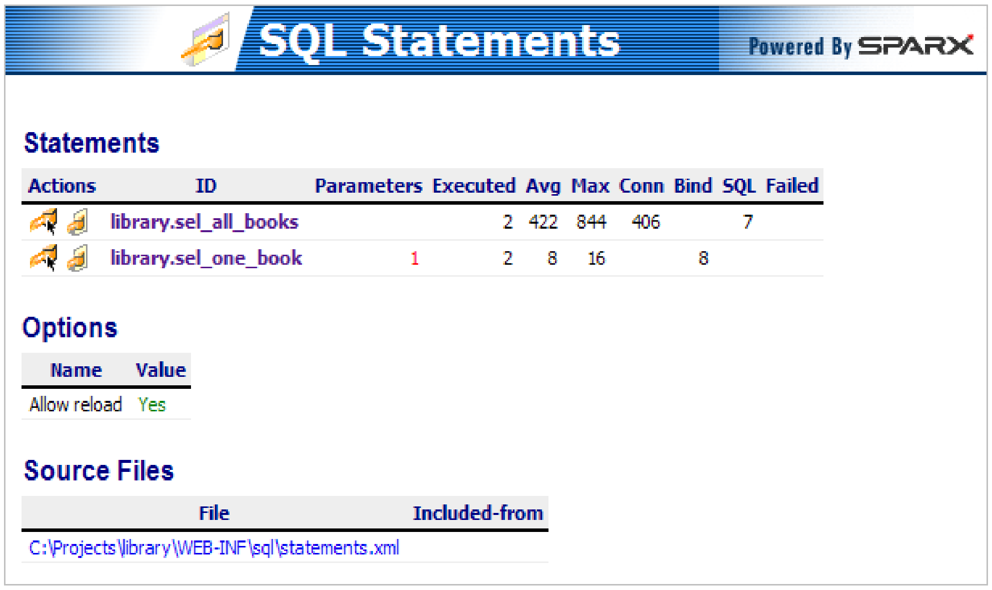
Application Logs Pages

The Application Logs pages (which are displayed when you choose one of the *Log* submenu item from the *Application* menu) provide the ability to review all the log files managed by Sparx. The actual location of the log files is controlled by `WEB-INF/classes/log4j.properties`. Please review the 🗀 *WEB-INF/log* section on page 7 for more information about the log files and their purpose.

ACE Database Menu Items

Database SQL Statements Page

The Database SQL Statements page (which is displayed when you choose the *SQL Statements* menu item from the *Database* menu) contains three sections: Statements, Options, and Source Files.



Statements

Actions	ID	Parameters	Executed	Avg	Max	Conn	Bind	SQL	Failed
	library.sel_all_books		2	422	844	406		7	
	library.sel_one_book	1	2	8	16		8		

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\Projects\library\WEB-INF\sql\statements.xml	

Figure 22: Sparx ACE Database SQL Statements Page

Statements section

This section is a list of all the static SQL statements defined in external SQL resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/sql/statements.xml` (and any files included by that file). Each of the statistics represented in the table are valid for a particular *session* (since the start of the app server or the initialization of the application, whichever comes later). If you'd like to review statistics across sessions, you need to take a look at the log file described by the `sparx.monitor.sql` category in `WEB-INF/classes/log4j.properties` file. The table contains the following columns.

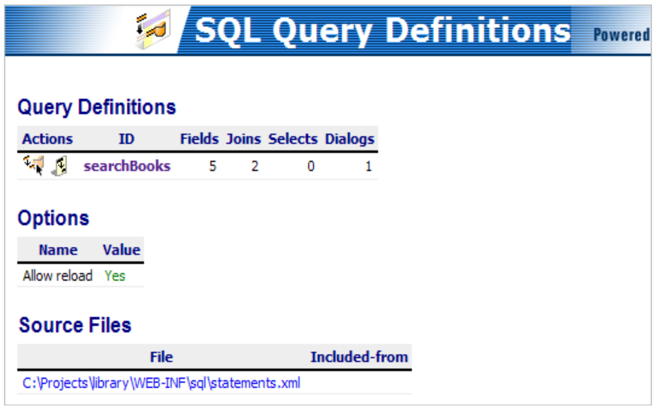
ENTRY	DESCRIPTION
Actions	The first icon (with the pointer) will allow you to *unit test* the statement.
ID	The complete package and name of the statement. Clicking this link will show the functional specification of the statement (including the SQL itself).
Parameters	The number of SQL bind parameters the SQL statement expects.
Executed	The number of times the SQL statement has been executed during this session (since the start of the app server or the initialization of the application, whichever comes later).
Avg	The average number of milliseconds it has taken to execute this statement during this session (including the connection, binding, and SQL run).
Max	The maximum number of milliseconds any particular invocation of this SQL statement has taken during this sessions.
Conn	The average number of milliseconds it has taken to make database connections for this statement during this session (excluding binding the parameters and the actual running of the SQL).
Bind	The average number of milliseconds it has taken to bind parameters for this statement during this session (excluding establishing the connection and the actual running of the SQL).

ENTRY	DESCRIPTION
SQL	The average number of milliseconds it has taken to run the SQL for this statement during this session (excluding establishing the connection and the binding of parameters).
Failed	The number of times the SQL statement has failed to run during the session.

Table 17: Sparx ACE Database SQL Statements Details

Database SQL Query Definitions Page

The Database SQL Query Definitions page (which is displayed when you choose the *SQL Query Definitions* menu item from the *Database* menu) contains three sections: Query Definitions, Options, and Source Files.



Query Definitions

Actions	ID	Fields	Joins	Selects	Dialogs
	searchBooks	5	2	0	1

Options

Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\Projects\library\WEB-INF\sql\statements.xml	

Figure 23: Sparx ACE Database SQL Query Definitions Page

Query Definitions section

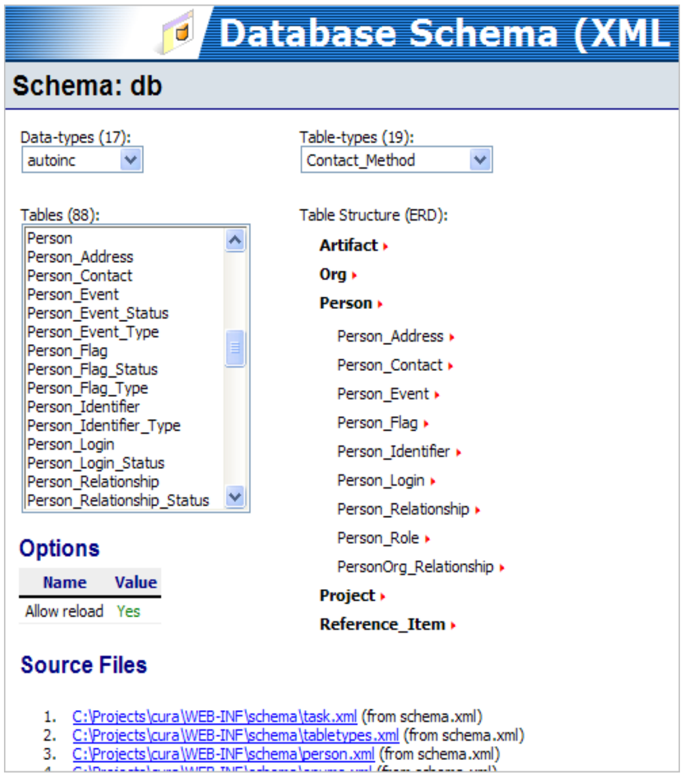
This section displays a list of all the dynamic SQL query definitions defined in external SQL resource files (XML). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/sql/statements.xml` (and any files included by that file. The table contains the following columns).

ENTRY	DESCRIPTION
Actions	The first icon (with the pointer) will allow you to *unit test* the query definition.
ID	The name of the query definition (query definitions are not placed into packages). Clicking on this click shows the functional specification of the query definition including all fields, joins, selects, and dialogs.
Fields	The number of fields defined by the query definition.
Joins	The number of joins defined by the query definition.
Selects	The number of selects (fixed SQL statements generated at runtime) defined by the query definition.
Dialogs	The number of special query-select-dialogs defined by the query definition. Each query-select-dialog is a special-purpose dialog that can accept input and automatically generate SQL, connection to a database, run the SQL, and display the results in a pageable view (or store the results in a file).

Table 18: Sparx ACE Database SQL Query Definitions Details

Database SchemaDoc Page

The Database SchemaDoc page (which is displayed when you choose the *SchemaDoc* menu item from the *Database* menu) contains 6 sections: Data-types, Table-types, Tables, Table structure (ERD), Options, and Source Files. This page displays the database documentation represented by an external SchemaDoc (XML resource file). Unless you've modified the location in `WEB-INF/conf/app-config.xml`, the default file that is read and displayed is `WEB-INF/schema/schema.xml` and all files included by that file.


Database Schema (XML

Schema: db

Data-types (17): autoinc

Table-types (19): Contact_Method

Tables (88):
Person
Person_Address
Person_Contact
Person_Event
Person_Event_Status
Person_Event_Type
Person_Flag
Person_Flag_Status
Person_Flag_Type
Person_Identifier
Person_Identifier_Type
Person_Login
Person_Login_Status
Person_Relationship
Person_Relationship_Status

Table Structure (ERD):
Artifact
Org
Person
Person_Address
Person_Contact
Person_Event
Person_Flag
Person_Identifier
Person_Login
Person_Relationship
Person_Role
PersonOrg_Relationship
Project
Reference_Item

Options

Name	Value
Allow reload	Yes

Source Files

1. C:\Projects\cura\WEB-INF\schema\task.xml (from schema.xml)
2. C:\Projects\cura\WEB-INF\schema\tabletypes.xml (from schema.xml)
3. C:\Projects\cura\WEB-INF\schema\person.xml (from schema.xml)

Figure 24: Sparx ACE Database SchemaDoc Page

Data-types section

Data-types are created to help maintain a RDBMS-neutral and consistent data dictionary. Data-types should be considered "column templates" that allow a programmer to define allowable column types. Data-types may be inherited from other data-types, allowing better reuse and object-orientation in relational databases.

Table-types section

Table-types are created to help define generic tables and behaviors that can be inherited by real tables. Table-types should be considered "table templates" or base entity objects. Table-types may be inherited from other table-types, allowing better reuse and object-orientation in relational databases.

Tables section

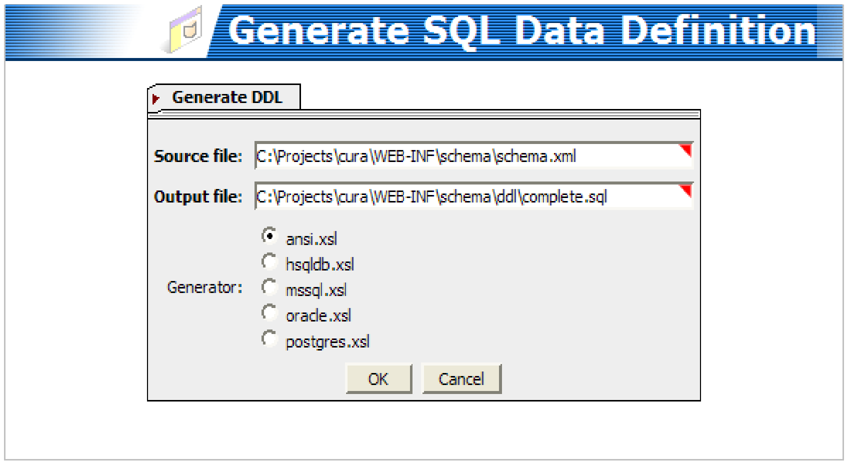
Tables are the actual data structures that will hold data in a relational database. This section is a simple alphabetical list of all tables defined in the SchemaDoc. Clicking on a table name takes you to the table definition document.

Table Structure (ERD) section

Tables can inherit and extend content from table-types and contain columns (which contain and extend content from data-types). This section is similar to the Tables section except this section shows the list of tables in a manner similar to a Entity-relationship Diagram (ERD).

Database Generate SQL DDL Page

The Database Generate SQL DDL page (which is displayed when you choose the *Generate SQL DDL menu* item from the *Database* menu) provides a form that allows you to use an existing SchemaDoc and generate database-specific SQL data definition language (DDL) for the schema.



Source file: C:\Projects\cura\WEB-INF\schema\schema.xml

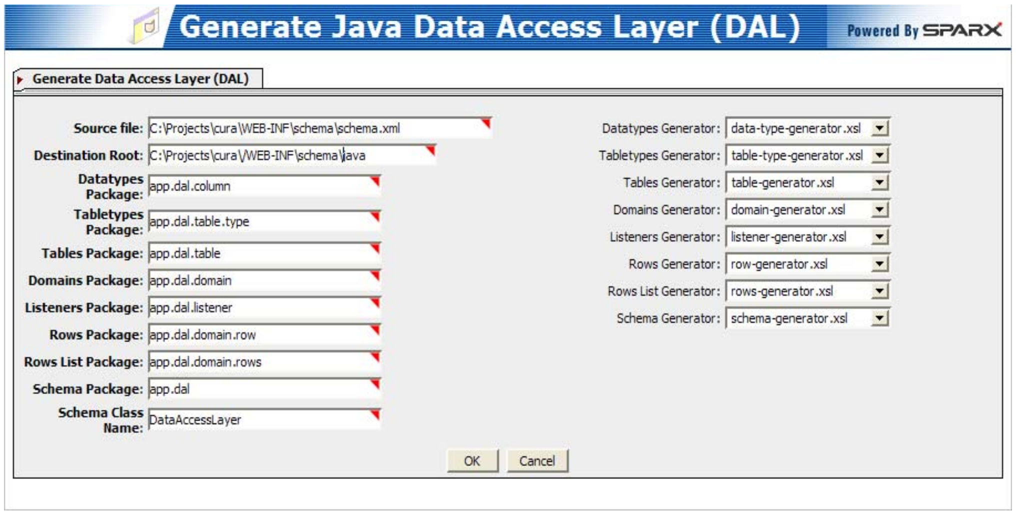
Output file: C:\Projects\cura\WEB-INF\schema\ddl\complete.sql

Generator: [✓ ansi.xsl] hsqldb.xsl mssql.xsl oracle.xsl postgres.xsl

Figure 25: Sparx Database Generate SQL DDL Page

Database Generate Java DAL Page

The Database Generate Java DAL page (which is displayed when you choose the *Generate Java DAL* menu item from the *Database* menu) provides a form that allows you to use an existing SchemaDoc and generate a database-independent Java Data Access Layer (DAL or object-relational map). This ACE menu option provides the same capability as the `dal` target of the Sparx build script (please see *Ant Build Scripts* on page 37 for more information on that target).



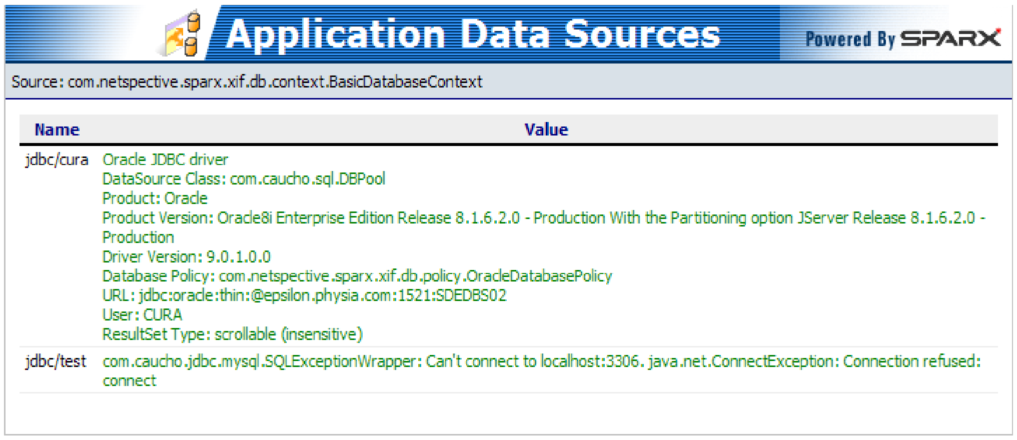
Generate Java Data Access Layer (DAL)
Powered By SPARX
Generate Data Access Layer (DAL)
OK
Cancel

Source file:	C:\Projects\cura\WEB-INF\schema\schema.xml	Datatypes Generator:	data-type-generator.xsl
Destination Root:	C:\Projects\cura\WEB-INF\schema\java	Tabletypes Generator:	table-type-generator.xsl
Datatypes Package:	app.dal.column	Tables Generator:	table-generator.xsl
Tabletypes Package:	app.dal.table.type	Domains Generator:	domain-generator.xsl
Tables Package:	app.dal.table	Listeners Generator:	listener-generator.xsl
Domains Package:	app.dal.domain	Rows Generator:	row-generator.xsl
Listeners Package:	app.dal.listener	Rows List Generator:	rows-generator.xsl
Rows Package:	app.dal.domain.row	Schema Generator:	schema-generator.xsl
Rows List Package:	app.dal.domain.rows		
Schema Package:	app.dal		
Schema Class Name:	DataAccessLayer		

Figure 26: Sparx Database Generate Java DAL Page

Database Data Sources Page

The Database Data Sources page (which is displayed when you choose the *Data Sources* menu item from the *Database* menu) displays the list of data sources defined by your application. This is a good way to unit test whether or not a given data source is being seen by Sparx and if Sparx can connect to the database defined by a data source.



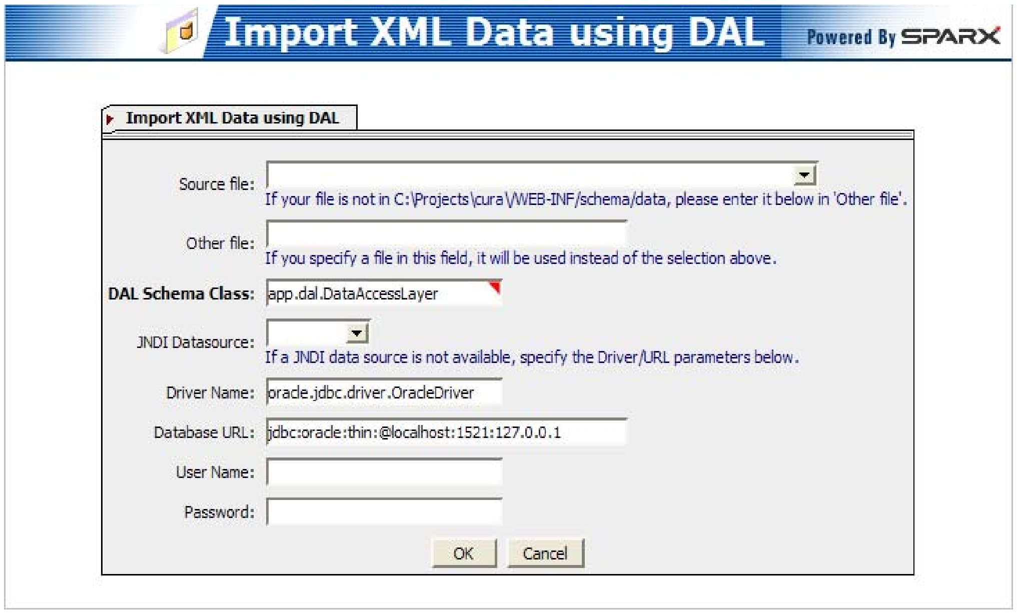
Source: com.netspective.sparx.xif.db.context.BasicDatabaseContext

Name	Value
jdbc/cura	Oracle JDBC driver DataSource Class: com.caucho.sql.DBPool Product: Oracle Product Version: Oracle8i Enterprise Edition Release 8.1.6.2.0 - Production With the Partitioning option JServer Release 8.1.6.2.0 - Production Driver Version: 9.0.1.0.0 Database Policy: com.netspective.sparx.xif.db.policy.OracleDatabasePolicy URL: jdbc:oracle:thin:@epsilon.physia.com:1521:SDEDBS02 User: CURA ResultSet Type: scrollable (insensitive)
jdbc/test	com.caucho.jdbc.mysql.SQLExceptionWrapper: Can't connect to localhost:3306. java.net.ConnectException: Connection refused: connect

Figure 27: Sparx Database Data Sources Page

Database Import DAL Data Page

The Database Import DAL Data page (which is displayed when you choose the *Import DAL Data* menu item from the *Database* menu) provides a form that allows you to specify an XML data file that can be used to import data into a database that has a Java Data Access Layer (DAL).

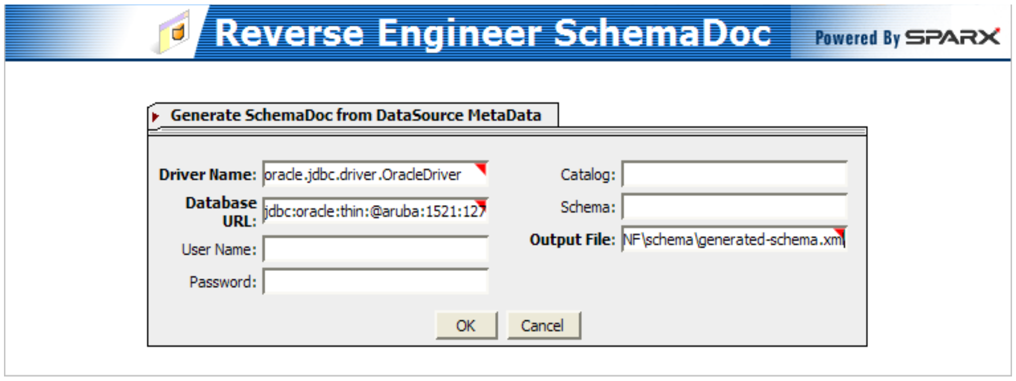


Source file:
If your file is not in C:\Projects\cura\WEB-INF/schema/data, please enter it below in 'Other file'.
Other file:
If you specify a file in this field, it will be used instead of the selection above.
DAL Schema Class: app.dal.DataAccessLayer
JNDI Datasource:
If a JNDI data source is not available, specify the Driver/URL parameters below.
Driver Name: oracle.jdbc.driver.OracleDriver
Database URL: jdbc:oracle:thin:@localhost:1521:127.0.0.1
User Name:
Password:

Figure 28: Sparx Database Import XML Data

Database Reverse Engineer Page

The Database Reverse Engineer page (which is displayed when you choose the *Reverse Engineer* menu item from the *Database* menu) provides a form that allows you to generate a SchemaDoc file based on an existing database that is accessible through a JDBC driver.


Reverse Engineer SchemaDoc
Powered By SPARX
Generate SchemaDoc from DataSource MetaData
OK
Cancel

Driver Name: oracle.jdbc.driver.OracleDriver
Database URL: jdbc:oracle:thin:@aruba:1521:127
User Name:
Password:
Catalog:
Schema:
Output File: NF\schema\generated-schema.xml

Figure 29: Sparx Database Reverse Engineer Page

ACE Documents Menu Items

Browse Source Code Online

You can review the source code for your application by choosing *Application* from the *Documents* menu. The first time you enter the application documentation section, you should see a list of all the directories that exist in the application's root directory. You can click on any one to navigate to it and view the list of files and sub-directories inside it. If you click on any XML, JSP or Java source file, you should also be able to see the source for those files directly from the browser.

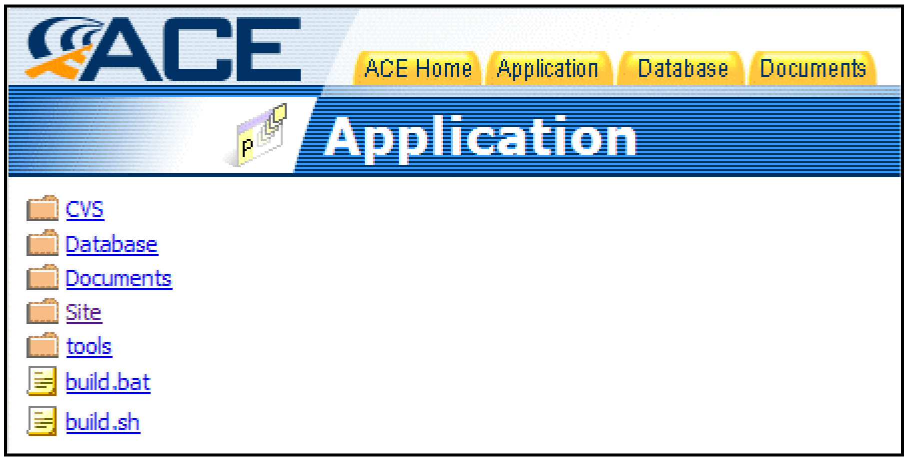
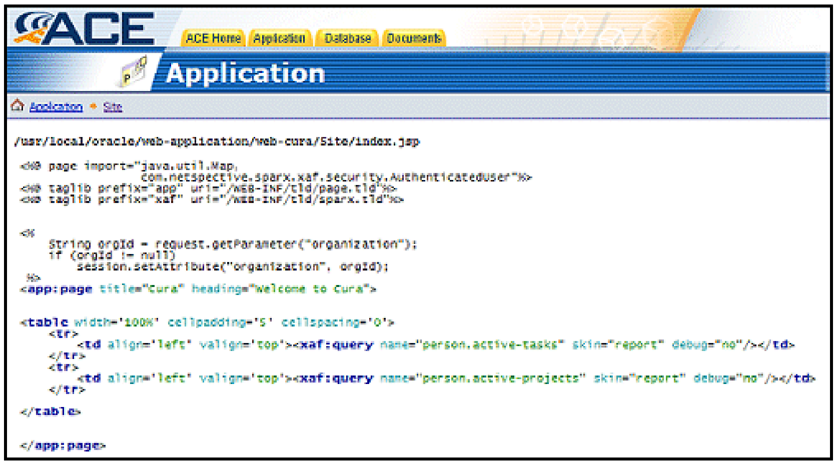
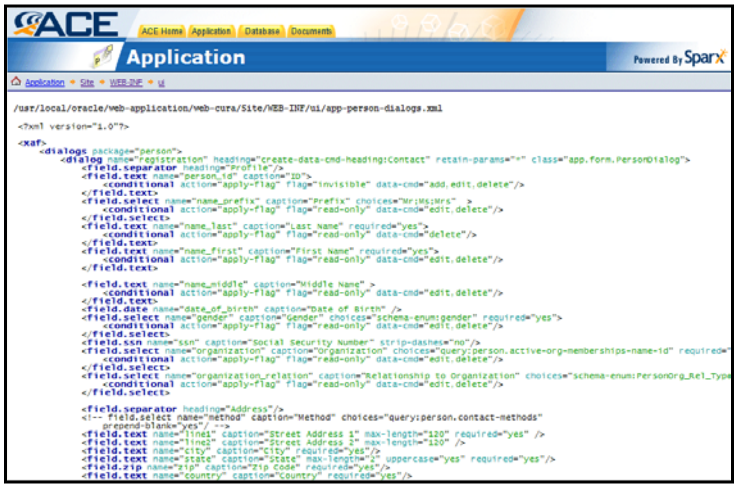


Figure 30: ACE showing directory structure of an application



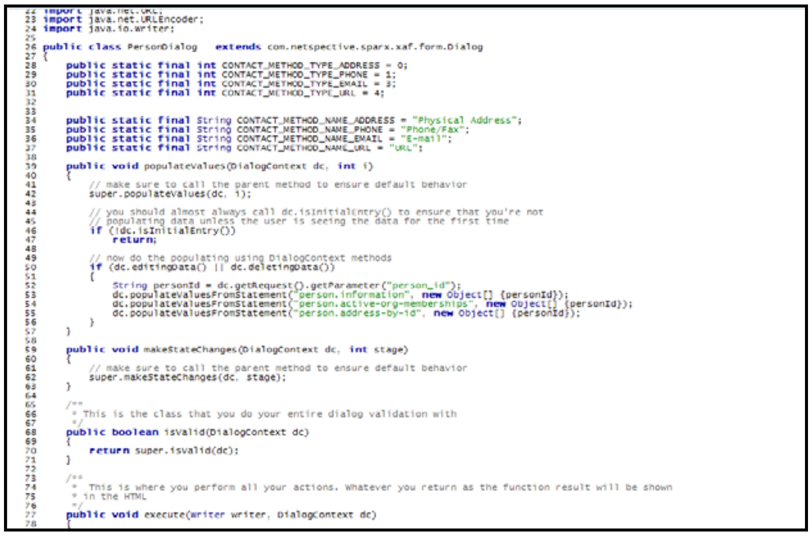
```
/usr/local/oracle/web-application/web-cura/Site/index.jsp

 <%@ page import="java.util.Map,
                  com.netspective.sparx.xaf.security.AuthenticatedUser"%>
 <%@ taglib prefix="app" uri="/WEB-INF/tld/page.tld"%>
 <%@ taglib prefix="xaf" uri="/WEB-INF/tld/sparx.tld"%>


 <%
     String orgId = request.getParameter("organization");
     if (orgId != null)
         session.setAttribute("organization", orgId);
  %>
 <app:page title="Cura" heading="Welcome to Cura">

 <table width='100%' cellpadding='5' cellspacing='0'>
     <tr>
         <td align='left' valign='top'><xaf:query name="person.active-tasks" skin="report" debug="no"/></td>
     </tr>
     <tr>
         <td align='left' valign='top'><xaf:query name="person.active-projects" skin="report" debug="no"/></td>
     </tr>

 </table>


 </app:page>
```

Figure 31: ACE showing how to view a JSP file



```
/usr/local/oracle/web-application/web-cura/Site/WEB-INF/ui/app-person-dialogs.xml

 <?xml version="1.0"?>

 <xaf>
     <dialogs package="person">
         <dialog name="registration" heading="create-data-cmd-heading:Contact" retain-params="*" class="app.form.PersonDialog">
             <field.separator heading="Profile"/>
             <field.text name="person_id" caption="ID">
                 <conditional action="apply-flag" flag="invisible" data-cmd="add,edit,delete"/>
             </field.text>
             <field.select name="name_prefix" caption="Prefix" choices="Mr;Ms;Mrs"  >
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.select>
             <field.text name="name_last" caption="Last Name" required="yes">
                 <conditional action="apply-flag" flag="read-only" data-cmd="delete"/>
             </field.text>
             <field.text name="name_first" caption="First Name" required="yes">
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.text>

             <field.text name="name_middle" caption="Middle Name" >
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.text>
             <field.date name="date_of_birth" caption="Date of Birth" />
             <field.select name="gender" caption="Gender" choices="schema-enum:gender" required="yes">
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.select>
             <field.ssn name="ssn" caption="Social Security Number" strip-dashes="no"/>
             <field.select name="organization" caption="Organization" choices="query:person.active-org-memberships-name-id" required="
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.select>
             <field.select name="organization_relation" caption="Relationship to Organization" choices="schema-enum:PersonOrg_Rel_Type
                 <conditional action="apply-flag" flag="read-only" data-cmd="edit,delete"/>
             </field.select>

             <field.separator heading="Address"/>
             <!-- field.select name="method" caption="Method" choices="query:person.contact-methods"
                 prepend-blank="yes"/ -->
             <field.text name="line1" caption="Street Address 1" max-length="120" required="yes" />
             <field.text name="line2" caption="Street Address 2" max-length="120" />
             <field.text name="city" caption="City" required="yes"/>
             <field.text name="state" caption="State" max-length="2" uppercase="yes" required="yes"/>
             <field.zip name="zip" caption="Zip Code" required="yes"/>
             <field.text name="country" caption="Country" required="yes"/>
```

Figure 32: ACE showing how to view an XML file



```
import java.net.URL;
import java.net.URLEncoder;
import java.io.Writer;

public class PersonDialog   extends com.netspective.sparx.xaf.form.Dialog
{
    public static final int CONTACT_METHOD_TYPE_ADDRESS = 0;
    public static final int CONTACT_METHOD_TYPE_PHONE = 1;
    public static final int CONTACT_METHOD_TYPE_EMAIL = 3;
    public static final int CONTACT_METHOD_TYPE_URL = 4;


    public static final String CONTACT_METHOD_NAME_ADDRESS = "Physical Address";
    public static final String CONTACT_METHOD_NAME_PHONE = "Phone/Fax";
    public static final String CONTACT_METHOD_NAME_EMAIL = "E-mail";
    public static final String CONTACT_METHOD_NAME_URL = "URL";

    public void populateValues(DialogContext dc, int i)
    {
        // make sure to call the parent method to ensure default behavior
        super.populateValues(dc, i);

        // you should almost always call dc.isInitialEntry() to ensure that you're not
        // populating data unless the user is seeing the data for the first time
        if (!dc.isInitialEntry())
            return;

        // now do the populating using DialogContext methods
        if (dc.editingData() || dc.deletingData())
        {
            String personId = dc.getRequest().getParameter("person_id");
            dc.populateValuesFromStatement("person.information", new Object[] {personId});
            dc.populateValuesFromStatement("person.active-org-memberships", new Object[] {personId});
            dc.populateValuesFromStatement("person.address-by-id", new Object[] {personId});
        }
    }

    public void makeStateChanges(DialogContext dc, int stage)
    {
        // make sure to call the parent method to ensure default behavior
        super.makeStateChanges(dc, stage);
    }

    /**
     * This is the class that you do your entire dialog validation with
     */
    public boolean isValid(DialogContext dc)
    {
        return super.isValid(dc);
    }

    /**
     *  This is where you perform all your actions. Whatever you return as the function result will be shown
     * in the HTML
     */
    public void execute(Writer writer, DialogContext dc)
    {
```

Figure 33: ACE showing how to view a Java file

As the screenshots have already shown, it is extremely easy to navigate the entire source tree for an application. Further, when viewing any of the source files of the application, they are presented in a easy to use form complete with syntax highlighting.

Execution Environment

Sparx supports the notion of environments like *Development*, *Testing*, and *Production*. When in the development environment, XML-based resource files like dialogs, schemas, SQL, and others are automatically reloaded. In Testing and Production environments automatic reloading is not enabled. Your applications can use the environment setting to make appropriate decisions about data sources and other environment-specific settings (like throwing exceptions in a development environment but e-mailing errors in testing or production).

How to specify the execution environment

The environment is specified in `WEB-INF/web.xml` by setting the context parameter variable `app-exec-environment`. That variable can have one of the following values (these values are case-sensitive): `Production`, `Testing`, or `Development`.

Sample WEB-INF/web.xml entry

```
<?xml version="1.0"?>

<web-app>

    <-- setup execution environment -->
    <context-param>
        <param-name>app-exec-environment</param-name>
        <!-- <param-value>Production</param-value> -->
        <!-- <param-value>Testing</param-value> -->
        <param-value>Development</param-value>
    </context-param>
```

```
</web-app>
```

Datasources and Database Connectivity

Sparx provides powerful database connection and aggregation services. Starting with a simple interface to one or more database connection and pooling engines and including such features as dynamic data source definitions and selection, the database connectivity support sets the stage for both static and dynamic SQL libraries and pooled/cached result sets.

The method of specifying datasources is dependent upon the application server being used. XIF's default Java class for managing datasources and their locations is `com.netspective.sparx.xif.db.context.BasicDatabaseContext` and it uses the JNDI provider to provide pooled database connections. For other application servers, XAF provides an interface called `com.netspective.sparx.xif.db.DatabaseContext` which can be used to implement a new database connection system dependent upon the application server's preferred way of obtaining datasources. The way to register the application server specific database object is to set the system property in `WEB-INF/conf/sparx.xml`:

```
<system-property
      name="com.netspective.sparx.xif.db.DatabaseContext.class"
      value="your.db.context.BasicDatabaseContext"/>
```

C H A P T E R 5

Sparx Forms Management

The previous chapters helped you get acquainted with Sparx, its development paradigms and the ease with which a Sparx application can go from concept to prototype. However it touched only very briefly with the capabilities of Sparx, concentrating more on its application development methodology instead of its depth of features.

In this chapter we will delve deeper into the kinds of user interface that can be created using Sparx. We will examine some sample dialogs that are shipped with the evaluation kit. These dialogs will demonstrate the ease with which Sparx allows a developer to create complex dialogs that have all the server-side and client-side functionality pre-coded. This allows developers to concentrate on creating the application rather than spending time creating the Javascript libraries or other custom widgets they need for the application.

Forms Processing Overview

This section discusses the overview of the forms creation, caching, state management, and output. The steps described here refer to the diagram on the following page.

	STEP	DESCRIPTION
1	The form specification	All specification may be performed in XML by functional or business analysts or Java programmers or in completely in Java by Java programmers. If the specification is done in XML, XSLT style sheets are provided that automatically generate functional specification documentation in ACE.
2	The forms pool	Contains all the forms as XML elements; each Dialog object is created the first time it is called (using the Dialog class) and then cached for future use.
3	The form model	Contains only the form's structural information, field types, rules, etc. and may be sub classed; the class is cached and reused whenever needed. Each application will have only one instance of the form model and that instance will be reused by every user of the application.
4	The controller, Dialog Context Bean (DCB) "data bean", and state machine	This is the state machine that manages field state and field data retrieval/storage and may be sub classed; a new class is created for each request. Since the form model is shared, this controller is what's specific to each user and request.
5	The view (the skin)	Contains the HTML rendering rules and may be sub classed; class is cached and reused. A single instance of any skin serves all users in the application.
6	Rendering the form (using the skin)	The dialog will automatically create HTML, perform client-side validation, wait for submission, perform server-side validation, and maintain the state until the dialog's input is
7	Form validation	

	STEP	DESCRIPTION
8	Form logic/execution	perfect; at this point, it will execute the form's custom functionality or process any XML-based tasks.

Table 19 : Steps involed in Processing Forms

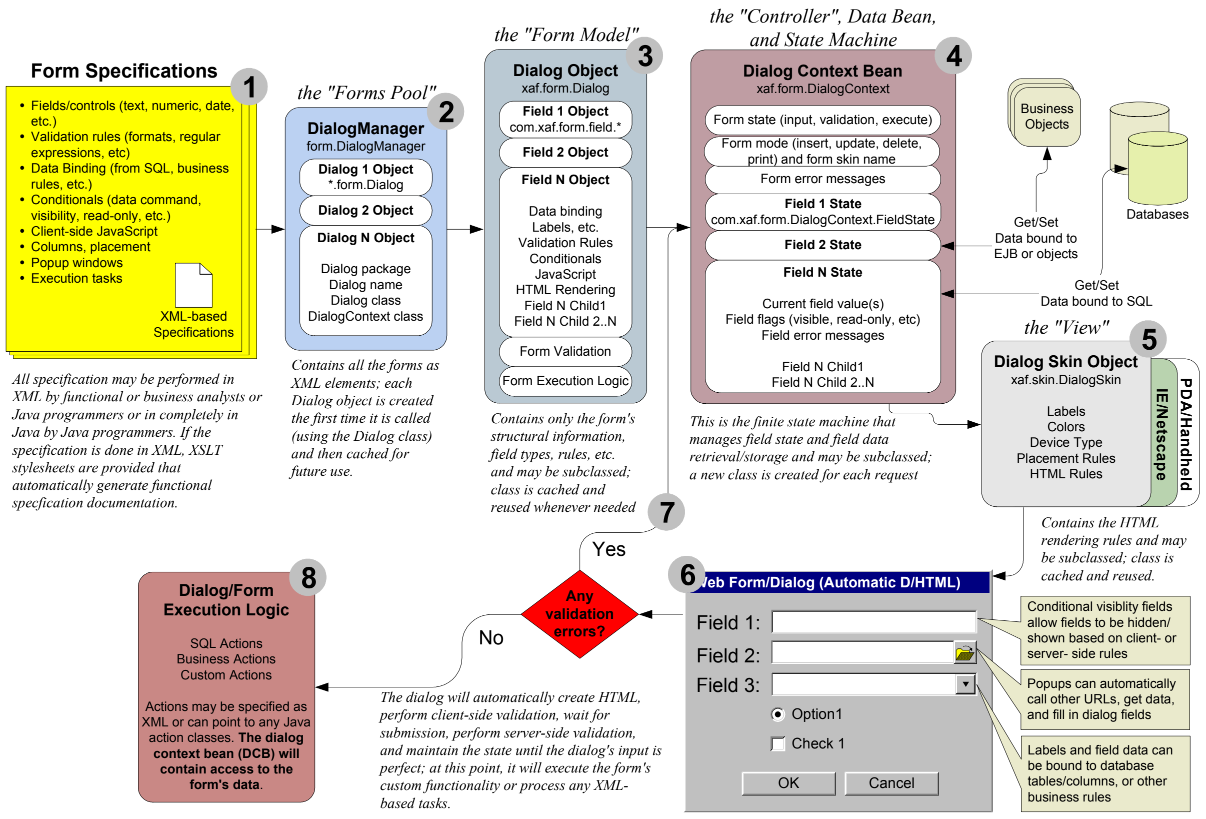

Form Specifications
1
Fields/controls (text, numeric, date, etc.)
Validation rules (formats, regular expressions, etc)
Data Binding (from SQL, business rules, etc.)
Conditionals (data command, visibility, read-only, etc.)
Client-side JavaScript
Columns, placement
Popup windows
Execution tasks
XML-based Specifications
All specification may be performed in XML by functional or business analysts or Java programmers or in completely in Java by Java programmers. If the specification is done in XML, XSLT stylesheets are provided that automatically generate functional specfication documentation.
the "Forms Pool"
2
DialogManager
form.DialogManager
Dialog 1 Object
*.form.Dialog
Dialog 2 Object
Dialog N Object
Dialog package
Dialog name
Dialog class
DialogContext class
Contains all the forms as XML elements; each Dialog object is created the first time it is called (using the Dialog class) and then cached for future use.
the "Form Model"
3
Dialog Object
xaf.form.Dialog
Field 1 Object
com.xaf.form.field.*
Field 2 Object
Field N Object
Data binding
Labels, etc.
Validation Rules
Conditionals
JavaScript
HTML Rendering
Field N Child1
Field N Child 2..N
Form Validation
Form Execution Logic
Contains only the form's structural information, field types, rules, etc. and may be subclassed; class is cached and reused whenever needed
the "Controller", Data Bean, and State Machine
4
Dialog Context Bean
xaf.form.DialogContext
Form state (input, validation, execute)
Form mode (insert, update, delete, print) and form skin name
Form error messages
Field 1 State
com.xaf.form.DialogContext.FieldState
Field 2 State
Field N State
Current field value(s)
Field flags (visible, read-only, etc)
Field error messages
Field N Child1
Field N Child 2..N
This is the finite state machine that manages field state and field data retrieval/storage and may be subclassed; a new class is created for each request
Business Objects
Databases
Get/Set
Data bound to EJB or objects
Get/Set
Data bound to SQL
the "View"
5
Dialog Skin Object
xaf.skin.DialogSkin
Labels
Colors
Device Type
Placement Rules
HTML Rules
IE/Netscape
PDA/Handheld
Contains the HTML rendering rules and may be subclassed; class is cached and reused.
7
Yes
No
Any validation errors?
6
Web Form/Dialog (Automatic D/HTML)
Field 1:
Field 2:
Field 3:
Option1
Check 1
OK
Cancel
Conditional visiblity fields allow fields to be hidden/ shown based on client- or server- side rules
Popups can automatically call other URLs, get data, and fill in dialog fields
Labels and field data can be bound to database tables/columns, or other business rules
8
Dialog/Form Execution Logic
SQL Actions
Business Actions
Custom Actions
Actions may be specified as XML or can point to any Java action classes. The dialog context bean (DCB) will contain access to the form's data.
The dialog will automatically create HTML, perform client-side validation, wait for submission, perform server-side validation, and maintain the state until the dialog's input is perfect; at this point, it will execute the form's custom functionality or process any XML-based tasks.

Figure 34: Steps involved in Processing Forms

Form Creation Flowchart

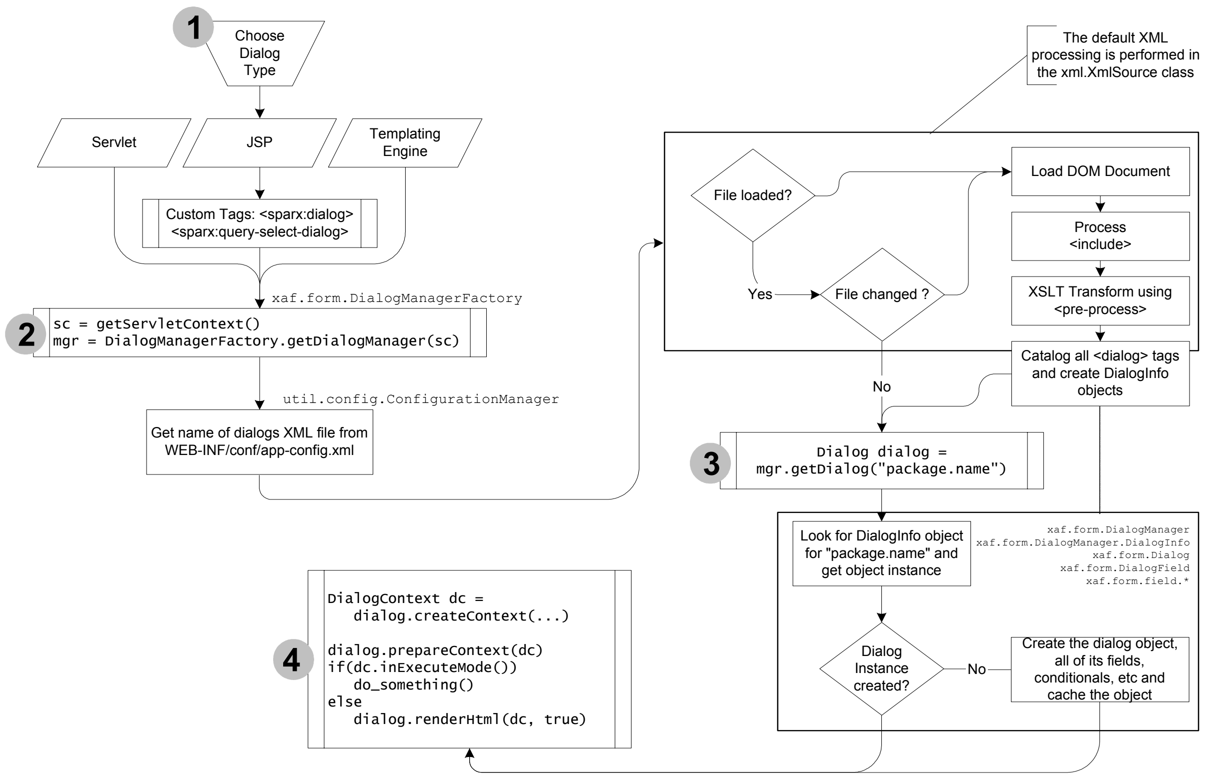
Forms are created using the following steps (which are shown in the diagram on the following page).

	PROCESS
1	Define dialog in `WEB-INF/ui/dialogs.xml` (the "dialog component").
2	Choose how you want to call the dialog (through a servlet, a JSP, or a templating engine).
3	If you use the JSP call, all of the code is automatic -- you just used the <sparx:dialog> or <sparx:query-select-dialog>.

	PROCESS

4	If you use the templating engine or Servlet model, you use factory methods and constructors and then use the following code:
	
	`sc = getServletContext()`
	`mgr = DialogManagerFactory.getDialogManager(sc)`
	`Dialog dialog = mgr.getDialog("package.dialog_name")`
	`DialogContext dc = dialog.createContext(...)`
	`dialog.prepareContext(dc)`
	`if(dc.inExecuteMode())`
	`  do_something()`
	`else`
	`  dialog.renderHtml(writer, dc, true)`

Table 20: Steps involved in Creating Forms


1
Choose Dialog Type
Servlet
JSP
Templating Engine
Custom Tags: <sparx:dialog> <sparx:query-select-dialog>
xaf.form.DialogManagerFactory
2
util.config.ConfigurationManager
Get name of dialogs XML file from WEB-INF/conf/app-config.xml
The default XML processing is performed in the xml.XmlSource class
File loaded?
Yes
File changed ?
No
Load DOM Document
Process <include>
XSLT Transform using <pre-process>
Catalog all <dialog> tags and create DialogInfo objects
3
Look for DialogInfo object for "package.name" and get object instance
xaf.form.DialogManager
xaf.form.DialogManager.DialogInfo
xaf.form.Dialog
xaf.form.DialogField
xaf.form.field.*
Dialog Instance created?
No
Create the dialog object, all of its fields, conditionals, etc and cache the object
4

```
sc = getServletContext()
mgr = DialogManagerFactory.getDialogManager(sc)
```

```
Dialog dialog =
mgr.getDialog("package.name")
```

```
DialogContext dc =
   dialog.createContext(...)

dialog.prepareContext(dc)
if(dc.inExecuteMode())
   do_something()
else
   dialog.renderHtml(dc, true)
```

Figure 35: Steps involved in Creating Forms

Form Beans: Dialog Context Beans (DCBs)

As shown in step 4 of the "Steps involved in Processing Forms" diagram, the DCB plays a pivotal role in dialog processing. Acting as the "controller" of the MVC paradigm, it is the **only** object in the entire lifecycle that is created for **each user**. The "form model" (Dialog) and "form view" (skin) are created, cached, and reused for each user; however, the `DialogContext` or DCB is a new object for each user **and** each request. The basic `DialogContext` object (found in the `com.netspective.sparx.xaf.form` package) that is used for state management contains a number of useful methods like:

```
public boolean hasValue(String qualifiedName)
public String getValue(String qualifiedName)
public String getValue(String qualifiedName, String defaultValue)
public void setValue(String qualifiedName, String value)
```

However, each of these methods requires that you know and specify the field names as they are defined in the XML. If the names of the fields ever change in the XML, you would not catch the error until run-time. One feature that Sparx provides in both ACE and in the `build` scripts is to generate form/dialog-specific Dialog Context Beans. These form-specific DCBs will generate a DialogContext subclass for each dialog defined in your XML files with convenience wrapper methods for each field. For example, assume the following fields:

```
<dialog class="app.form.BookInfo">
      <field.text caption="Book ID" name="bookId"/>
      … other definitions go here …
</dialog>
```

Using the XML shown above, Sparx will generate the following DCB – notice all the convenience wrappers that simply call the base `DialogContext` methods but with the appropriate field name.

```
import com.netspective.sparx.xaf.form.*;

public class BookInfoContext extends DialogContext
{

public boolean isBookIdValueSet() { return hasValue("bookId"); }
public boolean isBookIdFlagSet(long flag) { return
      flagIsSet("bookId", flag); }
public void setBookIdFlag(long flag) { setFlag("bookId", flag); }
public void clearBookIdFlag(long flag) { clearFlag("bookId", flag); }
public String getBookIdRequestParam() { return
      request.getParameter("_dc.bookId"); }
public DialogField getBookIdField() { return getField("bookId"); }
public DialogContext.DialogFieldState getBookIdFieldState() { return
      getFieldState("bookId"); }
public void addBookIdErrorMsg(String msg) { addErrorMessage("bookId",
      msg); }
public String getBookId() { return getValue("bookId"); }
public String getBookId(String defaultValue) { return
      getValue("bookId", defaultValue); }
public String getBookIdOrBlank() { return getValue("bookId", ""); }
public void setBookId(String value) { setValue("bookId", value); }
```

Using the `BookInfoContext` instead of `DialogContext` provides easier maintenance in the future because if the XML changes, these method names will change and break your code at compile (which is preferable to breaking at runtime).

Dialog Data Commands

XAF dialogs automatically understand and process common dialog data commands like `add`, `edit`, `delete`, `print`, and `confirm`. With these common commands and the ability to have declarative conditionals, fields can appear/disappear based on dialog data commands and population of fields can be made to happen on only when specific commands are encountered. Data commands can be hard-coded in the dialogs or may be specified in URLs for added flexibility (but a little less security). Using a single dialog XML specification and appropriate use of data commands, significant amounts of code can be reduced because the same dialogs code can handle all adding, updating, and deleting of complex data. Dialog Data Commands can eliminate dozens of pages from most complex web applications because you'll never create more than one page to handle all aspects of the same dialog (add, edit, delete, etc.).

Standard Dialog Data Commands

COMMAND	ACTION
add	The dialog should be processed for inserting records into a database.
edit	The dialog should be processed for updating records in a database. This mode will automatically make any primary-keys read-only.
delete	The dialog should be processed for deleting records in a database. This mode will automatically make all fields read-only (for confirmation) and allow submission.
print	The dialog should be processed for printing information on the screen. All the items become read-only and a few tweaks are made so that the dialog looks more like a report than a form.
confirm	This mode will automatically make all fields read-only (for confirmation) but does not infer a further action.

How to use Dialog Commands

As part of a URL

Assuming a dialog is being called through a Servlet or a JSP file, the `data_cmd` parameter establishes the mode. Although this method is more flexible (allowing a single Servlet or JSP page to do many actions using the same XML declarations and bound Java classes), it is less secure because it gives the ability for the user to change the mode. For example,

```
http://myserver/sample-app/sample-page.jsp?data_cmd=add
http://myserver/sample-app/sample-servlet?data_cmd=edit
http://myserver/sample-app/sample-page2.jsp?data_cmd=print
```

Inside a JSP or Servlet

Assuming a dialog is being called through a Servlet or a JSP file, the `data_cmd` request attribute establishes the mode. This mode is more secure in the sense that the end user may not override what the programmer wants to do. For example,

```
request.setAttribute("data_cmd", "add");
request.setAttribute("data_cmd", "edit");
```

Sample Dialog with Data Command Conditionals

```
<dialog name="DialogTest_10" heading="Test Data Command and
       Conditionals" loop="yes">
       <field.boolean name="checkbox_field" caption="Checkbox"
       style="checkalone"/>
       <field.static name="static_field2" default="Checkbox checked!">
              <conditional action="display" partner="checkbox_field"
       js-expr="control.checked == true"/>
       </field.static>
       <field.text name="text_field_1" caption="Text Field" size="50"
       hint="Invisible when ADD" default="I guess the data command is
       not 'add'">
```

```
            <conditional action="apply-flag" flag="invisible" data-
        cmd="add"/>
        </field.text>
        <field.select name="select_field" caption="Select Field"
            style="combo" choices="Choice 1=Choice 1's value;Choice
            2=Choice 2's value;Choice 3=Choice 3's value"
            default="Choice 1's value">
        <conditional action="apply-flag" flag="invisible" data-
        cmd="add"/>
        </field.select>
        <field.static name="static_field_4" default="The data command
        is not 'add' or 'edit'">
            <conditional action="apply-flag" flag="invisible" data-
        cmd="add"/>
            <conditional action="apply-flag" flag="invisible" data-
        cmd="edit"/>
        </field.static>
</dialog>
```

Dialog Execution (Sparx Dialogs as Web Services)

Dialogs allow data to be auto-populated and auto-executed. This means that a single dialog can serve multiple purposes: for use as a front-end for a user to edit/enter data and for a back-end process to enter data through a scripted process and automatically execute the same dialog as if a user had entered the data manually. This way, the same code can double as both a web-based application UI component and as a web service when the need arises.

Field names in Sparx versus HTML

As you have already learned, each field in a dialog has a unique name. When that unique field name is rendered into an HTML form, it is prefixed with a special string "`_dc.`" that uniquely identifies that it's part of a Sparx field (to prevent name collision with non-Sparx fields in a form). The string "`_dc.`" means "private dialog control". For example, assume the following fields:

```
<field.text name="text_01"/>
<field.composite name="composite_01">
        <field.integer name="integer_01"/>
        <field.text name="text_02"/>
</field.composite>
```

Although the first text field is known to Sparx as `text_01`, in HTML it is actually defined as `_dc.text_01`. Similarly, the first integer field in Sparx is known as `composite_01.integer_01` but in HTML it is actually defined as `_dc.composite_01.integer_01`. To see how the "`_dc.`" is added to your own dialogs, simply view the HTML source when you test your dialog in a browser.

Auto Population of Fields Values using URL Request Parameters

Now that you know how dialog field names are generated in HTML, you can use that knowledge to do what's called "auto-population" of the fields using simple URL strings. For example, if you wanted your fields to have default values you could use the `default` attribute built into most `<field.xxx>` tags like this:

```
<field.text name="text_01" default="Text Default"/>
```

```
<field.composite name="composite_01">
	<field.integer name="integer_01" default="1000"/>
	<field.text name="text_02" default="request:text_02_default"/>
</field.composite>
```

Armed with the `default` attribute, now the fields can have starter values in case a user does not enter any data. Note the default for `composite_01.text_02` – it is set to a request parameter SingleValueSource meaning it will look for a request parameter called `text_02_default` in the URL and assign it the default at runtime. So, if this dialog were placed into a JSP file called test_dialog.jsp you could fill in the default value like this:

```
http://host:port/app/test_dialog.jsp?text_02_default=TEST
```

If you wanted to setup the ability to do that for all of the fields in the dialog, you could use the `default="request:field_name_default"` attribute for each of the fields. However, given that each of the fields in a Sparx dialog already have special names like `_dc.text_01` and `_dc.composite_01.integer_01`, you can use those special names to specify default values that **override** any `default` attributes found in a <field.xxx> specification. So, for example, to specify field values for all fields in our fictitious dialog we could do the following:

```
http://host:port/app/test_dialog.jsp?_dc.text_01=TEST1&_dc.composite_
	01.integer_01=350&_dc.composite_01.text_02=TEST2
```

Please note that you can include as many fields in the URL as you like and each parameter's value will automatically populate the proper field at runtime.

Auto Execution for Web Services using URL Parameters

A special feature of all dialogs is that they can be auto-executed – meaning that they can contain a user-interface that can be bypassed (optionally still running the data validation, though). This is useful if you have a dialog running in a JSP or Servlet that you want to run through a single URL, background process, or as a web service without having to write any new code. To auto-execute any dialog (meaning have the dialog take the data and start processing it immediately without showing a form) you need to simply add a `_d_exec=1` command in the URL. Combining that command with the ability to auto-populate values using the URL becomes a very powerful combination. Here's how to populate some values and execute a dialog at the same time:

```
http://host:port/app/test_dialog.jsp?_dc.text_01=TEST1&_d_exec=1
```

Auto Execution for Web Services using Java

Calling dialogs through Java and not rendering HTML allows you wrap web services around existing Dialogs. You can use the following code to obtain a dialog and execute it directly without rendering HTML.

```
import javax.servlet.ServletContext;
import com.netspective.sparx.xaf.form.Dialog;
import com.netspective.sparx.xaf.form.DialogContext;
import com.netspective.sparx.xaf.form.DialogManager;
import com.netspective.sparx.xaf.form.DialogManagerFactory;
import com.netspective.sparx.xaf.form.DialogSkin;
import com.netspective.sparx.xaf.skin.SkinFactory;


ServletContext context = vc.getServletContext();
DialogManager manager = DialogManagerFactory.getManager(context);
Dialog dialog = manager.getDialog("dialogName");
DialogSkin skin = null;

DialogContext dc = dialog.createContext(context, getServlet(),
      (javax.servlet.http.HttpServletRequest) getRequest(),
      (javax.servlet.http.HttpServletResponse) getResponse(), skin);

dc.setValue("xyz", "abc");
dialog.execute(out, dc);
```

Test Dialogs

This set of sixteen dialogs exists for the express purpose of testing as many different Sparx widgets as possible. Each dialog demonstrates some different aspect of one or more widget. Integrating them in your own application for testing purposes is a matter of copying the XML file into your `WEB-INF/UI` directory and then including it into your main `dialogs.xml` file.

Adding Test Dialogs to Your Application

The XML definitions for the test dialogs are located in the main Sparx installation directory in a file called `test-dialogs.xml`. To copy this file over to your Sparx application, ensure that you are in the application's UI directory. Then copy the `test-dialogs.xml` file over to your application's `UI` directory using the command shown below.

```
copy WEB_APPS_HOME\cura\WEB-INF\ui\test-dialogs.xml[16] .
```

Your Sparx application's `WEB-INF\ui` directory should now contain the `test-dialogs.xml` and `dialogs.xml` files in addition to any other files that might already be there. To automatically include all the test dialogs in your application, you need to add one line to your `dialogs.xml` file. Therefore open up your `dialogs.xml` file in a text editor and insert the following line immediately under the `<xaf>` tag.

```
      <include file="test-dialogs.xml"/>
```

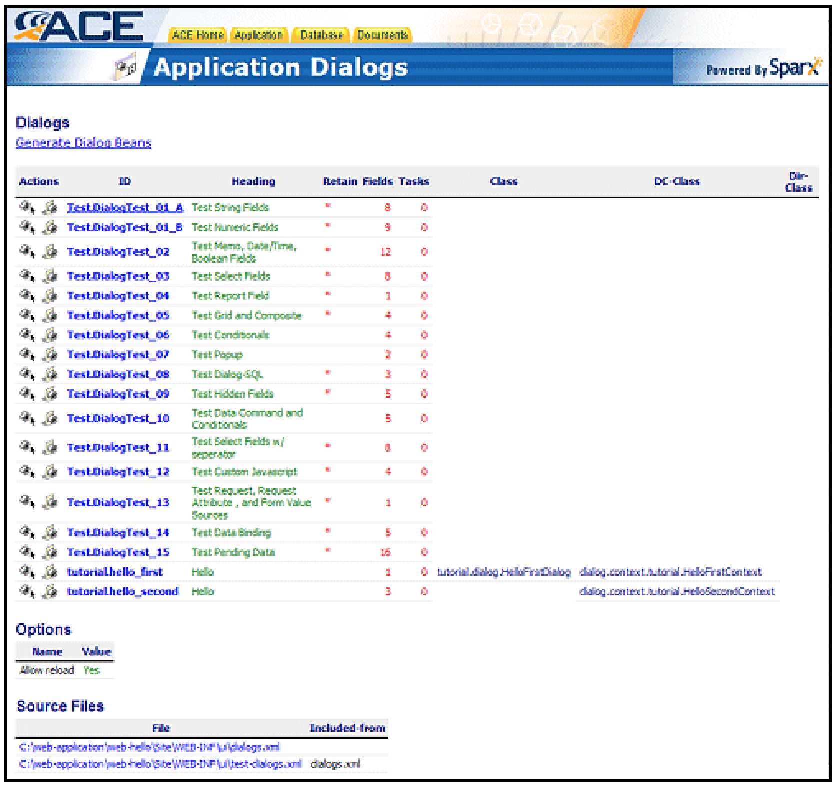
You can also open up the `test-dialogs.xml` file in a text editor to see its structure so you know what to expect when you go back to your application's ACE. It will also show you what your XML dialog files should look like if they are to be included in the main `dialogs.xml` file using an `<include>` tag.

The Test Dialogs in ACE

With the test dialogs added to your application, you can see the list of new dialogs by going to your application's ACE and choosing *Dialogs* from the *Application* menu. Each of the test dialogs has a somewhat descriptive heading that gives an indication

[16] By default, this is C:\Netspective\resin-x.y.z\webapps\cura\WEB-INF\ui\test-dialogs.xml

of its purpose. Let us go through each dialog and point out some of the important features.



Dialogs

Generate Dialog Beans

Actions	ID	Heading	Retain	Fields	Tasks	Class	DC-Class	Dir-Class
	Test.DialogTest_01_A	Test String Fields	*	8	0			
	Test.DialogTest_01_B	Test Numeric Fields	*	9	0			
	Test.DialogTest_02	Test Memo, Date/Time, Boolean Fields	*	12	0			
	Test.DialogTest_03	Test Select Fields	*	8	0			
	Test.DialogTest_04	Test Report Field	*	1	0			
	Test.DialogTest_05	Test Grid and Composite	*	4	0			
	Test.DialogTest_06	Test Conditionals		4	0			
	Test.DialogTest_07	Test Popup		2	0			
	Test.DialogTest_08	Test Dialog SQL	*	3	0			
	Test.DialogTest_09	Test Hidden Fields	*	5	0			
	Test.DialogTest_10	Test Data Command and Conditionals		5	0			
	Test.DialogTest_11	Test Select Fields w/ seperator	*	8	0			
	Test.DialogTest_12	Test Custom Javascript	*	4	0			
	Test.DialogTest_13	Test Request, Request Attribute , and Form Value Sources	*	1	0			
	Test.DialogTest_14	Test Data Binding	*	5	0			
	Test.DialogTest_15	Test Pending Data	*	16	0			
	tutorial.hello_first	Hello		1	0	tutorial.dialog.HelloFirstDialog	dialog.context.tutorial.HelloFirstContext	
	tutorial.hello_second	Hello		3	0		dialog.context.tutorial.HelloSecondContext	

Options

Name	Value
Allow reload	Yes

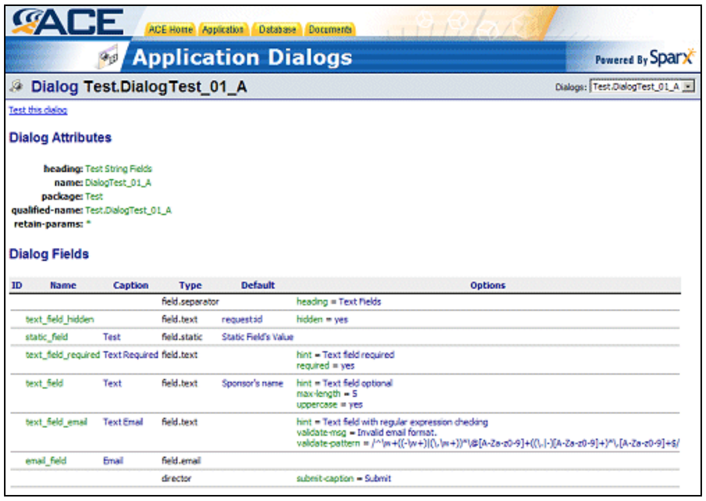
Source Files

File	Included-from
C:\web-application\web-hello\Site\WEB-INF\ui\dialogs.xml	
C:\web-application\web-hello\Site\WEB-INF\ui\test-dialogs.xml	dialogs.xml

Figure 36: A List Of All new And Old Dialogs.

Test Dialog 01 A

This dialog is a test of the various string fields that are built into Sparx. Click on the dialog name to see a listing of the different fields that are used in this dialog.



Dialog Test.DialogTest_01_A

Dialogs: Test.DialogTest_01_A

Test this dialog

Dialog Attributes

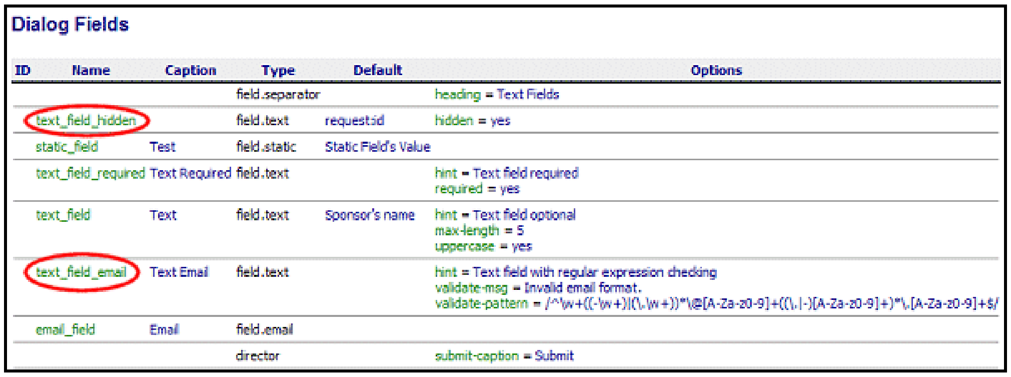
heading:	Test String Fields
name:	DialogTest_01_A
package:	Test
qualified-name:	Test.DialogTest_01_A
retain-params:	*

Dialog Fields

ID	Name	Caption	Type	Default	Options
			field.separator		heading = Text Fields
	text_field_hidden		field.text	request:id	hidden = yes
	static_field	Test	field.static	Static Field's Value	
	text_field_required	Text Required	field.text		hint = Text field required required = yes
	text_field	Text	field.text	Sponsor's name	hint = Text field optional max-length = 5 uppercase = yes
	text_field_email	Text Email	field.text		hint = Text field with regular expression checking validate-msg = Invalid email format. validate-pattern = /^\w+((-\w+)|(\.\w+))*\@[A-Za-z0-9]+((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$/
	email_field	Email	field.email		
			director		submit-caption = Submit

Figure 37: `Test.DialogTest_01_A` Attributes and Fields

The two fields that deserve special attention among the list are `text_field_hidden` and `text_field_email`.



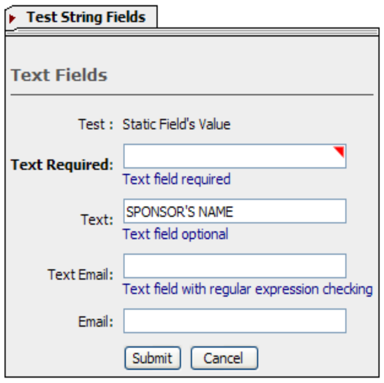
Dialog Fields

ID	Name	Caption	Type	Default	Options
			field.separator		heading = Text Fields
	text_field_hidden		field.text	request:id	hidden = yes
	static_field	Test	field.static	Static Field's Value	
	text_field_required	Text Required	field.text		hint = Text field required required = yes
	text_field	Text	field.text	Sponsor's name	hint = Text field optional max-length = 5 uppercase = yes
	text_field_email	Text Email	field.text		hint = Text field with regular expression checking validate-msg = Invalid email format. validate-pattern = /^\w+((-\w+)|(\.\w+))*\@[A-Za-z0-9]+((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$/
	email_field	Email	field.email		
			director		submit-caption = Submit

Figure 38: `text_field_hidden` and `text_field_email`

`text_field_hidden` is of interest primarily due to it's default value as seen in the ACE. The default value is shown as `request:id`. This is a Sparx-specific dynamic variable known as a *value source*. Value sources are written in a form similar to a URL (`name:param`). This particular value source (a request source) provides the value of the variable id as passed into the dialog using URL encoding. To test this field and value source out, click on the "Test this dialog" link and when the rendered dialog pops up in a new browser window, change the URL in the browser location bar by appending `?id=testThisDialog` to it. Now when you click on OK and the dialog is processed, you will notice that the value of `text_field_hidden` as reported by the default Java dialog is `testThisDialog`. This is one way in which you can access URL encoded parameters given to a dialog without using any custom Java.

text_field_email is of interest because of the validation regular expressions as seen in the ACE. All data entered in this field is matched against the Perl5 regular expression given in the `validate-pattern` option and, if it fails, the message given in the `validate-msg` option is output to the user to let him know the cause of failure. Go ahead and try it to see how robust the regular expression really is as well as to get an idea of how server-side validation works with Sparx.



Test String Fields

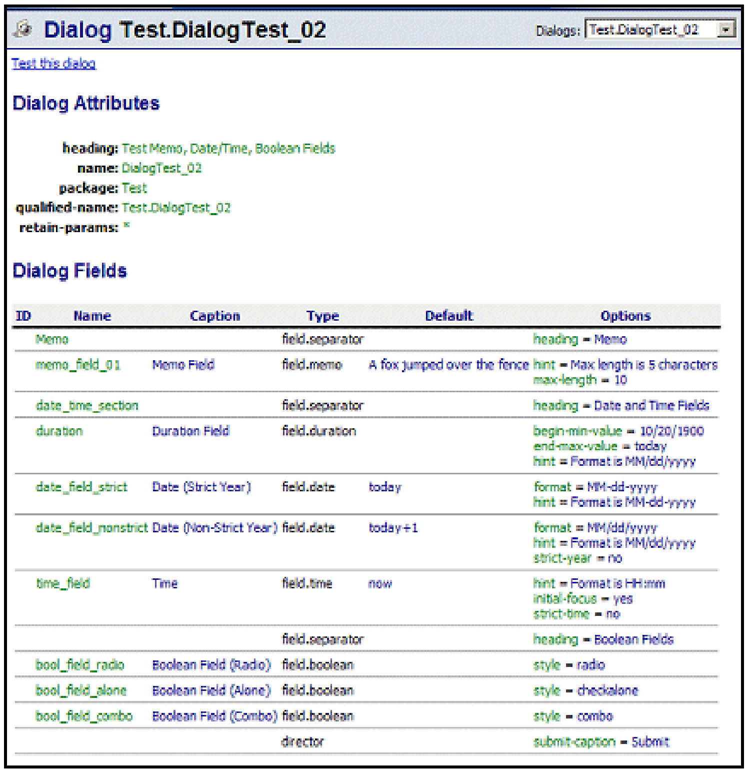
Text Fields

Test :	Static Field's Value
Text Required:	Text field required
Text:	SPONSOR'S NAME Text field optional
Text Email:	Text field with regular expression checking
Email:	

Figure 39: `Test.DialogTest_01` A Dialog Unit Test

Test Dialog 02

This dialog is a test of, among others, date/time and Boolean fields. Click on the dialog name to see a listing of the different fields that are used in this dialog. The interesting fields in this dialog are the `duration` field and the three `bool_field_*` fields which are the different visual representations of the same underlying data.



Dialog Attributes

heading: Test Memo, Date/Time, Boolean Fields
name: DialogTest_02
package: Test
qualified-name: Test.DialogTest_02
retain-params: *

Dialog Fields

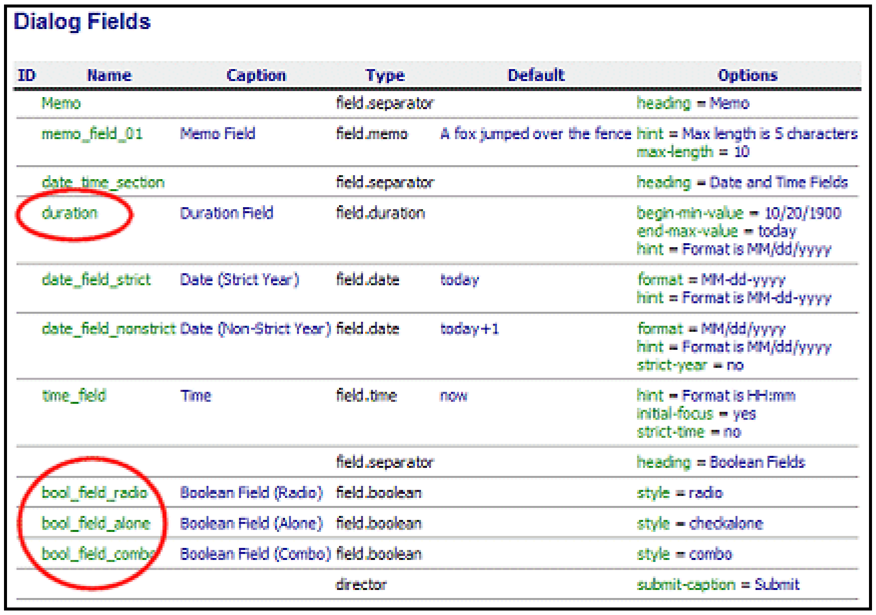
ID	Name	Caption	Type	Default	Options
	Memo		field.separator		heading = Memo
	memo_field_01	Memo Field	field.memo	A fox jumped over the fence	hint = Max length is 5 characters max-length = 10
	date_time_section		field.separator		heading = Date and Time Fields

	duration	Duration Field	field.duration		begin-min-value = 10/20/1900
					end-max-value = today
					hint = Format is MM/dd/yyyy

	date_field_strict	Date (Strict Year)	field.date	today	format = MM-dd-yyyy hint = Format is MM-dd-yyyy
	date_field_nonstrict	Date (Non-Strict Year)	field.date	today+1	format = MM/dd/yyyy hint = Format is MM/dd/yyyy strict-year = no
	time_field	Time	field.time	now	hint = Format is HH:mm initial-focus = yes strict-time = no
			field.separator		heading = Boolean Fields
	bool_field_radio	Boolean Field (Radio)	field.boolean		style = radio
	bool_field_alone	Boolean Field (Alone)	field.boolean		style = checkalone
	bool_field_combo	Boolean Field (Combo)	field.boolean		style = combo
			director		submit-caption = Submit

Figure 40: `Test.DialogTest_02` Attributes and Fields

`duration` is interesting since it is the first multi-widget field that we have encountered. Since this is a ranged field, a user will need to fill it such that the first date is earlier than the second date. The good news is that Sparx takes care of all that validation for you so you can concentrate on what you have to do with the date range once you have obtained it. Go ahead and try filling this field incorrectly and you will be greeted with an error message telling you what went wrong with the field.



Dialog Fields

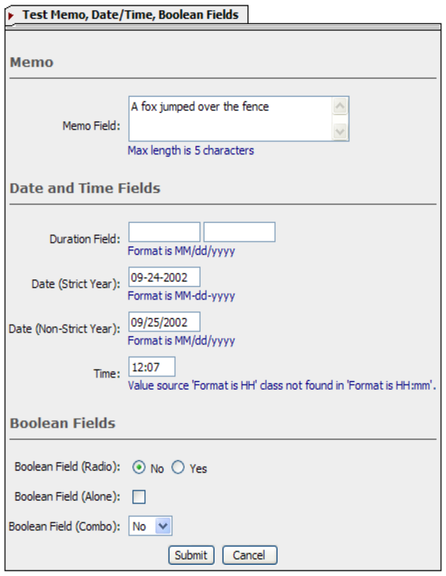
ID	Name	Caption	Type	Default	Options
	Memo		field.separator		heading = Memo
	memo_field_01	Memo Field	field.memo	A fox jumped over the fence	hint = Max length is 5 characters max-length = 10
	date_time_section		field.separator		heading = Date and Time Fields

	duration	Duration Field	field.duration		begin-min-value = 10/20/1900
					end-max-value = today
					hint = Format is MM/dd/yyyy

	date_field_strict	Date (Strict Year)	field.date	today	format = MM-dd-yyyy hint = Format is MM-dd-yyyy
	date_field_nonstrict	Date (Non-Strict Year)	field.date	today+1	format = MM/dd/yyyy hint = Format is MM/dd/yyyy strict-year = no
	time_field	Time	field.time	now	hint = Format is HH:mm initial-focus = yes strict-time = no
			field.separator		heading = Boolean Fields
	bool_field_radio	Boolean Field (Radio)	field.boolean		style = radio
	bool_field_alone	Boolean Field (Alone)	field.boolean		style = checkalone
	bool_field_combo	Boolean Field (Combo)	field.boolean		style = combo
			director		submit-caption = Submit

Figure 41: `duration` and `bool_field` fields

The `bool_field_*` fields are an interesting exercise in determining how much flexibility Sparx allows a developer. All three fields are of type field.boolean with the sole difference being the value of the style parameter. With this power in hand, you can work on what functionality is toggled by this boolean variable, not how it is rendered or interpreted by your code.



Test Memo, Date/Time, Boolean Fields

Memo

Memo Field: A fox jumped over the fence
Max length is 5 characters

Date and Time Fields

Duration Field:
Format is MM/dd/yyyy

Date (Strict Year): 09-24-2002
Format is MM-dd-yyyy

Date (Non-Strict Year): 09/25/2002
Format is MM/dd/yyyy

Time: 12:07
Value source 'Format is HH' class not found in 'Format is HH:mm'.

Boolean Fields

Boolean Field (Radio): [✓ No] Yes

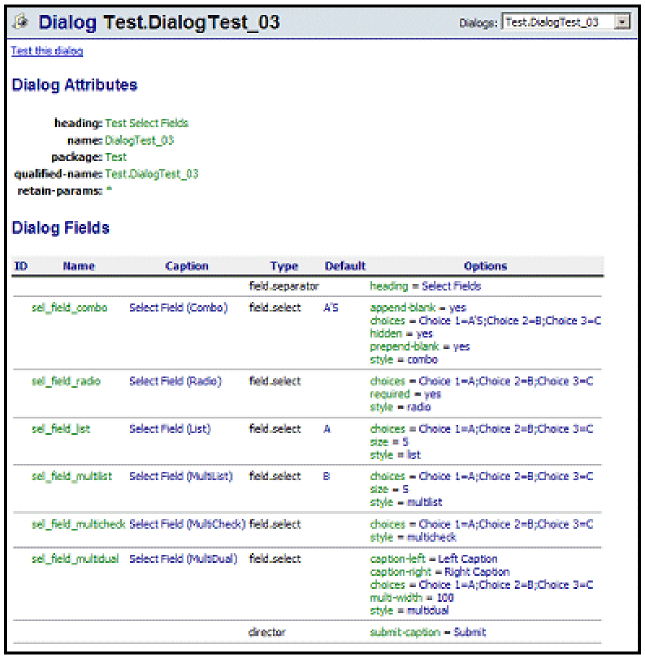
Boolean Field (Alone): ☐

Boolean Field (Combo): No

Figure 42: `Test.DialogTest_02` Unit Test

Test Dialog 03

This dialog is a test of select fields. Click on the dialog name to see a listing of the different fields that are used in this dialog. The one field that is most interesting in this dialog is the one named `sel_field_multidual`.



Dialog Test.DialogTest_03

Dialogs: Test.DialogTest_03

Test this dialog

Dialog Attributes

heading:	Test Select Fields
name:	DialogTest_03
package:	Test
qualified-name:	Test.DialogTest_03
retain-params:	*

Dialog Fields

ID	Name	Caption	Type	Default	Options
			field.separator		heading = Select Fields
	sel_field_combo	Select Field (Combo)	field.select	A'S	append-blank = yes choices = Choice 1=A'S;Choice 2=B;Choice 3=C hidden = yes prepend-blank = yes style = combo
	sel_field_radio	Select Field (Radio)	field.select		choices = Choice 1=A;Choice 2=B;Choice 3=C required = yes style = radio
	sel_field_list	Select Field (List)	field.select	A	choices = Choice 1=A;Choice 2=B;Choice 3=C size = 5 style = list
	sel_field_multilist	Select Field (MultiList)	field.select	B	choices = Choice 1=A;Choice 2=B;Choice 3=C size = 5 style = multilist
	sel_field_multicheck	Select Field (MultiCheck)	field.select		choices = Choice 1=A;Choice 2=B;Choice 3=C style = multicheck
	sel_field_multidual	Select Field (MultiDual)	field.select		caption-left = Left Caption caption-right = Right Caption choices = Choice 1=A;Choice 2=B;Choice 3=C multi-width = 100 style = multidual
			director		submit-caption = Submit

Figure 43: `Test.DialogText_03` Fields and Attributes

Firstly it is important to test this dialog and see the sheer variety of ways in which the same `field.select` can be represented visually. The best part about all this is that since Sparx collects the data from all these fields, as a developer you are presented with the same unified interface to the data for all these different visual renderings of the same field.

Secondly the `sel_field_multidual` field is an example of the enormous amount of work that exists in the form of the Sparx JavaScript libraries for the client-side and the Java libraries for the server-side of just this one field. You now have yet another option for your user interface, all without writing a single line of the code that makes the multidual field work.

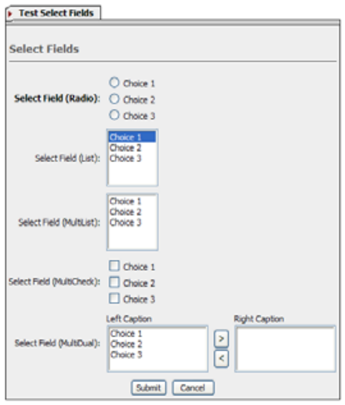
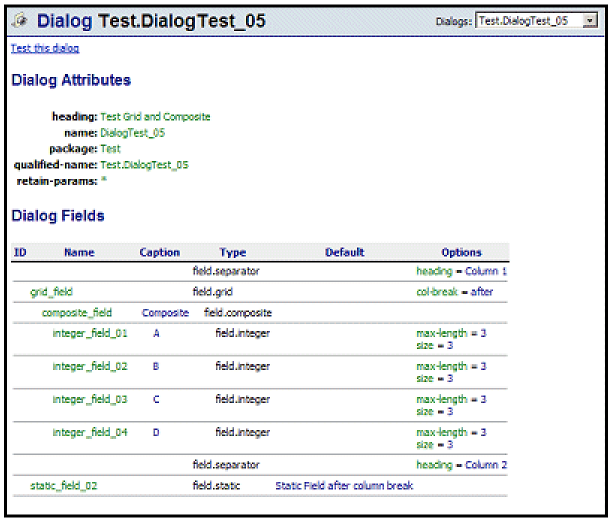


Figure 44: `Test.DialogTest_03` Unit Test

Test Dialog 05

This dialog is a test of grid and composite fields. Click on the dialog name to see a listing of the different fields that are used in the dialog. Pay careful attention to the nesting of the composite field inside the grid field.



Dialog Test.DialogTest_05

Dialogs: Test.DialogTest_05

Test this dialog

Dialog Attributes

heading: Test Grid and Composite
name: DialogTest_05
package: Test
qualified-name: Test.DialogTest_05
retain-params: *

Dialog Fields

ID	Name	Caption	Type	Default	Options
			field.separator		heading = Column 1
	grid_field		field.grid		col-break = after
	composite_field	Composite	field.composite		
	integer_field_01	A	field.integer		max-length = 3 size = 3
	integer_field_02	B	field.integer		max-length = 3 size = 3
	integer_field_03	C	field.integer		max-length = 3 size = 3
	integer_field_04	D	field.integer		max-length = 3 size = 3
			field.separator		heading = Column 2
	static_field_02		field.static	Static Field after column break	

Figure 45: `Test.DialogTest_05` Fields and Attributes

Both grid and composite fields are a good way to create multi-widget fields that can group lots of similar of disparate data together for easier data entry. In the case of the composite field example, you can see that not only do all four sub-fields have the same type (which is not necessary) but they also have captions displayed as column headings. In the case of the grid field example, you can see that the first field in the grid is the composite field and the second is of a totally different type: static.

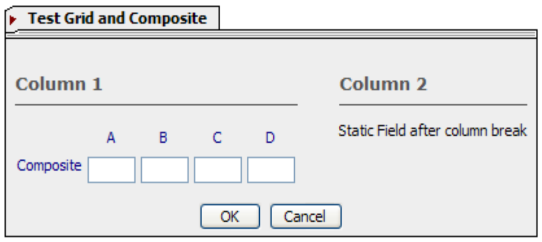
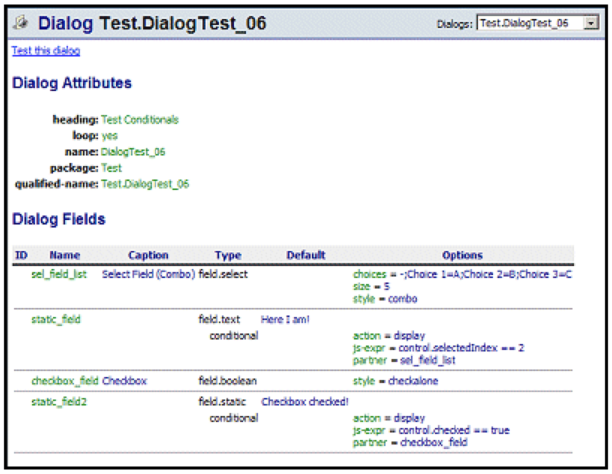


Figure 46: `Test.DialogTest_05` Unit Test

Test Dialog 06

This dialog is a test of conditional fields. Click on the dialog name to see a listing of the different fields that are used in the dialog. Pay careful attention to the `js-expr` and `partner` options of the two static fields.



Dialog Test.DialogTest_06

Dialog Attributes

heading:	Test Conditionals
loop:	yes
name:	DialogTest_06
package:	Test
qualified-name:	Test.DialogTest_06

Dialog Fields

ID	Name	Caption	Type	Default	Options
	sel_field_list	Select Field (Combo)	field.select		choices = -;Choice 1=A;Choice 2=B;Choice 3=C size = 5 style = combo
	static_field		field.text conditional	Here I am!	action = display js-expr = control.selectedIndex == 2 partner = sel_field_list
	checkbox_field	Checkbox	field.boolean		style = checkalone
	static_field2		field.static conditional	Checkbox checked!	action = display js-expr = control.checked == true partner = checkbox_field

Figure 47: `Test.DialogTest_06` Fields and Attributes

Test the dialog and you will notice that the two static fields are initially not visible but become visible when you choose "Choice 2" in the combo box and when you check the checkbox. Additionally, they disappear when the combo box choice changes or when the checkbox is unchecked. All this client-side interaction is handled by the Sparx Javascript library. In conjunction with the Sparx Java library on the server-side you are guaranteed to get only the information you want and no more or less.

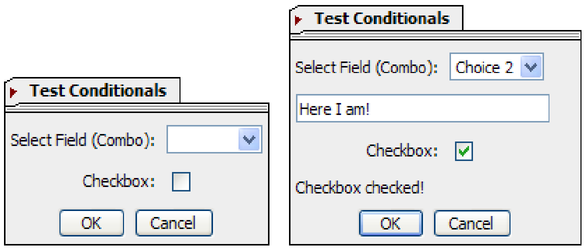


Figure 48: `Test.DialogTest_06` Unit Tests

Conclusion

With this small sampling of dialogs, we hope you now have a better understanding of the sheer power and flexibility that Sparx offers a developer on the client-side as well as the server-side. There are more test dialogs that you may want to explore and experiment with. Find out more about how Sparx can light the fire.

C H A P T E R 6

Library Tutorial

The Sparx Library is a project meant to take you a few steps beyond the basic *Hello World* application that you covered in an earlier chapter. Whereas the Hello World example got you familiar with Sparx development, this example will build upon that knowledge to teach you how to create a simple but complete and functional real world application.

Hello World dealt exclusively with the user interface aspect of application development and stayed away from the most important part of any application: the back end that involves database access and application of business logic. The Sparx Library will lead you through everything it takes to get an actual application up and running with special focus on developing the SQL and data management layers of your application.

Functionality

The Sparx Library is a small application meant to be used for a personal library of books. It allows you to track the books you have and add more books to your collection or edit information stored about existing books. It also allows you to search your collection for a particular book based on your own custom search criteria.

The overall functionality of the application is limited but complete. As such it demonstrates a few of the main types of data manipulation that developers need to take care of in every application. In the end the goal is to show you just how much power can be wielded with just a few lines of XML and Java code when armed with the strength of Sparx.

Design

Application Design

The Sparx Library is designed around the basic Sparx components you have already seen and a few that you might not have worked with yet. The Library's user interface will be a combination of XML dialogs for the front end data entry and validation while the back end will use Java dialogs for the processing and business logic required. It will use static SQL and associated reports to help you track the books stored in the application. Searches, on the other hand, require a dynamically generated SQL in the form of a Query Definition. The glue for all these different components will be the JSP pages that all these different Sparx components are embedded in. Last but not least, these JSP pages will have a consistent look and feel thanks in part to custom JSP tags that will be used to create the headers and footers for each JSP page.

Having taken care of the user interface and data processing components of the application, the last aspect of the Library is data storage. It is possible to use expensive and (in all certainty) overkill databases like Oracle for this application. Instead, the data storage of choice is the Java-based embedded database that ships with the Sparx evaluation kit: HypersonicSQL. Sparx-based applications scale exceptionally well from the individual level to the enterprise level. Therefore, if you need to switch to another database engine at a later date, it would be a simple matter of reconfiguring the data source for the Library and setting the new database up with all existing data.

Database Design

The Library deals with books and only books. Therefore the information that needs to be stored in the database will be about books. The four pieces of information that the Library will store for this example are its title, genre, author, and ISBN number. Of these, the genre is the only one that can be common across multiple books. In database language, then, the genre (book type) has a one-to-many relationship[17] with the books.

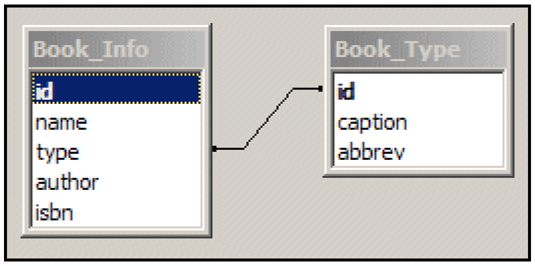


Figure 49: Baisc E-R Diagram for Library's Database

The figure shows the entity-relationship diagram for the data we will be using. The database for the Library will be designed to store each entity (and its attributes) in a separate table. As such this will be the first normalized form of storage. As with the application design, the database design will become clearer when it is implemented later in this chapter.

Creating the Schema (Data Layer)

When describing the implementation of The Sparx Library, this tutorial will be more concise than in the description of the earlier Hello World project. This section assumes you have already gone through the Hello World project and are familiar with a lot of the development paradigms that are associated with Sparx. This includes a good understanding of the purposes and locations of the various directories and files in a Sparx application, a good understanding of all the different Sparx components

[17] To learn more about database design, one-to-many relationships and entity-relation diagrams, and the overall topic of Database Normalization, please go to http://www.devshed.com/Server_Side/MySQL/Normal/Normal1/print

already used in Hello World, and a familiarity with the variety of other components that Sparx has to offer, both for the front end (e.g. ACE for UI design testing etc) and the back end (database integration etc).

This tutorial will also assume that you have the Sparx evaluation kit installed with the default values for path and port. The list of paths that this tutorial will refer to is shown here. If the actual values for these paths are different, you should change any different paths to match yours. The `x.y.z` template next to a directory name is a version number (like `2.1.1`).

PROPERTY	DEFAULT
Installation Path	`c:\Netspective`
SPARX_HOME	`c:\Netspective\sparx-x.y.z`
RESIN_HOME	`c:\Netspective\resin-x.y.z`
WEB_APPS_HOME	`c:\Netspective\resin-x.y.z\webapps`
Hello World Install Path	`c:\Netspective\resin-x.y.z\webapps\library`
Resin Web Server Port	8080

Table 21: Default Installation Paths for Sparx Evaluation Kit's Library Sample Application

Starting the Application

If you're using the library application from the evaluation kit, it's already setup for you automatically so the following is informational only (in case you want to know how to do it yourself in the future).

Creation of a Sparx application involves the same basic steps that you went through when creating the Hello World application. Please refer to the section titled *Setting up the Application* on page 15 for more information on how to achieve this.

Following the directions for the Hello World project, create the directory named `c:\Netspective\resin-x.y.z\webapps\library` to store all files related to the Sparx Library. You should verify that the new application (and its corresponding context path) was created successfully by restarting Resin and accessing the application using a web browser.

Setting up the Database

With the empty application successfully created and running, it is time to work on the backbone of the Sparx Library: the database. There are two steps towards setting up the database.

- Decide where you want to store the database files. Based on the directory structure given in Chapter 2, the most logical choice would be the `APP_ROOT/WEB-INF/database` directory.
- Configure your Sparx application to have a database connection pointing towards your new database. This will be the only way for your Sparx applications to be aware of a database and to communicate with it. For Resin, this is accomplished by setting up a `<resource-ref>` in `WEB-INF/web.xml` and is the method used by the evaluation kit.

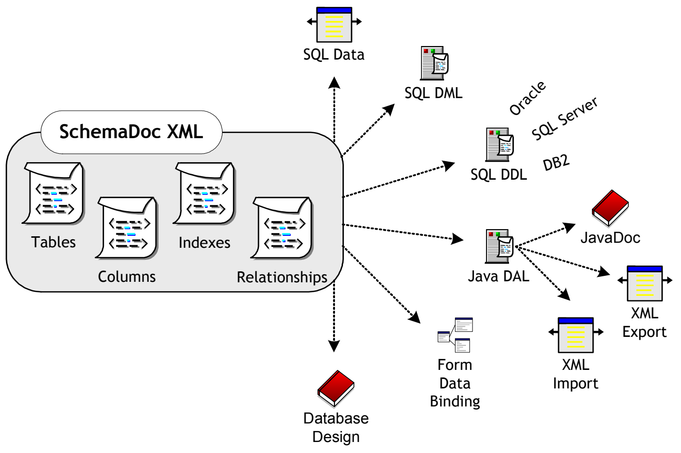
Once you have a data source configured, you can use ACE to verify that it is indeed recognized as a valid data source and that the JDBC driver for the source has been located and bound to the data source. To do this, open the Sparx Library ACE in a web browser. After logging in, go to the *Database* menu and choose the *Data Sources* item.

If everything is configured properly, the Sparx Library ACE will show the name of the data source you just created as well as details about the driver, some JDBC specific information about the kind of driver you are using with this data source and the username you configured the data source with.

The Database Schema

After analyzing the information that needs to be stored in the database and judging from the E-R diagram shown earlier, you can derive the database schema that is necessary for the Sparx Library. It is a very simple schema consisting of only two tables, one to store information about the different genres of books (book type) and the other to store all four attributes of the books in the library (book info). The two tables are 1:n related by the `type` stored in both tables.

This entire schema, and other larger and more complex ones that you might develop for enterprise applications, can be represented for Sparx entirely in self-documenting XML (called a *SchemaDoc*). Once entered as XML, this schema is available for platform-independent database access from any Sparx application. In addition, the entire schema (including relationships, data and metadata) can be viewed using a standard browser through your application's ACE. Within the ACE you can also use the XML schema to generate database-specific DDL that can be interpreted by the DBMS of your choice. Further, this XML schema is also used to generate the Data Access Layer to provide strongly typed Java classes for seamless programmatic access to your data. As depicted in the diagram, the exact same XML SchemaDoc source file feeds multiple outputs.



The XML SchemaDoc is stored in the `WEB-INF/schema/schema.xml` directory. The database schema described above and shown in the diagram is represented to Sparx using the XML contents of the two files shown below. As a first step towards creating the Sparx Library you should ensure that these files are created, with these exact contents, in the `WEB-INF/schema` directory of the Sparx Library.

The XML below belongs in the `schema.xml` file.

```
<?xml version="1.0"?>
```

```
<schema name="db">
 <include file="datatypes.xml"/>
 <include file="tabletypes.xml"/>
 <include file="enums.xml"/>

 <table name="Book_Info" abbrev="bkI" type="Default">
      <column name="id" type="text" size="10" primarykey="yes"
      descr="Unique ID for every book in the database"/>
                                                     e of the
      book"/>
               ame="type" lookupref="Book_Type"/>
      <column name="author" type="text" size="64" descr="Name of the
      author(s)"/>
                    isbn" type="text" size="10" descr="The 10 digit
      ISBN number"/>
 </tabl
</schema>
```

The XML below belongs in the `enums.xml` file.

```
<?xml version="1.0"?>

<schema name="enumerations">
 <include file="datatypes.xml"/>
 <include file="tabletypes.xml"/>

 <table name="Book_Type" abbrev="bkT" type="Enumeration">
      <enum>Science Fiction</enum>
      <enum>Mystery</enum>
      <enum>Business</enum>
      <enum>Information Technology</enum>
      <enum>Nuclear Physics</enum>
      <enum>Chemistry</enum>
 </table>
</schema>
```

Step by Step Explanation

Looking first at the contents of `schema.xml`, the first thing you should notice between the opening and closing `schema` tags are the `include` tags. Each `include` tag instructs the XML parser to load insert the specified file at the current location in the schema document. This ability to split up a schema into multiple files for easier management allows a more organized approach to specifying a database schema to Sparx.

The first file that is included (`datatypes.xml`) contains XML definitions for default data types that are used in the rest of the schema. This file must be included in all XML schemas in all your applications. The second included file contains XML definitions for default table types used in the rest of the schema. This file should be modified and included in all XML schemas for all your applications. The third and final included file contains XML definitions of any enumerated tables that might be a part of your schema.

Before moving on further, default table types need to be explained in a little more detail. These tables are templates for your own tables containing fields to store such metadata as record creation timestamps, etc. Developers should modify the default table types to their specifications and derive all custom tables from one or more base table types. Of course you can also choose to create a second layer of table types which are geared towards a particular application and then derive all actual tables from this layer of table types; the figurative sky is the limit. This ability to derive tables from other tables unleashes the ability for tables to inherit the properties of their parent table types. It is very useful for grouping similar types of tables together, allowing a developer to change a few things in the base table to effect a group-wide change in all derived tables.

On to the first table defined in the schema: `Book_Info`. This table is where the Sparx Library will store all information regarding its books. As you can see, all the tags in between the starting and ending `table` tag are `column` tags. Each `column` tag defines a field. The attributes of the tag determine the characteristics of the field as created in the database.

The most common attributes for a `<column>` tag are shown in four of the five fields defined in the `Book_Info` table and are described in the table below. A complete reference is available at http://developer.netspective.com/xif/tables.html.

NAME	DESCRIPTION
name	The name of the column. Each column is usually named as a singular noun in all lower case with each word inside a name separated by underscores. For instance, `person_id` is a good column name but `personid` is harder to read. If you use the appropriate naming convention, the Data Access Layer generator will produce better output.
type	The name of data-type to inherit. All of the attributes and elements from the other data-type will be inherited and any attributes and elements defined in this data-type will override those values. You can get an idea of all the types available for use with this attribute by looking at the datatypes.xml file.
size	The size attribute of the specified data type. What the size means depends on the data type. For example, in text fields, it specifies the number of characters used.
primarykey	Specifies whether or not this column is a primary key. Only a single primary key column per table is supported.
required	Specifies whether or not this column is a required column. If the column is required, XIF generates a not null constraint.
lookupref	Format is `Table_X.Column_Y`. Specifies a general foreign key relationship from the defining column which references the foreign `Column_Y` of `Table_X` (creates a 1:1 or 1:N relationship between defining column and the referenced column). If you use this attribute, the `type` attribute is not required (it's set to the same type as the referenced column).
parentref	Format is `Table_X.Column_Y`. Specifies a parent/child foreign key relationship which indicates that `Table_X` is a parent of the defining column using `Column_Y`'s value (creates a 1:N relationship between `Table_X` and the column defining the `parentref`). If you use this attribute, the type attribute is not required (it's set to the same type as the referenced column).
usetype	Format is "`Table_X.Column_Y`". This attributes instructs the XIF to look up the type definition of `Column_Y` in `Table_X` and copy the type definition for the foreign column. This is a special way of maintaining that two columns share the same data-type and size. If you use this attribute, the `type` attribute is not required (it's set to the same type as the referenced column).

NAME	DESCRIPTION
selfref	Format is "`Table_X.Column_Y`". Specifies a a self-referential foreign key relationship which indicates that `Table_X Column_Y` is used to maintain an internal hierarchy (creates a 1:N relationship between itself). If you use this attribute, the `type` attribute is not required (it's set to the same type as the referenced column).
unique	Specifies whether this column's values should be unique (meaning no two rows should share the same value for this column). When this value is set to yes, this attribute creates a unique index based on this single column. If more than one column needs to be unique (as a composite), use the `<index>` child element of the table element to create a unique index based on multiple columns.
indexed	Specifies whether this column's values should be indexed (for increasing search performance). When this value is set to yes, this attribute creates a search index based on this single column. If more than one column needs to be unique (as a composite), use the `<index>` child element of the table element to create an index based on multiple columns.
descr	Specifies usage information for the column.
default	Specifies the SQL expression that will be used as the column's default value in the SQL create table statement

Table 22: Sparx XIF `<column>` tag attributes

Enumeration Tables

The second table in our schema, the `Book_Type` table, is used to store information about the different genres of books stored in the Sparx Library. This table is defined as an Enumeration to Sparx and is conventionally stored in the file `enums.xml` that is then included in the `schema.xml`.

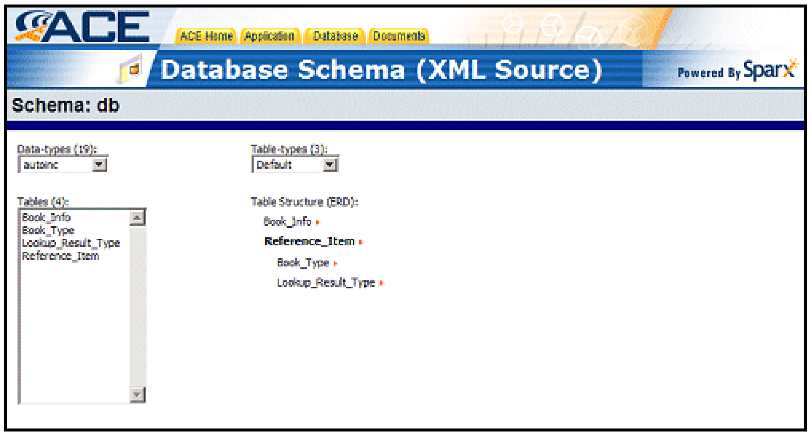
An enumeration is a special type of table that is generated by Sparx. It consists of three fields per record: a unique `id` which is used to relate the enumeration table in a 1:n manner with other tables, a non-null `caption` that is used to provide a short description of each value in the enumeration and an optional `abbrev`(iation) for the caption. An enumeration table is always of a fixed length since each record that goes into the table must be defined in the XML by hand. The syntax of an enumeration table is unlike that of regular tables. However, once parsed and interpreted, enumeration tables are translated into a set of regular tables for relational integrity purposes.

Using enumeration tables can significantly improve performance for static and lookup tables and still allow normal relational logic (foreign keys). All lookups in Sparx dialogs using the `schema-enum` value source directly read and cache enumeration XML files instead of asking the database for the contents of the database. Since the DDL generated by XIF for enumeration tables is normal SQL (using standard `create table` commands), relational logic is maintained but performance is significantly improved.

An enumeration table is used to establish a 1:n relationship between an attribute of an object (e.g. the genre of the book in this case) and the object itself (in this case, the book). It does this by letting the `id` field of the attribute enumeration table be inserted as a foreign key in the table containing records for the object. In this particular scenario, the `lookupref` attribute of the type field in the `Book_Info` table makes that happen.

Testing the Schema

Having saved the contents of the `schema.xml` and `enums.xml` files, you can now test your newly defined database schemas in the ACE. Go to the Sparx Library ACE[18] in a web browser and, after logging in, choose the item *SchemaDoc (XML)* from the *Database* menu.



Database Schema (XML Source)

Schema: db

Data-types (19): autoinc

Table-types (3): Default

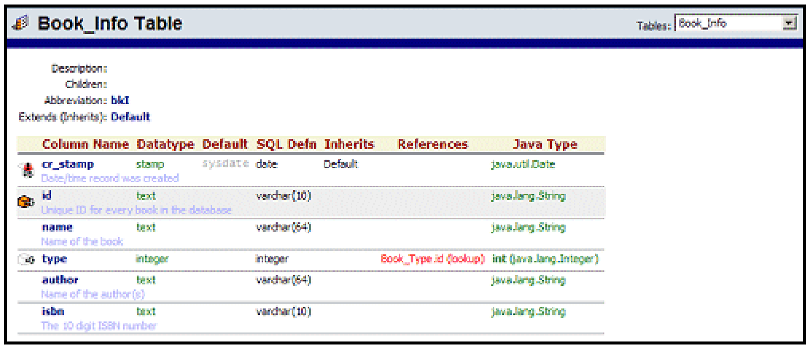
Tables (4):
Book_Info
Book_Type
Lookup_Result_Type
Reference_Item

Table Structure (ERD):
Book_Info
Reference_Item
Book_Type
Lookup_Result_Type

Figure 50: Database Schema (XML Source)

Near the top should be two combo boxes. One labeled Data-types contains a list of all the data types imported from the `data-types.xml` file. The second should be labeled Table-types and contains a list of the table types imported from the `table-types.xml` file.

Under these two combo boxes and to the left should be a list box that contains a list of all the tables in the schema. It should list a total of 4 tables, of which the most important to you are the ones you explicitly created: `Book_Info` and `Book_Type`. Go ahead and choose the `Book_Info` table to see more details about the table.



Book_Info Table

Tables: Book_Info

Description:
Children:
Abbreviation: **bkI**
Extends (Inherits): **Default**

Column Name	Datatype	Default	SQL Defn	Inherits	References	Java Type
cr_stamp Date/time record was created	stamp	sysdate	date	Default		java.util.Date
id Unique ID for every book in the database	text		varchar(10)			java.lang.String
name Name of the book	text		varchar(64)			java.lang.String
type	integer		integer		Book_Type.id (lookup)	**int** (java.lang.Integer)
author Name of the author(s)	text		varchar(64)			java.lang.String
isbn The 10 digit ISBN number	text		varchar(10)			java.lang.String

Figure 51: `Book_Info` Table

[18] If you are evaluating Sparx online, you can access the Sparx Collection ACE at http://developer.netspective.com/samples/library/ace

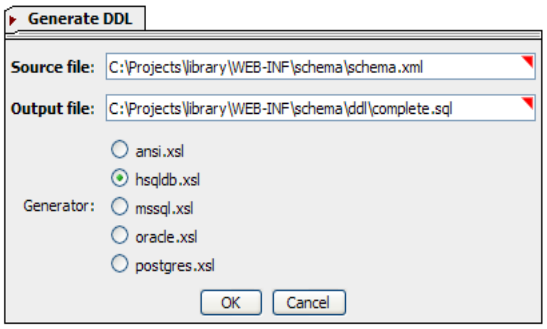
As you can see, the detailed view of the `Book_Info` table gives a lot of information about the table and the information stored in it. For each field in the table you can see its name (and description), the data type it was declared as (imported from `data-types.xml`), its default value (if any), the actual SQL data type it was created as, the table from which the current table inherits the field, whether it is a field that references other fields (such as the type field that references the `Book_Type` table) and finally the Java data type that is mapped to it.

You can view details for a different table by using the combo box that lists all the tables in the Sparx Library on the top right hand corner of the ACE screen. When you are done experimenting with the different schema related aspects of the ACE, you can get ready to install the schema to the database.

Generating the Data Description Language (DDL)

The DDL representation of your schema consists of the actual commands that you need to issue to a database to create the tables you specified in the schema and to populate them with any static data (such as the one stored in enumeration tables) if necessary. These commands are DBMS-specific and, for example, HypersonicSQL cannot necessarily interpret and execute DDL meant for Oracle.

Sparx uses standard W3C XSLT style sheets and an associated Database Policy class to translate its internal representation of your database schema into the DDL appropriate for your DBMS. Thus it is able to maintain a database-independent XML representation for internal and developer use while retaining the ability to generate a database-specific representation as needed.



Source file: C:\Projects\library\WEB-INF\schema\schema.xml
Output file: C:\Projects\library\WEB-INF\schema\ddl\complete.sql
Generator: ansi.xsl [✓ hsqldb.xsl] mssql.xsl oracle.xsl postgres.xsl

Figure 52: Generate DDL

To generate the DDL for the HypersonicSQL DBMS, get back into the Sparx Library ACE and select *Generate SQL DDL* from the *Database* menu. A dialog pops up that shows you the input schema file, the output SQL file and gives you a choice of translator to use. Ensure that the generated DDL goes into a file inside your `WEB-INF/schema/ddl` directory and that you have chosen the hsqldb translator. Press the OK button to have Sparx generate the SQL DDL for HypersonicSQL. Once generated, you can open the SQL DDL and examine the file to get an idea of how things get translated. Then, using a JDBC compliant tool, connect to the Sparx Library's HypersonicSQL database and execute the SQL DDL inside it to create the tables and insert all static data into the reference tables.

With this final step completed you should be ready to add, update, delete and query data from the database using the Sparx Library. To do that, however, we need a user interface that will allow us to manipulate data as well as query what is stored in the database.

Generating the Object-relational Map (Data Access Layer)

The Sparx DAL, or *Data Access Layer*, is a fully type-sensitive and completely database-independent Java mapping for each of the tables that are defined in the `schema.xml` file. Every table in the schema becomes multiple Java classes that can be used to access data, insert, update, and remove table rows, import and export XML data, and generally allow each of your tables to become easily used through Java without SQL. These classes are generated automatically using `schema.xml` whenever an application is built. The DAL API for an application along with its documentation allows a developer to quickly get up to speed on how to access data in any table of a database without going through the complicated setup, execution and result retrieval phases of JDBC programming. The whole process is painless and the end result is very human-readable Java code.

Generating the object-relational DAL Java files

This section is a quick reference to generating the DAL; you should refer to *Ant Build Scripts* on page 37 for more details about all of the various build targets.

```
cd APP_ROOT/WEB-INF

build dal
```

The example above shows how to run only the "dal" target. This would just generate and compile the DAL classes in the `WEB-INF/schema/java` directory but would not generate the associated JavaDocs for it. Please refer to page 8 for information on what is stored in the `WEB-INF/schema/java` directory.

```
cd APP_ROOT/WEB-INF

build dal dal-doc
```

The example above shows how to run only the "dal" target. This would generate and compile the DAL classes in the `WEB-INF/schema/java` directory and would also automatically generate the JavaDocs for it in the `WEB-INF/documents/javadoc/dal` directory.

```
cd APP_ROOT/WEB-INF

build
```

This example above shows how to the default target of "all" which would do build everything in the application including the DAL and JavaDocs related to the DAL.

A brief look at the generated DAL classes

When the DAL is generated, each of the tags in a SchemaDoc XML file becomes a Java class or interface using the following rules.

- Each <datatype> becomes a Java class that extends the `com.netspective.sparx.xif.dal.AbstractColumn` class and implements the `com.netspective.sparx.xif.dal.Column` interface. For example, the "text" datatype becomes a `app.dal.column.TextColumn` class and the "integer" datatype becomes the `app.dal.column.IntegerColumn` class.
- Each <table> generates three different Java files which represent the data stored in any given row in a table. In the Library example, the `Book_Info` table generates the following classes.
  - The `app.dal.domain.BookInfo` interface specifies the get and set methods that will allow access to the data stored in a single row. The reason this is an interface is so that it can easily be extended by classes that may need to store data in something other than a database.
  - The `app.dal.domain.row.BookInfoRow` class is a concrete class that represents an actual row in the physical `Book_Info` table. All of the columns become member variables and have
    - Get/set methods for all column data
    - For each child table, access to child rows that are represented by the <column parent="X.Y"> specification.
    - The `app.dal.domain.row.BookInfoRows` class represents a list of `BookInfoRow` objects and allows adding to the list and getting a specific row by number.
- Each <table> also generates its own `Table` class to allow inserts, updates, deletes, triggers, and other functions associated with a table. Whereas the row data mere holds data for a table row, the `Table` class actually performs database operations like inserts and updates. In the Library example, the `app.dal.table.BookInfoTable` class is generated and has the following features.
  - Methods like createBookInfoRow() and createBookInfoRows() to create row objects.
  - Methods like insert(conn, bookInfoRow), update(conn, bookInfoRow), and delete(conn, bookInfoRow) to perform SQL DML tasks (with transaction protection if needed).
  - Methods like getBookInfoByBookId() that retrieves a row by primary key or other specified manner.
  - If the table is a child table, it allows its parent to retrieve all child tables based on the foreign-key reference.

Although this section is a good reference for what's generated, the best way to learn the entire DAL API for any specific schema is to review the JavaDocs generated by the build script in the `WEB-INF/documents/javadoc/dal` directory. The JavaDocs will thoroughly describe each of the generated classes.

Creating the User Interface (Presentation Layer)

You are already familiar with the process of creating a user interface in XML and testing it in ACE. Therefore, this section will not dwell on that process. Instead, it will provide the XML definition of the dialog and explain how this one dialog will be able to accomplish most of the functionality of the Sparx Library.

```
<?xml version="1.0"?>
```

```
<xaf>
 <dialogs package="library">
 <dialog class="app.form.BookInfo" heading="create-data-cmd-
       heading:Book Information" name="bookInfo" retain-params="*">

       <field.text caption="Book ID" name="bookId" required="yes" max-
       length="10"/>
       <field.text caption="Name" name="bookName" required="yes"/>
       <field.text caption="Author" name="bookAuthor" required="yes"/>
       <field.select caption="Genre" choices="schema-enum:Book_Type"
       name="bookType" prepend-blank="yes" required="yes"/>
       <field.text caption="ISBN" name="bookISBN" max-length="10"
       required="yes" validate-msg="Please enter an ISBN of the form
       X-XXXX-XXXX-X i.e. with dashes, not spaces. Thank you."
       validate-pattern="/^\d+$/"/>
 </dialog>
 </dialogs>
</xaf>
```

There are two important items in this XML definition; both of them new Value Sources that are being used for the first time.

The first is the `create-data-cmd-heading` value source, which is used to create the heading of the dialog. This value source takes its argument (in this case the string "Book Information") and prepends a word depending on what URL encoded parameters were passed to the dialog. This allows the dialog to have a heading that changes dynamically based on what mode the dialog is being called in. Of the different dialog modes that are possible, this dialog needs only the add, edit, and delete modes which, when active change not only the heading to an appropriate one but also modify the behavior (and, sometimes, the appearance) of the dialog. The Java dialog that will be used in conjunction with this XML dialog will be able to determine the mode that the dialog is in and process the input data differently based on this knowledge. The Sparx Library relies on these three modes of execution for the same dialog to accomplish most of its functionality.

The second is the `schema-enum` value source, which is used to populate the `field.select` with the `id`s and `caption`s in the specified enumeration table (in this case, the `Book_Type` enumeration). This allows the field to automatically reflect any changes made to the enumeration without having to worry about updating the dialog.

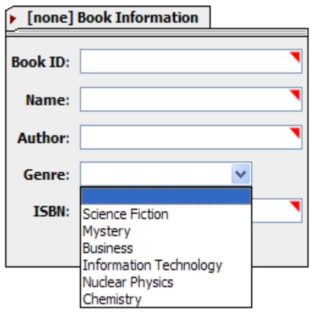


Figure 53: Book Information

Now that you have created the XML dialog, you can test it out in ACE, paying close attention to how the Genre field is able to show you the data stored in the `Book_Type` enumeration earlier. Note that this XML definition names a Java dialog (`app.form.BookInfo`) to complement it. Since that class will not be dealt with until the next section, you should remove the `class` attribute from the `dialog` tag for testing in ACE. Afterwards, you will have to add the directive back to the dialog definition.

The Java Dialog

The Java dialog that complements the XML dialog definition given previously is responsible for roughly half the magic in the Sparx Library. It takes care of processing the different modes of operation of the one dialog defined in the XML as well as all of the database access needed to make the Sparx Library a working application.

Here, then, is the source code for the Java dialog. This should be stored in file named `BookInfo.java` in the `library` subdirectory of the Sparx Library's `WEB-INF/classes/app/form` directory. A step by step explanation of the code will follow immediately after it.

```java
package app.form;

import com.netspective.sparx.xaf.form.Dialog;
import com.netspective.sparx.xaf.form.DialogContext;
import com.netspective.sparx.xaf.task.TaskExecuteException;
import com.netspective.sparx.xaf.sql.StatementManager;
import
 com.netspective.sparx.xaf.sql.StatementNotFoundException;
import com.netspective.sparx.xif.db.DatabaseContext;
import com.netspective.sparx.xif.db.DatabaseContextFactory;

import com.netspective.sparx.xif.dal.ConnectionContext;
import app.dal.table.BookInfoTable;
import app.dal.domain.row.BookInfoRow;
import app.dal.DataAccessLayer;

import app.form.context.library.BookInfoContext;
import com.netspective.sparx.xaf.sql.DmlStatement;

import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.math.BigDecimal;
import java.sql.SQLException;
import java.util.Map;
import java.util.Date;
import java.net.URL;
import java.net.URLEncoder;
i
import java.io.IOException;
import java.sql.PreparedStatement;
import java.util.List;


public class BookInfo extends Dialog
{

    /**
     * This is the class that you do your entire dialog
 validation with
     */
    public boolean isValid(DialogContext dc) {
        return super.isValid(dc);
    }


    public void populateValues(DialogContext dc, int i) {
        // make sure to call the parent method to ensure
default behavior
        super.populateValues(dc, i);

        // you should almost always call dc.isInitialEntry() to
ensure that you're not
        // populating data unless the user is seeing the data
for the first time
        if (!dc.isInitialEntry())
            return;

        // now do the populating using DialogContext methods
        if (dc.editingData() || dc.deletingData()) {
                 nfoContext dcb = (BookInfoContext) dc;
            String bookId =
dc.getRequest().getParameter("bookid");

            BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();

            try {
                ConnectionContext cc =
dcb.getConnectionContext();

                // Grab the information from the BookInfo table
into a new BookInfoRow ...
                        Row bkInfoRow =
bkInfoTbl.getBookInfoById(cc, bookId);

                dcb.setBookId(bkInfoRow.getId());
                dcb.setBookAuthor(bkInfoRow.getAuthor());
                dcb.setBookName(bkInfoRow.getName());
                dcb.setBookType(bkInfoRow.getTypeInt());
                dcb.setBookISBN(bkInfoRow.getIsbn());
            } catch (NamingException ne) {
                ne.printStackTrace();

                se.printStackTrace();
            }

    }



    /**
     *  This is where you perform all your actions. Whatever
you return as the function result will be shown
     * in the HTML

    public void execute(Writer writer, DialogContext dc)
    {
        // if you call super.execute(dc) then you would execute
the <execute-tasks> in the XML; leave it out
        // to override
        // super.execute(dc);

        HttpServletRequest request =
(HttpServletRequest)dc.getRequest();
        String redirectURL = request.getContextPath() +
"/index.jsp";
        String executeStatus;

        // What to do if the dialog is in add mode ...
        if (dc.addingData()) {
            boolean status = processAddAction(writer, dc);
        }

                                                 de ...
        if (dc.editingData()) {
            boolean status = processEditAction(writer, dc);
        }

        // What to do if the dialog is in delete mode ...

            boolean status = processDeleteAction(writer, dc);
        }

        try    {
((HttpServletResponse)dc.getResponse()).sendRedirect(redirectUR
L);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * Process the delete action
     */
    protected boolean processDeleteAction(Writer writer,
DialogContext dc) {
        BookInfoContext dcb = (BookInfoContext) dc;
        BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();
        boolean status = false;
        String bookId = dc.getRequest().getParameter("bookid");

        try {
            ConnectionContext cc = dcb.getConnectionContext();


            BookInfoRow bkInfoRow =
bkInfoTbl.getBookInfoById(cc, bookId);

            status = bkInfoTbl.delete(cc, bkInfoRow);
            cc.commitTransaction();
        } catch (NamingException ne) {
            ne.printStackTrace();
        } catch (SQLException se) {
            se.printStackTrace();
        }

        return status;
    }

    /**
     * Process the update action
     */
    protected boolean processEditAction(Writer writer,
DialogContext dc) {
        BookInfoContext dcb = (BookInfoContext) dc;
        BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();

        String bookId = dc.getRequest().getParameter("bookid");

        try {
            ConnectionContext cc = dcb.getConnectionContext();

            // Create a new BookInfo record and insert it...
            BookInfoRow bkInfoRow =
bkInfoTbl.getBookInfoById(cc, bookId);
            bkInfoRow.setId(dcb.getBookId());
            bkInfoRow.setAuthor(dcb.getBookAuthor());
            bkInfoRow.setName(dcb.getBookName());
            bkInfoRow.setType(dcb.getBookTypeInt());
            bkInfoRow.setIsbn(dcb.getBookISBN());

            status = bkInfoTbl.update(cc, bkInfoRow);
            cc.commitTransaction();
                     ngException ne) {
            ne.printStackTrace();
        } catch (SQLException se) {

        }

        return status;
    }

    /**
     * Process the new data
     */
                                        riter writer,
DialogContext dc) {
        BookInfoContext dcb = (BookInfoContext) dc;
        BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();
        boolean status = false;

        try {
            ConnectionContext cc = dcb.getConnectionContext();

            // Create a new BookInfo record and insert it...
            BookInfoRow bkInfoRow =
bkInfoTbl.createBookInfoRow();
            bkInfoRow.setCrStamp(null);
            bkInfoRow.setId(dcb.getBookId());
            bkInfoRow.setAuthor(dcb.getBookAuthor());
            bkInfoRow.setName(dcb.getBookName());
            bkInfoRow.setType(dcb.getBookTypeInt());
            bkInfoRow.setIsbn(dcb.getBookISBN());

            status = bkInfoTbl.insert(cc, bkInfoRow);
            cc.commitTransaction();
                     ngException ne) {
            ne.printStackTrace();
        } catch (SQLException se) {

        }


        return status;
    }
}
```

Step by Step Explanation

There are five major parts to the source code of the `app.form.BookInfo` class shown above. Each of the five parts is one of the five procedures in the class. These are listed below and explained further down.

METHOD	DESCRIPTION
`populateValues`	This method is first declared in the base Dialog class and this class overrides it. The method is responsible for getting data from a database (or otherwise obtaining it) and populating the fields of the dialog with it. This procedure is called immediately before the dialog is actually rendered onto the screen.
`execute`	This method is first declared in the base Dialog class and this class overrides it. The method is responsible for processing all the data that is input using the dialog. It is also responsible for dealing with all the different modes a dialog might be called in, which is one of the functions it performs in the `app.form.BookInfo`class.

METHOD	DESCRIPTION
`processAddAction`	This method is defined specifically in this class (does not inherit it from the base Dialog class) and is called from execute and is responsible for processing data that is meant to be added to the database.
`processEditAction`	This method is defined specifically in this class (does not inherit it from the base Dialog class) and is called from execute and is responsible for processing a request to update an existing database record.
`processDeleteAction`	This method is defined specifically in this class (does not inherit it from the base Dialog class) and is called from execute and is responsible for processing a request to delete a record from the database.

Dissecting populateValues

```
    public void                              ontext dc, int i) {
        // make sure to call the parent method to ensure
default behavior
        super.po

        // you s                             l dc.isInitialEntry() to
ensure that you're not
        // popul                                                   a
for the first ti
        if (!dc.
            retu

        // now do the populating using DialogContext methods
        if (dc.editingData() || dc.deletingData()) {
            BookInfoContext dcb = (BookInfoContext) dc;
            String bookId =
dc.getRequest().getParameter("bookid");

            BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();


                ConnectionContext cc =
dcb.getConnectionContext();

                // Grab the information from the BookInfo table
into a new BookInfoRow ...
                BookInfoRow bkInfoRow =
bkInfoTbl.getBookInfoById(cc, bookId);

                dcb.setBookId(bkInfoRow.getId());
                                         ow.getAuthor());
                dcb.setBookName(bkInfoRow.getName());
                dcb.setBookType(bkInfoRow.getTypeInt());
                                              sbn());
            } catch (NamingException ne) {
                ne.printStackTrace();
            } catch (SQLException se) {
                             Trace();
            }
        }

```

The work that the `populateValues` procedure accomplishes can be explained simply. First, let Sparx handle populating this dialog as it would if a custom `populateValues` procedure were not there. Having done that, check to see whether the dialog was called in edit or delete mode. If so, get the value of the URL encoded parameter `bookid` and use it to fetch the record (or row) for the corresponding book from the

`Book_Info` table. Then take every field in the newly fetched row and insert its value into the corresponding field in the context of the current dialog.

```
super.populateValues(dc, i);
```

This line calls the `populateValue` procedure in this dialog's parent class which is the default Sparx dialog class.

```
if (!dc.isInitialEntry())
return;
```

These lines check to see whether this is the first time that this dialog has been entered in this run. If not, then it is possible that the user previously entered data that should be displayed here (courtesy of the `populateValue` method in the parent class). In that case, there is no more work left to be done here and the method ends here.

```
if (dc.editingData() || dc.deletingData()) {
```

This line causes the rest of the block to execute if (and only if) the dialog is in either edit mode or delete mode. In both modes, it will pull up the record for the book from the database and populate the dialog. The purpose behind the population, however, is different for the two modes. In edit mode it is done so the user can see the existing values stored for the book's record and modify it. In delete mode it is done so the user can verify that he is deleting the book record that he actually intends to delete.

```
BookInfoContext dcb = (BookInfoContext) dc;
```

This line takes the generic dialog context (`dc`) and casts it to a dialog context geared specifically to the `BookInfo` dialog. This specific dialog context is automatically generated by Sparx from the XML definition of the dialog. This new dialog context provides easier and stronger typed access to the individual fields that make up the dialog.

```
String bookId = dc.getRequest().getParameter("bookid");
```

This line gets the value of the `bookid` parameter that is passed into the dialog as a URL encoded parameter.

```
      BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();
```

This line instantiates a Java representation of the BookInfo table as stored in the database. The BookInfoTable is a class generated automatically by Sparx from the XML definition of the database schema. Similar classes exist for all parts of the XML schema. Collectively, all these generated classes are built upon and referred to as the Sparx Data Access Layer or DAL. As you can see in the rest of the source code, the DAL makes programming database access much easier than conventional JDBC methods.

```
ConnectionContext cc = dcb.getConnectionContext();
```

This line gets the connection context for the current dialog context. This connection context allows you to perform database access to manipulate and/or read data as necessary.

```
BookInfoRow bkInfoRow =
  bkInfoTbl.getBookInfoById(cc, bookId);
```

This line uses the connection context to read in the record (or row) from the `Book_Info` table that corresponds to the `bookid` passed in as a URL encoded parameter. The class `BookInfoRow` is the next lower level class from the `BookInfoTable` and is part of the automatically generated DAL.

```
 dcb.setBookId(bkInfoRow.getId());
 dcb.setBookAuthor(bkInfoRow.getAuthor());
 dcb.setBookName(bkInfoRow.getName());
 dcb.setBookType(bkInfoRow.getTypeInt());
 dcb.setBookISBN(bkInfoRow.getIsbn());
```

These lines all perform the same basic function. Each reads in the value of a field and inserts that into the corresponding field in the dialog using the dialog context. These lines, in effect, populate the dialog using the data that has already been read in from the database.

If the column names in the database are identical to the form field names, you can use a special method that will simply populate the DialogContext's field values with the Row's column values that match the names of the columns.

```
bkInfoRow.setData(dcb);  // this would eliminate the dcb.setXXX calls
```

Dissecting execute

```
    public void execute(Writer writer, DialogContext dc)
    {
        // if you call super.execute(dc) then you would execute the <execute-tasks> in the XML; leave it out
        // to override
        // super.execute(dc);

        HttpServletRequest request = (HttpServletRequest)dc.getRequest();
        String redirectURL = request.getContextPath() + "/index.jsp";
        String executeStatus;

        // what to do if the dialog is in add mode ...
        if (dc.addingData()) {
            boolean status = processAddAction(writer, dc);
        }

                                              mode ...
        if (dc.editingData()) {
            boolean status = processEditAction(writer, dc);
        }

                                      in delete mode ...
        if (dc.deletingData()) {
               lean status = processDeleteAction(writer, dc);
        }

        try    {
            ((HttpServletResponse)dc.getResponse()).sendRedirect(redirectURL);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

The execute method has one simple purpose: determine what mode the dialog was called in and dispatch this execute request to the appropriate method. Once that is done, it redirects the user to the "home" of the Sparx Library. In essence, then, it allows you to add, edit or delete books and, once you're done, it redirects you to the Sparx Library's home page.

```
HttpServletRequest request =
(HttpServletRequest)dc.getRequest();
String redirectURL = request.getContextPath()
  + "/index.jsp";
```

The first of these two lines obtains a HttpServletRequest object to gather information about the Sparx Library. This information is provided to the HttpServletRequest object by the Resin application server. The second line uses that object to find out what the Library uses as its context path. In this case, it is the `/library` prefix that is a part of all URLs belonging to the Sparx Library application. The `redirectURL`, which is the URL where the dialog will go to after it's done processing all input data, is created by adding `/index.jsp` to the end of this context path. In this way, if you happen to change the context path in the application server, you do not have to come back and change it in the Java dialog; the change will be picked up automatically.

```
if (dc.addingData())
boolean status = processAddAction(writer, dc);

// What to do if the dialog is in edit mode ...
if (dc.editingData())
boolean status = processEditAction(writer, dc);

// What to do if the dialog is in delete mode ...
if (dc.deletingData())
boolean status = processDeleteAction(writer, dc);
```

Each of these three statements performs the same function. Each checks to see whether the dialog is in a particular mode (add, edit or delete) and, if so, calls an appropriate method with the same parameters as the execute method was called. These methods return a status which determines whether they were successful or not.

```
((HttpServletResponse)
 dc.getResponse()).sendRedirect(redirectURL);
```

Finally, this line is responsible for redirecting the user's web browser to the URL stored in the `redirectURL` variable.

Dissecting processEditAction

All three of the processor methods, namely processAddAction, processEditAction and processDeleteAction, have very similar structures. Therefore, it will suffice to dissect one to understand all three.

```
    protected boolean processEditAction(Writer writer,
DialogContext dc) {
        BookInfoContext dcb = (BookInfoContext) dc;
        BookInfoTable bkInfoTbl =
DataAccessLayer.instance.getBookInfoTable();
        boolean status = false;
        String bookId = dc.getRequest().getParameter("bookid");

        try {
            ConnectionContext cc = dcb.getConnectionContext();
```

```

             // Create a new BookInfo record and insert it...
             BookInfoRow bkInfoRow =
    bkInfoTbl.getBookInfoById(cc, bookId);
             bkInfoRow.setId(dcb.getBookId());
             bkInfoRow.setAuthor(dcb.getBookAuthor());
             bkInfoRow.setName(dcb.getBookName());
             bkInfoRow.setType(dcb.getBookTypeInt());
             bkInfoRow.setIsbn(dcb.getBookISBN());

             status = bkInfoTbl.update(cc, bkInfoRow);
             cc.commitTransaction();

             ne.printStackTrace();
         } catch (SQLException se) {
             se.printStackTrace();
         }

         return status;
     }
```

After instantiating a dialog-specific dialog context, a Sparx DAL representation of the `Book_Info` table and parsing out the URL encoded `bookid` parameter, this method gets to the meat of its operation. The lines of code corresponding to this central operation are shown below.

```
// Create a new BookInfo record and insert it...
BookInfoRow bkInfoRow =
bkInfoTbl.getBookInfoById(cc, bookId);
bkInfoRow.setId(dcb.getBookId());
bkInfoRow.setAuthor(dcb.getBookAuthor());
bkInfoRow.setName(dcb.getBookName());
bkInfoRow.setType(dcb.getBookTypeInt());
bkInfoRow.setIsbn(dcb.getBookISBN());
```

This chunk of code first fetches from the `Book_Info` table the row that corresponds to the passed-in value of the `bookid` parameter. It then proceeds to set the value of each field in this row of data to the value of the corresponding field as passed in to the dialog. Thus any fields that the user modified in the dialog will now overwrite the value of those fields as stored in the `Book_Info` database table.

```
bkInfoRow.populateDataByNames(dcb);  // bkInfoRow.setXXX not required
```

If the column names in the database are identical to the form field names, you can use a special method that will simply populate the Row's column values with the values of the fields in the DialogContext that match the names of the columns.

```
status = bkInfoTbl.update(cc, bkInfoRow);
cc.commitTransaction();
```

Once the table row is updated in memory, it is then updated in the database by calling the update function on the Sparx DAL representation of the `Book_Info` table and passing in the updated row of values as a parameter. The final step is to finalize this update by committing it to the database.

After all is said and done, you have used a little less than 130 lines of Java code (including lines that simply contain closing braces but excluding all comments and blank lines) to create a very robust dialog that can add data to a database as well as edit or delete data that already exists in the database. Even with such a meager line count, your Java code is still very readable thanks to the power and beauty of the Sparx DAL.

Unit Testing with ACE

After saving this Java dialog in the `WEB-INF\classes\app\form\BookInfo.java` file of the Sparx Library, you should launch a web browser and test out the new dialog in ACE. Using the default "Test this dialog" links, it should not feel much different until you press the OK button. At this point, since you have not created an `index.jsp` file in your `Site` directory, pressing OK will redirect you to a non-existent location.

However, it is still possible to use ACE to test out the new dialog albeit in a slightly unconventional manner. You can manually provide to the dialog all the information it needs for proper functioning using the following steps.

This first test puts the dialog through its paces to add a book to the Library. For this, you need to put the dialog into the Add mode so that the execute method can take care of adding the data you enter into the database. To achieve this, follow the steps outlined below.

- Open a web browser and go to the Library's ACE[19]. Choose Dialogs from the Application menu and click on the name of the only dialog listed there. This should bring you to a field list for this dialog. Click on the "Test this dialog" link at the top left of this page.

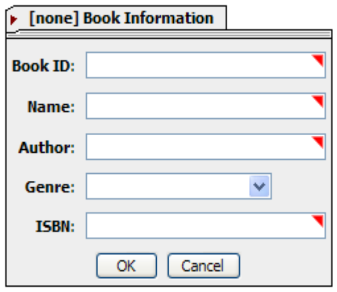


Figure 54: Testing The Dialog

- In the new window that opens up (the test window) notice that the first word in the dialog heading is "[none]". This implies that the `create-data-cmd-header` Value Source (used in the dialog heading) has not detected whether the dialog has been called with a valid mode parameter or not.
- To call the dialog in Add mode, change the URL shown in the Address bar of the browser (where the URL of the current page is shown) by appending `?data_cmd=add` to it. Press enter or have the test window go to the new URL for testing the dialog.

[19] If you are evaluating Sparx online, you can access the Sparx Collection ACE at http://developer.netspective.com/samples/library/ace

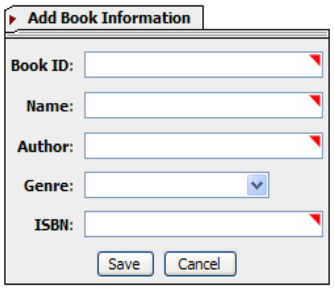
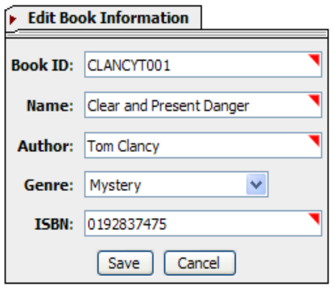


Figure 55: The Dialog In Add Mode

- Notice that now the dialog heading reads “Add Book Information”. This implies that the `create-data-cmd-header` Value Source has successfully detected the current mode the dialog was called in and added an appropriate prefix to the heading.
- Add a book to the Library’s database by filling out the dialog and pressing OK. Make sure you remember the Book ID you enter since you will be using this to verify that things went ok.
- You will be redirected to the non-existent `index.jsp` page. This will cause your browser to show you an error. This behavior is expected.

The second test puts the dialog into edit mode to verify that the record you created does, in fact, exist in the database and can therefore be edited. To achieve this, follow the steps outlined below.

- From the Library’s ACE[20] test the only dialog listed in the Application Dialogs page.



Book ID:	CLANCYT001
Name:	Clear and Present Danger
Author:	Tom Clancy
Genre:	Mystery
ISBN:	0192837475

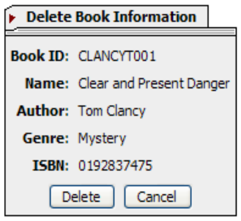
Figure 56: Dialog In Edit Mode

- In the new window that pops up (the test window) change the URL by appending `?data_cmd=edit&bookid=YOURBOOKID` to it. Make sure you replace the word `YOURBOOKID` with the Book ID you chose while testing the dialog’s Add mode above. Thus, if the book you added to the Library earlier had a Book ID of TESTBOOKID, you will append `?data_cmd=edit&bookid=CLANCYT001` to the URL shown in the test window.
- If your previous test to add a book to the Library was successful, this new test should be able to pull up the record of that book (by using its Book ID) and allow you to edit the record. Additionally, you will notice that the heading now says “Edit Book Information”.
- Change one of the fields and press Save. Your browser should still give you an error since you do not have an index.jsp in place. This is expected.

[20] If you are evaluating Sparx online, you can access the Sparx Collection ACE at http://developer.netspective.com/samples/library/ace.

The final test puts the dialog into delete mode to verify that the edits you made in the previous test succeeded as well as to verify that deleting an existing record work. To achieve this, follow the steps outlined below.

- From the Library's ACE test the only dialog listed in the Application Dialogs page.


Delete Book Information

Book ID: CLANCYT001
Name: Clear and Present Danger
Author: Tom Clancy
Genre: Mystery
ISBN: 0192837475

Figure 57: Delete Book Result

- In the new window that pops up (the test window) change the URL by appending `?data_cmd=delete&bookid=YOURBOOKID` to it. Make sure you replace the word `YOURBOOKID` with the Book ID you chose while testing the dialog's Add mode above.
- If your previous test to edit an existing book's information was successful, this new test should be able to pull up the modified record of that book and ask you whether you want to delete it or not.
- The most obvious difference between the add/edit mode(s) and the delete mode is the fact that all the fields in the delete mode are static and, therefore, read-only. The other difference, which you must have noticed by now, is that the heading now says "Delete Book Information".
- Press the Delete button to delete this record from the Sparx Library database. The browser will, as before, fail to load the non-existent index.jsp file. This is expected.
- You can follow up on this test by attempting to retry the steps you used to test editing an existing record. Since the book is now deleted from the database, an attempt to edit that record will yield nothing. For now this can serve as a verification of the deletion of the record from the database.

Creating the SQL (Data Layer)

The Sparx Library needs SQL for every bit of its operation, whether implicitly like in the case of the Sparx DAL or explicitly like in the case that will be demonstrated shortly. Sparx supports three different forms of SQL: static, dynamic, and DAL (Data Access Layer).

A static SQL statement is merely an encapsulation of a regular SQL statement within the XML definition required by Sparx to interpret it. This type of statement supports SQL bind parameters making it more powerful than it might seem at first. The utility and ease of use of this type of SQL construct will be demonstrated in this section.

Dynamic SQL is supported in the form of Query Definitions. These are XML definitions of a set of fields and their relationships. When used in an application, these allow an end-user to create and execute any custom SQL statement that the Query Definition allows. The power of this SQL construct will be demonstrated in the next section.

Introduction to Static SQL Libraries

Static SQL libraries are analogous to Dialog packages stored in the `dialogs.xml` file inside the `ui` directory of the Library. Static SQL libraries are stored in the `statements.xml` file in the `sql` directory of the Library. The `sql` directory is directly under the `WEB-INF` directory of an application.

Each library is organized in a hierarchical manner. The top level is `xaf`, just like in a Dialog package. Immediately under that is the `sql-statements` level which is analogous to the `dialogs` level in the `dialogs.xml` file. One `statements.xml` file can have multiple `sql-statements` packages. Below the `sql-statements` level lie the actual SQL statements enclosed in `statement` tags.

To get a better idea of what this structure looks like, you can look at the exact set of static SQL statements used by the Sparx Library. Following this XML definition of the Library's SQL statements is a detailed explanation that may help you understand some of the constructs better.

```
<?xml version="1.0"?>

<xaf>
 <sql-statements package="library">
 <statement name="sel_all_books">
       select
              book_info.id,
              book_info.name,
              book_type.caption as genre,
              book_info.author,
              book_info.isbn,
              book_info.id
       from
              book_info,
              book_type
       where
              book_info.type = book_type.id;

 <report>
       <column align="center" heading=" " index="0"
              output="&lt;a
       href=&quot;bookInfo.jsp?data_cmd=edit&amp;bookid=${0}&quot;&gt;
       edit&lt;/a&gt;"/>
       <column index="1" output="&lt;a
       href=&quot;viewBook.jsp?bookid=${0}&quot;&gt;${1}&lt;/a&gt;"/>
       <column heading="Genre" index="2"/>
       <column align="center" heading=" " index="5" output="&lt;a
       href=&quot;bookInfo.jsp?data_cmd=delete&amp;bookid=${0}&quot;&g
       t;delete&lt;/a&gt;"/>
 </report>
 </statement>

 <statement name="sel_one_book">
       select

                          me,
              book_type.caption as genre,

              book_info.isbn
       from


       where
              book_info.type = book_type.id and
       book_info.id = ?;
 <params>
 <param value="request:bookid"/>
 </params>

 <report>
      <column heading="Action" index="0" output="&lt;a
      href=&quot;bookInfo.jsp?data_cmd=edit&amp;bookid=${0}&quot;&gt;
      edit&lt;/a&gt;"/>
      <column heading="Name" index="1"/>
      <column heading="Genre" index="2"/>
      <column heading="Author" index="3"/>
 </report>
 </statement>
 </sql-statements>
</xaf>
```

Step by Step Explanation

The XML shown above defines two SQL statements: `sel_all_books` and `sel_one_book`. Both are very similar with the one difference being that `sel_one_book` uses a SQL bind parameter which needs some explanation. Since `sel_all_books` covers a little more material, it is best to explain that in detail and end with a brief explanation of how the SQL bind parameter is used in `sel_one_book`.

```
 <statement name="sel_all_books">
      select
             book_info.id,
             book_info.name,
             book_type.caption as genre,
             book_info.author,
             book_info.isbn,

      from
             book_info,
             book_type
      where
             book_info.type = book_type.id;
```

These initial few lines declare the statement name and body. The body contains the full SQL statement and, as you can see, can be indented according to your aesthetic needs. However, the power of the SQL statement is shown in the `report` part of the statement.

```
 <report>
      <column align="center" heading=" " index="0"
      output="&lt;a
      href=&quot;bookInfo.jsp?data_cmd=edit&amp;bookid=${0}&quot;&gt;
      edit&lt;/a&gt;"/>
      <column index="1" output="&lt;a
      href=&quot;viewBook.jsp?bookid=${0}&quot;&gt;${1}&lt;/a&gt;"/>
      <column heading="Genre" index="2"/>
      <column align="center" heading=" " index="5" output="&lt;a
      href=&quot;bookInfo.jsp?data_cmd=delete&amp;bookid=${0}&quot;&g
      t;delete&lt;/a&gt;"/>
 </report>
```

Before explaining this section of the code, you should know that what appears to be a jumble of letters in the code is actually HTML encoded in a way that would be suitable for use in XML. After decoding this for explanation purposes only, the code above looks like the one shown below. Keep in mind that the XML shown below is **not** valid XML; only its encoded representation shown above is valid and can be used in your applications. The explanations below will show both the encoded and decoded versions of this XML for easier understanding.

```
<report>
      <column align="center" heading=" " index="0"
              output="<a
      href=”bookInfo.jsp?data_cmd=edit&bookid=${0}”>edit</a>"/>
      <column index="1" output="<a
      href=”viewBook.jsp?bookid=${0}”>${1}</a>"/>
      <column heading="Genre" index="2"/>
      <column align="center" heading=" " index="5" output="<a
      href=”bookInfo.jsp?data_cmd=delete&bookid=${0}”>delete</a>"/>
</report>
```

Explaining Reports

The XML you just saw is what is known in Sparx terminology as a report. A report is a way to customize the output of a SQL statement on the page. Please note that each SQL statement may have any number of `<report>` tags as necessary. That way, a single statement can be viewed differently depending upon what context it's being called in. The default output of a SQL statement when tested in ACE or embedded in a JSP page is to display a table such that each row is a record and each column is a field in that record.

Whereas it is possible to change the layout of report completely, the basic concept remains one of rows and columns even for a layout where those do not make sense. Therefore, in every SQL report you will notice the use of column tags that are used to customize the appearance of a particular column or, more accurately, a particular field.

A column tag can take many different attributes, the common ones having been demonstrated in the XML shown above. What follows is a brief explanation of each attribute used in the column tags shown above.

- `align`: The `align` attribute specifies the alignment of the field in it's table column. This is similar to the align attribute used in HTML tables.
- `heading`: The `heading` attribute specifies the title for the column. This heading is placed above the first row of the table that is displayed. In this case a heading of " " would make Sparx make the heading appear blank.
- `index`: The `index` is an attribute that is present in all column tags implicitly. If you do not specify any index attributes in a series of report columns, Sparx assumes the first column has an index of 0 and the next one has an index of 1 and so on. If within this sequence of columns with implicit index values you explicitly specify the index for one of the columns, all indices between that column and the previous column are discarded and the sequence of implicit indexes continues from the value you specified explicitly.

```
<report>
 <c
 <c
 <column heading=”column 2”/>
   olumn heading=”column 5” index=”5”/>
 <c
</r
```

The XML snippet shown above illustrates the modus operandi of index in an easier to understand manner. Since columns 0 through 2 do not explicitly specify an index attribute, they are implicitly assigned indices from 0 to 2. This means that column 0 has an implicit index of 0 while column 2 has an implicit index of 2. Column 5, on the other hand, has an index value of 5 that is explicitly specified. Therefore, this column tag will apply to the sixth column (sixth because columns are indexed from 0) of the report table. The column immediately after it has no index value specified explicitly. However, since the implicit index sequence was disrupted by the explicit index value, the new sequence will continue from the value specified in column 5. This would make the implicit value of the next column equal to 6. Thus this next column and its attributes will apply to the sixth column in the table.

The last point to keep in mind regarding columns is that you can have more column definitions than the number of fields that your statement returns. The extra columns are just tacked on at the end of the table.

- ♦ `output`: The `output` attribute allows you to determine exactly how a column will be displayed. You can insert HTML (albeit escaped so it does not interfere with XML tags), JavaScript or anything similar. The value of the `output` attribute is displayed as is in place of the original contents of the affected column of the table. However, there is one thing special about the `output` attribute: it allows you to use what can be called placeholder variables in the output string. These variables have the following format: `${columnNumber}`, where `columnNumber` can be 0 to one less than the maximum number of columns in the output of the SQL. Sparx replaces these variables with the value of the field in the specified column of the SQL statement.

```
<co
 heading=" "
 index="0"
 ou
       ”bookInfo.jsp?data_cmd=edit&bookid=${0}”>edit</a>"/>
```

This is a snippet of the decoded report shown above. You can notice the `${0}` near the end of the output attribute. By looking at the complete statement, you will also realize that the 0th column of the output of the SQL statement is the `book_info.id` field which is the Book ID. Therefore, instead of just displaying the Book ID in the 0th column, this report column transforms the text in that column to a link (named “edit”) that accesses the `bookInfo.jsp` file (which will contain our `library.bookInfo` dialog) in the Edit mode (`data_cmd=edit`) and passes in the Book ID as a URL encoded parameter (`bookid=${0}`).

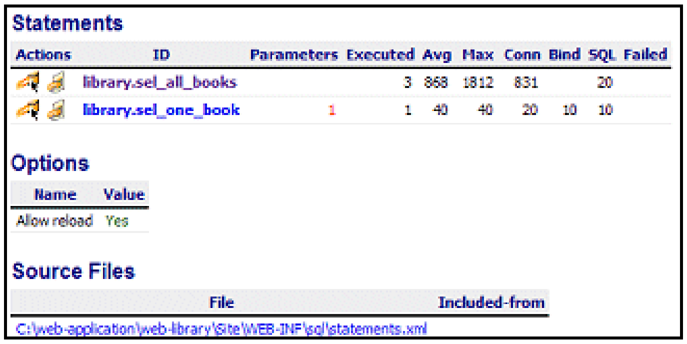
```
<column align="center"
 heading=" "
 index="5"
 ou
       ”bookInfo.jsp?data_cmd=delete&bookid=${0}”>delete</a>"/>
```

Similarly, this XML snippet from the sixth column (which is one more than the number of columns output by the SQL statement) creates a link to delete to the same `bookInfo.jsp` file but this time in `delete` mode and with the Book ID as a URL encoded parameter.

You will remember that both these scenarios are things that your Java dialog is not only prepared for but indeed counts on. The output format of the 0th column allows you to delete the record shown in that row of the table while the output format of the 6th column allows you to delete the record. With the editing and deletion of a book’s information taken care of, the only thing left is to allow people to add books to the Library and to search the Library for particular books. So the plan unfolds.

Unit Testing in ACE

Having saved the XML source you just saw in the `statements.xml` file in the Library's `WEB-INF\sql` directory, you will be able to see the statements in action by going to the Library's ACE[21]. After logging in, choose the *SQL Statements* item from the *Database* menu to being testing. You will only be able to test out `library.sel_all_books` properly at this time.



Statements

Actions	ID	Parameters	Executed	Avg	Max	Conn	Bind	SQL	Failed
	library.sel_all_books		3	868	1812	831		20	
	library.sel_one_book	1	1	40	40	20	10	10	

Options

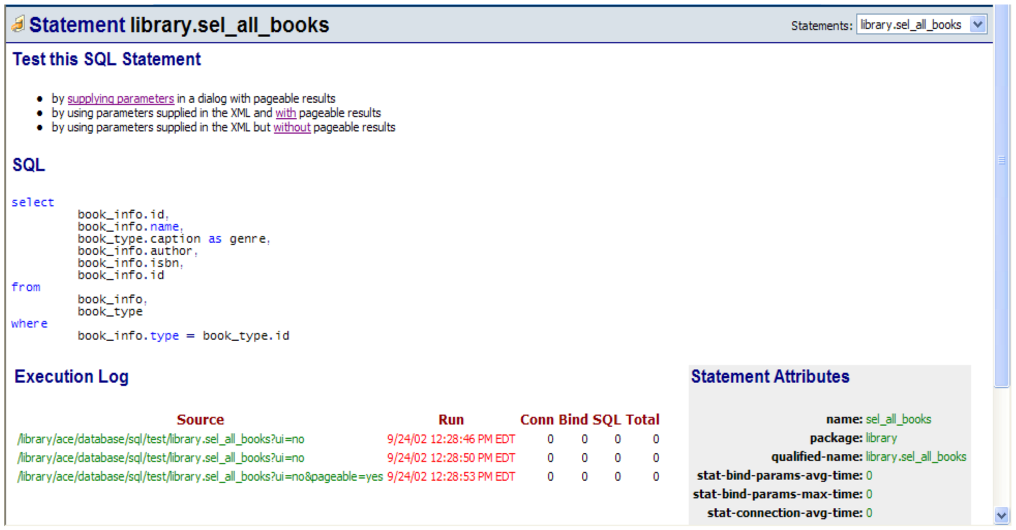
Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\web-application\web-library\Site\WEB-INF\sql\statements.xml	

Figure 58: Unit Testing In ACE

♦ Click on the name of the library.sel_all_books statement to see details about the statement. These include details about the statement as well as about database benchmarks of this statement like average time to execute etc.



Statement library.sel_all_books

Statements: library.sel_all_books

Test this SQL Statement

- by supplying parameters in a dialog with pageable results
- by using parameters supplied in the XML and with pageable results
- by using parameters supplied in the XML but without pageable results

SQL

```
select
          book_info.id,
          book_info.name,
          book_type.caption as genre,
          book_info.author,
          book_info.isbn,
          book_info.id
from
          book_info,
          book_type
where
          book_info.type = book_type.id
```

Execution Log

Source	Run	Conn	Bind	SQL	Total
/library/ace/database/sql/test/library.sel_all_books?ui=no	9/24/02 12:28:46 PM EDT	0	0	0	0
/library/ace/database/sql/test/library.sel_all_books?ui=no	9/24/02 12:28:50 PM EDT	0	0	0	0
/library/ace/database/sql/test/library.sel_all_books?ui=no&pageable=yes	9/24/02 12:28:53 PM EDT	0	0	0	0

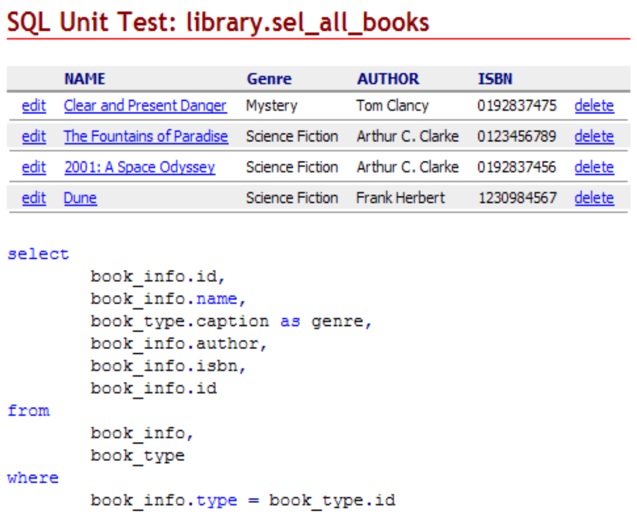
Statement Attributes

name:	sel_all_books
package:	library
qualified-name:	library.sel_all_books
stat-bind-params-avg-time:	0
stat-bind-params-max-time:	0
stat-connection-avg-time:	0

Figure 59: Statement Details

♦ Use the "Test this SQL Statement by supplying parameters" link to run the SQL and test its functionality.

[21] If you are evaluating Sparx online, you can access the Sparx Collection ACE at http://developer.netspective.com/samples/library/ace



SQL Unit Test: library.sel_all_books

	NAME	Genre	AUTHOR	ISBN	
edit	Clear and Present Danger	Mystery	Tom Clancy	0192837475	delete
edit	The Fountains of Paradise	Science Fiction	Arthur C. Clarke	0123456789	delete
edit	2001: A Space Odyssey	Science Fiction	Arthur C. Clarke	0192837456	delete
edit	Dune	Science Fiction	Frank Herbert	1230984567	delete

```
select
        book_info.id,
        book_info.name,
        book_type.caption as genre,
        book_info.author,
        book_info.isbn,
        book_info.id
from
        book_info,
        book_type
where
        book_info.type = book_type.id
```

Figure 60: Unit Test of Statement

- The window that pops up should contain the SQL for the statement and, if everything goes well, a table at the bottom showing the results of executing the statement.
- Pay careful attention to all the links shown in the table. Move your mouse over them to find out what page they are pointing to. Compare this with the XML for this statement's report to get an idea of how the source led to the result.

Creating Query Definitions (Dynamic SQL)

Query definitions are one of the most powerful features that Sparx provides developers. Using query definitions a developer wields extreme flexibility and power with an ease rivaled by few, if any, other components of Sparx. Because of their uniqueness, query definitions are difficult to explain independently. The best analogy is to compare a query definition to a database view.

Just like a database view, a query definition requires actual tables with actual fields to exist. It is able to take fields from disparate tables (while maintaining all join conditions that are necessary for those tables). Using custom dialogs created by developers (you can have multiple dialogs associated with one query definition), users can query the fields in the query definition for data.

That brings up the question of why we need query definitions. The Sparx Library has, at the moment, most of the functionality originally intended for it. All that is needed is a little glue in the form of JSP pages (which this tutorial already alludes to in several places) and the application should be done. The only bit of functionality that is left is that of being able to *search* the Library for books matching any criteria the user chooses. For a small collection, the result of the `library.sel_all_books` is small enough to browse manually for the information pertaining to a book. For larger numbers of books the power of a query definition can come in very handy to search the database for all books matching any criteria specified by the end user.

A possible query definition that can be used to implement such a search function is shown below. Whereas you can design your own query definition, you should use this exact query definition while following this tutorial.

```
<query-defn id="searchBooks">
 <field id="book_id" caption="Book ID" join="Book_Info" column="id">
 <report heading="ID"/>
 </field>

 <field id="book_name" caption="Name" join="Book_Info"
      column="name"/>
 <field id="book_author" caption="Author" join="Book_Info"
      column="author"/>
 <field id="book_genre" caption="Genre" join="Book_Type" column="id"
      column-expr="Book_Type.caption"/>
 <field id="book_isbn" caption="ISBN" join="Book_Info"
      column="ISBN"/>

 <join id="Book_Info" table="Book_Info" condition="Book_Info.type =
      Book_Type.id" imply-join="Book_Type"/>
 <join id="Book_Type" table="Book_Type"/>

 <select-dialog name="searchDialog" heading="Search Books">
      <field.text query-field="book_id"/>
      <field.select query-field="book_genre" choices="schema-
      enum:Book_Type" prepend-blank="yes"/>
      <field.text query-field="book_name"/>
      <field.text query-field="book_author"/>
      <field.text query-field="book_isbn"/>

      <select heading="Book Search Results">
             <display field="book_id"/>
             <display field="book_genre"/>
             <display field="book_name"/>
             <display field="book_author"/>
             <display field="book_isbn"/>

             <condition field="book_id" allow-null="no"
      comparison="starts-with" value="form:book_id" connector="and"/>
             <condition field="book_genre" allow-null="no"
      comparison="contains" value="form:book_genre" connector="and"/>
             <condition field="book_name" allow-null="no"
      comparison="contains" value="form:book_name" connector="and"/>
             <condition field="book_author" allow-null="no"
      comparison="contains" value="form:book_author"
      connector="and"/>
             <condition field="book_isbn" allow-null="no"
      comparison="contains" value="form:book_isbn" connector="and"/>
 </select>
 </select-dialog>
</query-defn>
```

Step by Step Explanation

The first thing you should notice about a query definition is that it is a hierarchical structure. A `query-defn` tag contains other tags itself but it also contains `select-dialog` tags which are their own entities. A list of the various tags and constructs that appear in the query definition is shown below along with explanations of the functionality of each.

```
<field id="book_id" caption="Book ID" join="Book_Info" column="id">
 <report heading="ID"/>
</field>

<field id="book_name" caption="Name" join="Book_Info" column="name"/>
<field
      column="author"/>
```

```
<field id="book_genre" caption="Genre" join="Book_Type" column="id"
      column-expr="Book_Type.caption"/>
<field id="book_isbn" caption="ISBN" join="Book_Info" column="ISBN"/>
```

♦ `field`: The `field` tag is used to define the fields that will be available to the query definition. Each `field` tag has, among other attributes, a `join` attribute which determines how the query definition will make that field available to itself and its components. The value of the `join` attribute is a reference to a `join` tag later on in the query definition.

```
<field id="book_id" caption="Book ID" join="Book_Info" column="id">
 <report heading="ID"/>
</field>
```

Additionally, a `field` tag can have sub-tags like the `report` tag in the source shown above. This `report` tag functions similarly to the `report` tag inside a `statement` except it applies to just the one field: the one it is nested under.

In the context of the final SQL statement that Sparx generates, the `field` tags become a part of the `select` clause, i.e. the part of a `select` statement that determines which fields need to be returned.

```
<join id="Book_Info" table="Book_Info" condition="Book_Info.type =
      Book_Type.id" imply-join="Book_Type"/>
<join id="Book_Type" table="Book_Type"/>
```

♦ `join`: The `join` tag is used to let the query definition know of the list of tables and (database) joins that is necessary to be able to get all the fields that are a part of the query definition. In other words, while the `field` tags specify the “What” to the query definition, the `join` tags specify the “How”. The “What” cannot exist without the “How”.

In the context of the final SQL statement that Sparx generates, the `join` tags become a part of the `where` clause to signify the relationships between tables (if any exists).

```
<select-dialog name="searchDialog" heading="Search Books">
 <field.text query-field="book_id"/>
 <field.select query-field="book_genre" choices="schema-
      enum:Book_Type" prepend-blank="yes"/>
 <f
 <f
 <f

 <select heading="Book Search Results">
 <display field="book_id"/>
 <d
 <d
 <d
 <display field="book_isbn"/>

 <condition field="book_id" allow-null="n     mparison="starts-with"
                                            "/>
 <condition field="book_genre" allow-null="no" comparison="contains"
      value="form:book_genre" connector="and"/>
 <c
      value="form:book_name" connector="and"/>
 <c                                                           ns"
      value="form:book_author" connector="and"/>
 <condition field="book_isbn" allow-null="no" comparison="contains"
      value="form:book_isbn" connector="and"/>
 </select>
</s
```

♦ `select-dialog`: The `select-dialog` tag is essentially a dialog embedded inside a query definition. The purpose of this dialog is to provide a user interface for the query definition that can be used to embed the query definition in your own JSP pages. One query definition can have multiple `select-dialogs` defined. If the `fields` and `joins` can be considered the fuel for the Sparx query definition engine, `select-dialogs` can be considered the steering wheel.

```
<field.text query-field="book_id"/>
<field.select query-field="book_genre" choices="schema-
      enum:Book_Type" prepend-blank="yes"/>
<field.text query-field="book_name"/>
<field.text query-field="book_author"/>
<field.text query-field="book_isbn"/>
```

The first part of a `select-dialog` is a declaration of all the fields (and their corresponding UI representations) that the `select-dialog` will use. You will notice that whereas most of the fields declared in the `select-dialog` are going to be represented in the UI as `field.texts`, the genre field is declared to be a `field.select`. The `query-field` attribute determines what field (as declared in the query definition) a `select-dialog field` is referring to.

```
<select heading="Book Search Results">
 <displ
 <display field="book_genre"/>
 <display field="book_name"/>
 <display field="book_author"/>
 <display field="book_isbn"/>

 <c                                                             h"
      value="form:book_id" connector="and"/>
 <c                                                           ins"
      value="form:book_genre" connector="and"/>
 <c                                              son="contains"
      value="form:book_name" connector="and"/>
 <condition field="book_author" allow-null="no" comparison="contains"
      value="form:book_author" connector="and"/>
 <condition field="book_isbn" allow-null="no" comparison="contains"
      value="form:book_isbn" connector="and"/>
</select>
```

The second part of a `select-dialog` is a `select` section which has two parts in itself. The `select` component is what determines which of the `select-dialog`'s declared fields are actually displayed in the dialog for this `select-dialog`. It also determines how each field will be interpreted by the query definition engine once the dialog is submitted.

```
 <d
 <display field="book_genre"/>
 <display field="book_name"/>
 <display field="book_author"/>
 <display field="book_isbn"/>
```

These `display` tags have one attribute, `field`, which is the name of the field to make a visible part of the select-dialog's UI. The value of the `field` attribute points to the name of a field declared in the main query definition.

```
 <condition field="book_id" allow-null="no" comparison="starts-with"
      value="form:book_id" connector="and"/>
 <condition field="book_genre" allow-null="no" comparison="contains"
      value="form:book_genre" connector="and"/>
 <condition field="book_name" allow-null="no" comparison="contains"
      value="form:book_name" connector="and"/>
 <condition field="book_author" allow-null="no" comparison="contains"
      valu
           field="book_isbn" allow-null="no" comparison="contains"
      value="form:book_isbn" connector="and"/>
```

These `condition` tags determine how the data input from the `select-dialog`'s UI will be interpreted by the query definition engine. Each `condition` tag has a few attributes that are explained below.

The `field` attribute determines which query definition field this condition is referring to.

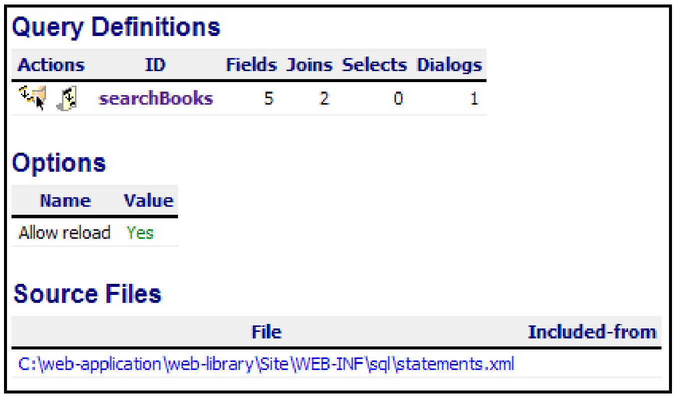
The `allow-null` attribute determines the behavior of the query definition engine if this field is left empty when the dialog is submitted. If this attribute has a value of "yes", the select generated by the query definition engine will include this field but will have it set to the value null. This may be a desired consequence in other applications you write but in the case of the Library, this is not needed. If set to "no", which is the case for the Library, the select generated will omit the field if the corresponding dialog field happens to be empty.

The `comparison` attribute determines what criterion will be used to match the field values stored in the database against the value entered by the user in the dialog. A `comparison` of, say, "starts-with" tells the query definition engine to match all those records in the database whose values for this field starts with the value entered by the user. Similarly a se of contains tells the query definition engine to match all those records in the database whose values for this field contain the value entered by the user.

Finally the `connector` attribute determines how many `field` criteria each record in the database has to match before it is selected. If all the fields have a `connector` value of "and", a record would have to match all the `field` criteria to be selected. However, a `connector` value of `or` would allow a record to be selected if it matched any of the field criteria.

Unit Testing in ACE

First make sure you save the entire query definition (everything between the starting and ending `query-defn` tags) by adding it to your `statements.xml` file as a child of (or inside) the `xaf` level.



Query Definitions

Actions	ID	Fields	Joins	Selects	Dialogs
	searchBooks	5	2	0	1

Options

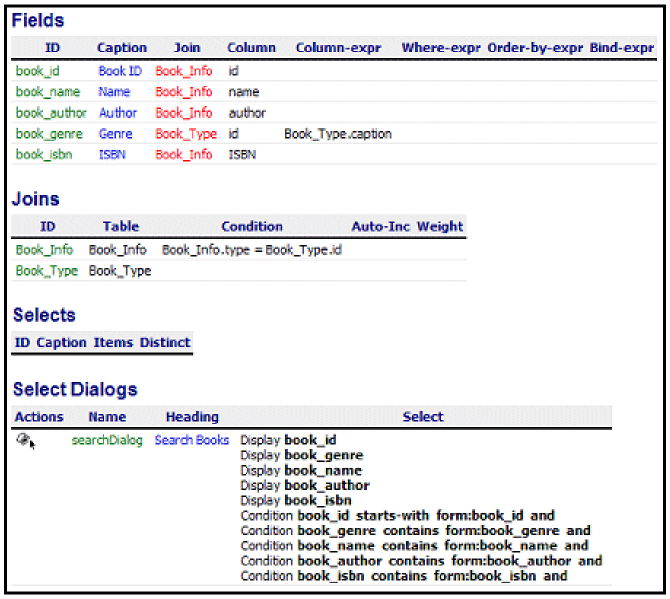
Name	Value
Allow reload	Yes

Source Files

File	Included-from
C:\web-application\web-library\Site\WEB-INF\sql\statements.xml	

Figure 61: Unit Testing In ACE

Now in a web browser open up the Library's ACE[22] and, after logging in, choose the *SQL Query Definitions* item from the *Database* menu. You should see a list of all your query definitions shown here. In the case of the Library, there should only be the one query definition named `searchBooks`.



Fields

ID	Caption	Join	Column	Column-expr	Where-expr	Order-by-expr	Bind-expr
book_id	Book ID	Book_Info	id				
book_name	Name	Book_Info	name				
book_author	Author	Book_Info	author				
book_genre	Genre	Book_Type	id	Book_Type.caption			
book_isbn	ISBN	Book_Info	ISBN				

Joins

ID	Table	Condition	Auto-Inc	Weight
Book_Info	Book_Info	Book_Info.type = Book_Type.id		
Book_Type	Book_Type			

Selects

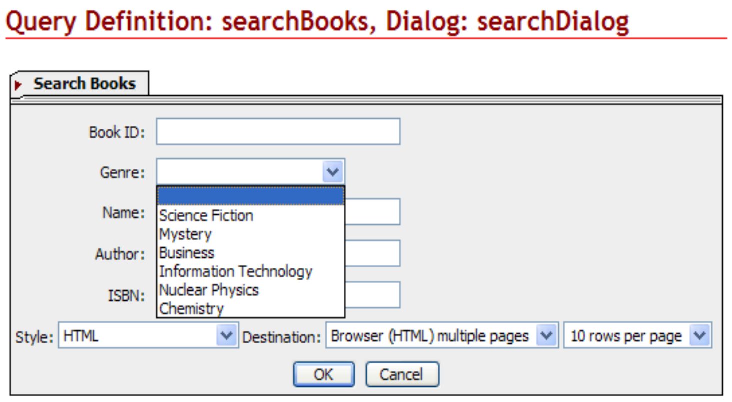
ID	Caption	Items	Distinct

Select Dialogs

Actions	Name	Heading	Select
	searchDialog	Search Books	Display **book_id** Display **book_genre** Display **book_name** Display **book_author** Display **book_isbn** Condition **book_id starts-with form:book_id and** Condition **book_genre contains form:book_genre and** Condition **book_name contains form:book_name and** Condition **book_author contains form:book_author and** Condition **book_isbn contains form:book_isbn and**

Figure 62: Query Definitions

- Click on the query definition's name to bring up a detailed breakdown of its components including fields, joins, selects and select-dialogs. In the case of the Library, there should be only one select-dialog listed.
- In the Select Dialogs section, click on the icon under the Actions column. This should pop up in a new browser window the select dialog you created earlier.



Query Definition: searchBooks, Dialog: searchDialog

Search Books

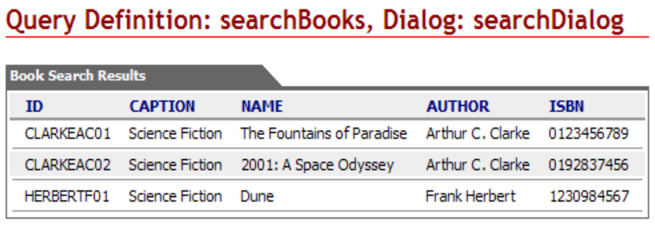
Book ID:	
Genre:	
Name:	
Author:	
ISBN:	

Style: HTML Destination: Browser (HTML) multiple pages 10 rows per page

Figure 63: Select Dialog You Created

- Notice how all the fields are of type field.text except the genre field, which is a field.select.

[22] If you are evaluating Sparx online, you can access the Sparx Collection ACE at http://developer.netspective.com/samples/library/ace



Query Definition: searchBooks, Dialog: searchDialog

Book Search Results

ID	CAPTION	NAME	AUTHOR	ISBN
CLARKEAC01	Science Fiction	The Fountains of Paradise	Arthur C. Clarke	0123456789
CLARKEAC02	Science Fiction	2001: A Space Odyssey	Arthur C. Clarke	0192837456
HERBERTF01	Science Fiction	Dune	Frank Herbert	1230984567

Figure 64: Results of field_select.

- At this point, if you still have data left in the database from when you tested the Add/Edit and Delete dialog modes in ACE, you can search for a Book ID you entered or a book name you entered to see the results this select-dialog gives back. If you do not have any data in the database, move on to the next section where you can finally glue all these different components together to form a final useable application.

Integrating the Data and Presentation Layers

Now that the major pieces of the Sparx Library are complete and tested, it is time to integrate them all into the final application. Just as a record of what you have accomplished so far, the different pieces of functionality that are complete are listed below along with brief implementation notes.

- A database schema to store data about books.
- A user interface to allow addition of books to the database, editing of existing books in the database and deletion of books from the database.
- SQL statements to allow listing all the books in the database or to list details about one book.
- Query definition to allow searching for books in the database by any criteria of the users' choosing.
- Data Access Layer (DAL) for inserting, updating, and removing data to and from the database.

To integrate all these components into an application you need two major items. The first is a common look and feel for the application and the second is a collection of JSP files that embed each of these components and can be called at will.

The fastest way to achieve a common look and feel is to use a custom JSP tag to create a template that will encase all custom content you add to a JSP page, such as embedded Sparx components. This template should allow the user to access all parts of the application from it. Once this is achieved, all pages that are a part of the Sparx Library will not only look and feel the same but will also be able to take you to any other part of the application with minimal work. However, since all the Library's functionality is created using Sparx components, they need to be embedded in known JSP pages so you can create links to them in the JSP custom tag template.

All the source code presented in this chapter so far has hinted at specific names for JSP files that are planned on being used in the final application. The JSP page layout you should use for this tutorial is shown in the list below.

- ♦ `index.jsp`: Being the first page the user will see, this should give him as much information as possible while giving him access to as many application features as possible. Other than the template, then, the only thing needed here is a list of all the books. Achieve this by embedding the `library.sel_all_books` SQL statement in this page. This page should be liked as the "Home" in the custom JSP tag that you create for the Sparx Library template.
- ♦ `bookInfo.jsp`: This page is where the `library.bookInfo` dialog is embedded. By passing dialog commands to the JSP you can make this one page serve to add, edit and delete a book. This page should be linked from the custom JSP template tag (for adding books) as well as being used to edit and delete records in the SQL statement reports.
- ♦ `search.jsp`: This page is where the `searchBooks` query definition is embedded. This page should be linked from the custom JSP template tag to allow the user to quickly search for books in the database.
- ♦ `viewBook.jsp`: This page is where the `library.sel_one_book` SQL statement is embedded. Here you can click on a book title in a book listing and get detailed information listed in a different format.

Creating a Custom JSP Template Tag

Having determined what needs to be done, you can start by creating a custom JSP tag that will be used in all four JSP pages that the Sparx Library needs. The basic process of creating a custom JSP tag is simple in concept: you create a Java class with two main methods, one of which determines what happens when the opening custom tag is encountered and the other determines what happens when the closing custom tag is encountered. In the present case, you want to output the beginning of the template in the opening tag and the end of the template in the closing tag. Once done, all pages that use the custom tag will automatically ensure their content fits in the area provided for them by the template.

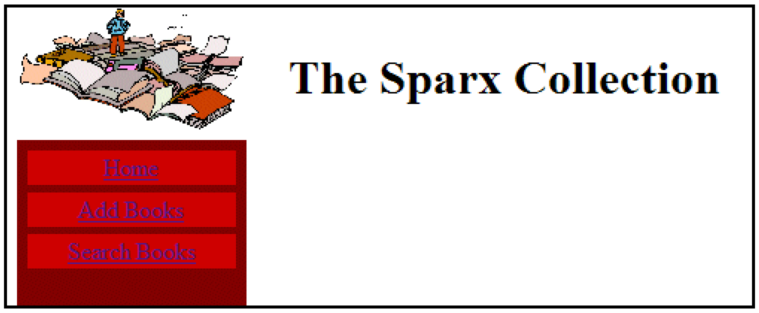


Figure 65: The End Result Of The Template With A Blank Page

The source for the custom JSP tag is shown here and the important parts will be highlighted after the source.

```java
package app.tag;

import java.io.*;
import java.util.*;
import java.math.BigDecimal;
import java.net.URLEncoder;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

import com.netspective.sparx.util.config.*;
import com.netspective.sparx.xaf.form.*;
import com.netspective.sparx.xaf.html.*;
import com.netspective.sparx.xaf.html.component.*;
import com.netspective.sparx.xaf.navigate.*;
import com.netspective.sparx.xaf.page.*;
import com.netspective.sparx.xaf.security.*;
import com.netspective.sparx.xaf.skin.*;
import com.netspective.sparx.util.value.*;

import app.security.AppLoginDialog;

public class PageTag extends
 com.netspective.sparx.xaf.taglib.PageTag
{
    static private VirtualPath menuStructure;

    public int doStartTag() throws JspException
    {
        doPageBegin();

        JspWriter out = pageContext.getOut();

        HttpServletRequest req = (HttpServletRequest)
 pageContext.getRequest();
        HttpServletResponse resp = (HttpServletResponse)
 pageContext.getResponse();
        ServletContext servletContext =
 pageContext.getServletContext();

        HttpSession session = req.getSession();

        try
        {
            if(! hasPermission())
            {

 out.print(req.getAttribute(PAGE_SECURITY_MESSAGE_ATTRNAME));
                return SKIP_BODY;
            }
            String rootPath = req.getContextPath();
            String resourcesUrl = rootPath + "/resources";

                                           e a special border
 around them. If this is not a sample app, then comment out
              doSamplePageBegin() and uncomment the
 <!DOCTYPE>... through <body> */

            doSamplePageBegin(resourcesUrl
 +"/css/library.css");
            /*
            out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD
 HTML 4.01 Transitional//EN\">");
            out.println("<html>");
                           ead>");
            out.println("  <title>" + getTitle() + "</title>");
                            link rel='stylesheet' href='"+
 resourcesUrl +"/css/library.css'>");

            out.println("  <meta http-equiv=\"content-type\"
 content=\"text/html; charset=ISO-8859-1\">");
            out.println("</head>");
            out.println("  <body>");
            out.println("");
            */
            out.println("<table cellpadding=\"2\"
 cellspacing=\"5\" border=\"0\" >");

            out.println("    <tr>");
            out.println("      <td valign=\"top\"
 width=\"100\"><img src=\"" + resourcesUrl +
"/images/scatteredBooks.jpg\" alt=\"Library logo\"
width=\"150\" height=\"84\">");
            out.println("       <br>");
            out.println("       </td>");

align=\"center\">");
                                 1>" + getHeading() + "</h1>");
            out.println("       </td>");
            out.println("     </tr>");
                       ("     <tr>");
            out.println("       <td valign=\"top\"
bgcolor=\"#333333\">");
                                   le width=\"100%\"
border=\"0\" cellspacing=\"5\" cellpadding=\"2\">");
            out.println("         <tbody>");
            out.println("           <tr>");
            out.println("             <td valign=\"middle\"
bgcolor=\"#EEEEEE\" align=\"center\"><font color=\"#ffffff\"><a
href=\"" + rootPath + "/index.jsp\">Home</a></font><br>");
            out.println("             </td>");

            out.println("           <tr>");
            out.println("             <td valign=\"middle\"
bgcolor=\"#EEEEEE\" align=\"center\"><font color=\"#ffffff\"><a
href=\"" + rootPath + "/bookInfo.jsp?data_cmd=add\">Add
Books</a></font><br>");
                                      </td>");
            out.println("           </tr>");
            out.println("           <tr>");
            out.println("             <td valign=\"middle\"
bgcolor=\"#EEEEEE\" align=\"center\"><font color=\"#ffffff\"><a
href=\"" + rootPath + "/search.jsp\">Search
Books</a></font><br>");
            out.println("             </td>");
            out.println("           </tr>");
            out.println("        </tbody>");
                      n("      </table>");
            out.println("       <br>");
            out.println("       </td>");
            out.println("       <td align=\"center\"
valign=\"top\" bgcolor=\"white\"><br>");


        catch(Exception e)

            StringWriter stack = new StringWriter();
            e.printStackTrace(new PrintWriter(stack));
            throw new JspException(e.toString() +
stack.toString());


        if(handleDefaultBodyItem())
            return SKIP_BODY;
        else



    public int doEndTag() throws JspException
    {
        JspWriter out = pageContext.getOut();
        String rootPath = ((HttpServletRequest)
pageContext.getRequest()).getContextPath();


            out.println("   </td>");
            out.println("     </tr>");
            out.println("  </tbody>");
            out.println("</table>");
            out.println(" <p>&nbsp;");
            out.println(" <p>");
            //out.println("<table width=100%><tr><td
align=right><a target='netspective'
```

```
href='http://www.netspective.com'><img border='0' alt='Powered
by Netspective Sparx' src='"+ rootPath
+"/sparx/resources/images/powered-by-
sparx.gif'></a></td><td><font size=1>"+
com.netspective.sparx.BuildConfiguration.getVersionAndBuildShor
t() +"</font></td></table></body>");
            //out.println("</body>");
                     ntln("</html>");
            doSamplePageEnd(); // remove if this is not a
Netspective "Sample" application
        }
        catch(IOException e)
        {
            throw new JspException(e.toString());
        }

        doPageEnd();
        return EVAL_PAGE;
    }
}
```

The important thing to notice in the source shown above is that this Java class is derived from a Sparx `PageTag` class. This is important because the Sparx PageTag class has the ability to add user authentication (i.e. login based access to the Library) with just a few more lines of Java. Using this code, therefore, is a good way to ensure easier extensibility.

This code belongs to the `app.tag` package and should therefore be saved in a file called `PageTag.java` under the `WEB-INF\classes\app\tag` directory so it is be available to all pages that require it.

Creating the JSP Files

Once you have created the custom JSP template tag, it is time to create the content for the JSP files that need to glue the application into a useable whole.

index.jsp

```
<%@ taglib prefix="sparx" uri="/WEB-INF/tld/sparx.tld"%>
<%@ taglib prefix="app" uri="/WEB-INF/tld/page.tld"%>

<app:page title="The Sparx Collection" heading="The Sparx
      Collection">
      <sparx:query name="library.sel_all_books"/>
</app:page>
```

bookInfo.jsp

```
<%@ taglib prefix="sparx" uri="/WEB-INF/tld/sparx.tld"%>
<%@ taglib prefix="app" uri="/WEB-INF/tld/page.tld"%>

<app:page title="The Sparx Collection" heading="Book Information">
      <sparx:dialog name="library.bookInfo"/>
</app:page>
```

search.jsp

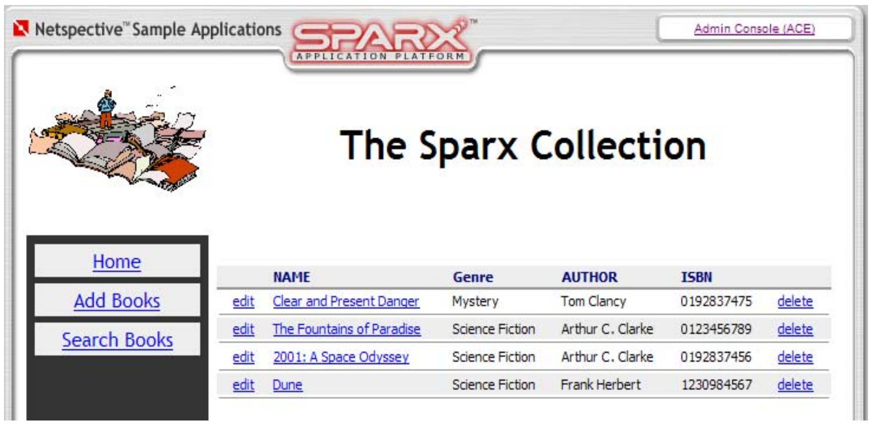
```
<%@ taglib prefix="sparx" uri="/WEB-INF/tld/sparx.tld"%>
<%@ taglib prefix="app" uri="/WEB-INF/tld/page.tld"%>

<app:page title="The Sparx Collection" heading="Search Books">
      <sparx:query-select-dialog source="searchBooks"
      name="searchDialog"/>
```

```
</app:page>
```

viewBook.jsp

```
<%@ taglib prefix="sparx" uri="/WEB-INF/tld/sparx.tld"%>
<%@ taglib prefix="app" uri="/WEB-INF/tld/page.tld"%>

<app:page title="The Sparx Collection" heading="Book Information">
      <sparx:query name="library.sel_one_book" skin="detail"/>
</app:page>
```

Conclusion

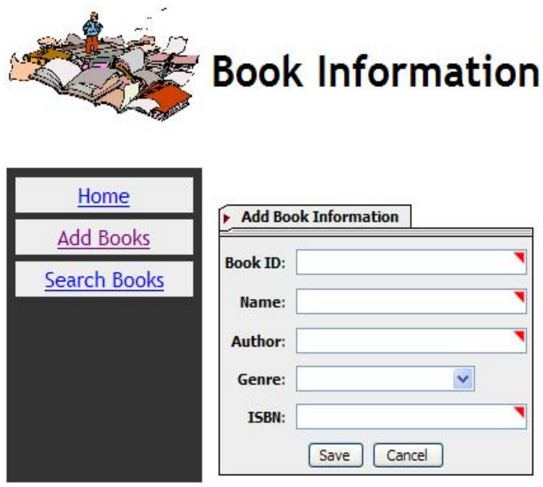


	NAME	Genre	AUTHOR	ISBN	
edit	Clear and Present Danger	Mystery	Tom Clancy	0192837475	delete
edit	The Fountains of Paradise	Science Fiction	Arthur C. Clarke	0123456789	delete
edit	2001: A Space Odyssey	Science Fiction	Arthur C. Clarke	0192837456	delete
edit	Dune	Science Fiction	Frank Herbert	1230984567	delete

Figure 66: The Completed Sparx Library

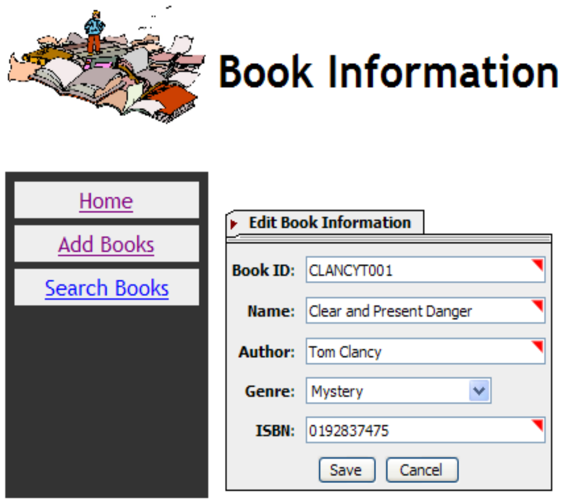
Congratulations! The Sparx Library is complete. Now you should open up your browser window and go to the Library's main page[23]. You should see either a blank list of books or a list of all the books that are already a part of the database. If there are no books in the database, try adding a few, editing some more and deleting a few. Try searching for books by genre or name or any other field for that matter.

[23] If you are evaluating Sparx online, you can access the Sparx Collection at http://developer.netspective.com/samples/library



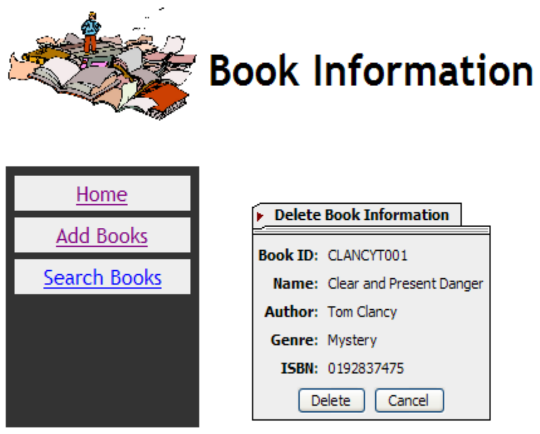
Book ID:	
Name:	
Author:	
Genre:	
ISBN:	

Figure 67: Adding a Book to the Library



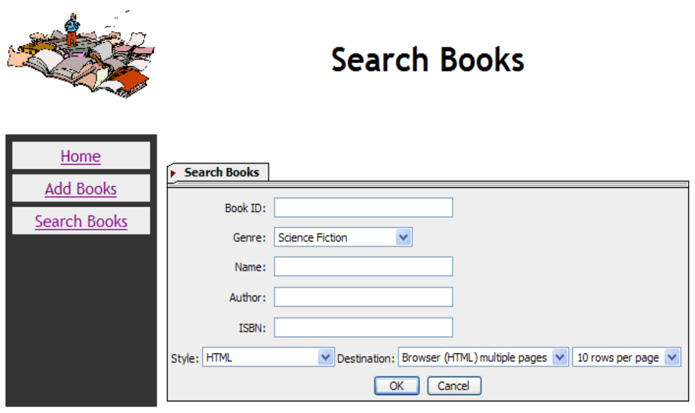
Book ID:	CLANCYT001
Name:	Clear and Present Danger
Author:	Tom Clancy
Genre:	Mystery
ISBN:	0192837475

Figure 68: Editing a Book in the Library



Book ID:	CLANCYT001
Name:	Clear and Present Danger
Author:	Tom Clancy
Genre:	Mystery
ISBN:	0192837475

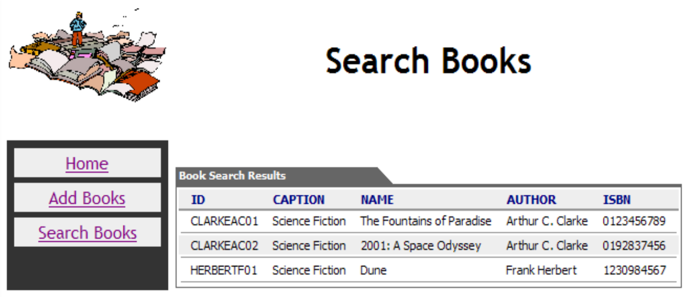
Figure 69: Deleting a Book from the Library



Book ID:	
Genre:	Science Fiction
Name:	
Author:	
ISBN:	

Style: HTML Destination: Browser (HTML) multiple pages 10 rows per page

Figure 70: Searching For Books In The Library



Book Search Results

ID	CAPTION	NAME	AUTHOR	ISBN
CLARKEAC01	Science Fiction	The Fountains of Paradise	Arthur C. Clarke	0123456789
CLARKEAC02	Science Fiction	2001: A Space Odyssey	Arthur C. Clarke	0192837456
HERBERTF01	Science Fiction	Dune	Frank Herbert	1230984567

Figure 71: Search Books Results

Your first Sparx application that does something meaningful is complete and you have gone through the shallowest parts of almost everything Sparx has to offer. You can now continue to improve the Library with newer ideas, delving deeper into Sparx or you can start a new application and use what you learned here to see how fast you can have a running system.

Conclusion

Although a great deal of information has been presented in this document, it's only a small portion of the information available online and in other documents. This guide has covered the following topics.

- How to evaluate Sparx online and on your own system.
- What the major modules of Sparx are and how to use them.
- The Sparx directory structure and the purpose of each entry.
- How to create a Hello World sample application.
- How to use the Sparx build scripts and ACE to help develop your application.
- How Sparx manages the user interface (forms and dialogs) interaction.
- How to create a small but complete Library sample application.

What's Next

Sparx Reference Guide

This user's guide has explained the basic usage of Sparx, but does not contain a reference guide to all of the XML tags and usage scenarios possible. For more information, the best place to start is http://developer.netspective.com.

Cura Sample Application and Tutorial

The two sample applications included in this user's guide (Hello World and Library) only present the simplest Sparx usage scenarios. Another application, called Cura, is a complete web-base project management application and demonstrates the advanced capabilities of Sparx. Please review that sample application and the *Sparx Application Platform Developers Guide Volume II* for additional tutorials and information.