

Volume

1



---

Developer Documentation Series

# Sparx Tutorial

A SPARX TUTORIAL

# How to Make Sparx Fly

---

? Netspective Inc.,  
Palmer Business Park • 4550 Forbes Blvd, Suite 320  
Lanham, MD 20706  
Phone 301.879.3321

---

# Table of Contents






<b>AN OVERVIEW OF SPARX</b>	<b>1</b>	<b>Success</b>	<b>31</b>
Why Sparx?	1	<b>ADVANCED DIALOGS</b>	<b>32</b>
Sparx Benefits	1	The Test Dialogs	32
<b>EXTENSIBLE APPLICATION FRAMEWORK</b>	<b>2</b>	Adding Test Dialogs to Your Application	32
XAF Features	3	The Test Dialogs in ACE	33
<b>APPLICATION COMPONENT EXPLORER</b>	<b>5</b>	<i>Test Dialog 01 A</i>	33
ACE Features	5	<i>Test Dialog 02</i>	35
<b>EXTENSIBLE INFORMATION FRAMEWORK</b>	<b>6</b>	<i>Test Dialog 03</i>	38
XIF Benefits	6	<i>Test Dialog 05</i>	40
<b>SPARX DATA ACCESS LAYER (DAL)</b>	<b>7</b>	<i>Test Dialog 06</i>	41
DAL Benefits	7	Conclusion	42
DAL Features	7	<b>THE SPARX COLLECTION</b>	<b>43</b>
<b>SPARX: STARTING THE FIRE</b>	<b>8</b>	Functionality	43
Application Root	9	Application Design	43
Database	9	Database Design	44
Site	9	Implementation	44
WEB-INF	10	Starting the Application	45
UI	10	Setting up the Database	45
SQL	10	The Database Schema	46
Classes	11	<i>Step by Step Explanation</i>	47
Lib	11	<i>Testing the Schema</i>	49
Security	12	<i>Generating the Data Description Language</i>	51
Log	12	Creating the User Interface	52
TLD	12	<i>The Java Dialog</i>	53
<b>APPLICATION CONFIGURATION</b>	<b>13</b>	<i>Step by Step Explanation</i>	57
web.xml	13	<i>Dissecting populateValues</i>	58
configuration.xml	13	<i>Dissecting execute</i>	60
<b>HELLO WORLD</b>	<b>14</b>	<i>Dissecting processEditAction</i>	61
Functionality	14	<i>Unit Testing with ACE</i>	63
Design	14	Creating a Static SQL Library	66
Implementation	15	<i>Introduction to Static SQL Libraries</i>	66
<b>SETTING UP THE APPLICATION</b>	<b>15</b>	<i>Step by Step Explanation</i>	68
<b>CREATING THE USER INTERFACE</b>	<b>17</b>	<i>Explaining Reports</i>	69
Creating the XML Dialog	17	<i>Unit Testing in ACE</i>	71
<i>Step by Step Explanation</i>	18	Creating Query Definitions	73
Unit Testing with ACE	19	<i>Step by Step Explanation</i>	74
Embedding Dialogs into JSP Pages	24	<i>Unit Testing in ACE</i>	78
<i>Step by Step Explanation</i>	24	The Collection Integrates	80
<i>Testing Embedded Dialogs</i>	25	<i>Creating a Custom JSP Template Tag</i>	81
Java Dialogs	26	<i>index.jsp</i>	85
<i>Step by Step Explanation</i>	27	<i>bookInfo.jsp</i>	85
Binding Java to XML	28	<i>search.jsp</i>	85
<i>Compiling the Application</i>	28	<i>viewBook.jsp</i>	85
<i>Testing the Finished Dialog</i>	29	Conclusion	86

---

## An Overview of Sparx

**N**etspective's open-source Sparx framework allows enterprises to build and deploy more e-business Java applications using fewer programmers, in less time, with higher-quality, and better documentation than conventional application servers and Servlet engines alone. Using Sparx, you can stop building applications from scratch each time and leverage your existing talent and components. As a pure Java library, Sparx can be integrated into existing systems at any development stage. Sparx will work with any Java2 JDK and Servlet engine and can work equally as well as a JSP library (with advanced, pre-defined custom tags) or a Servlet library (providing MVC-based page control). Some of our customers have been able to cut their development budgets in half (in some cases by up to 75%) by reducing the number or qualifications of their programmers. This is because Sparx allows junior and mid-level Java developers to be as or more effective than senior and more experienced developers that do not use Sparx.




### ICON KEY

	Directory names
	File names
	Tips
	Screenshots
	Links to API Reference

### Why Sparx?

Sparx is a complete framework which helps with the entire software development lifecycle. Design and prototyping, implementation, automatically generating unit tests, automatically creating implementation documentation, and providing production logs and metrics are just some of the development deliverables and phases that Sparx helps accelerate and standardize.

### Sparx Benefits

-  Application developers spend time on real features significant to end-users instead of infrastructure issues that are important only to programmers.
-  Technical managers can better manage their application development projects by utilizing the built-in project management, application documentation, unit-testing, and artifact-generation tools.
-  Most of the user interface and database logic is coded in a declarative style using XML instead of programmatic style using Java. This significantly reduces the amount of code (as much as 50-75% of code can be eliminated), increases re-use, and increases code quality.

- ✍ Analysts can use the declarative user interface features to create prototypes that can later be completed by programmers (no more throw-away prototypes).
- ✍ Applications are built by assembling declared UI (forms/dialogs) and database (SQL) components combined with application-specific business logic using single or multiple distributed application tiers.
- ✍ Sparx is not a templating system that simply generates HTML but a feature-rich framework that significantly reduces the time to produce high-quality data-intensive thin-client applications.
- ✍ Sparx does not favor Servlets over JSPs or JSPs over servlets and can work in one, the other, or both environments simultaneously with no loss of functionality in either environment.
- ✍ Although Sparx favors XML for specifications of forms and database components, programmers can choose to eliminate XML and use the Sparx APIs and program completely in Java.
- ✍ Implements common design patterns like MVC and factories. Skins infrastructure allow identical business logic to be used across different user interfaces for a variety of browsers and platforms like handhelds.
- ✍ Netspective understands the rigors of methodologies like waterfall and the agility of methodologies like eXtreme Programming (XP) and created Sparx to work equally well in various environments.

In addition, one of the most important benefits that Sparx brings to your development environment is *lack* of a change in the existing code-base. Sparx is a pure library, unlike other similar frameworks which are containers for your application. It consists of one JAR containing the Sparx binary and a set of HTML and XML resources like XSLT style sheets, icons and an extensive JavaScript library. This simple structure affords developers a lot of flexibility in how they want to use Sparx. Therefore, you do not have to redesign or recode your application to comply with the limitations of a framework container. Instead, you can start out by using a few Sparx features here and there and adopting more of the Sparx ease and speed of development as the need arises.

## eXtensible Application Framework

The eXtensible Application Framework (XAF) is Sparx's Java library consisting of over 250 re-usable classes that greatly simplify the development and deployment of small-, medium-, and large-scale thin client, browser-based, data-driven, dynamic web applications. Based on Enterprise Java standards like J2EE, XML, Servlets, JSPs, JDBC, and JNDI, XAF uses proven re-use development methodologies that can be applied to produce robust Java web applications in half the time. Using Sparx, small development teams can produce and deploy significant applications with higher quality

code and automated implementation documentation. Sparx promises that a small number of average programmers can create great software that will delight end-users.

## XAF Features

XML Resources	XAF specifications are performed using XML -- all dialogs, fields, validation rules, some conditional processing, all SQL statements, dynamic queries, configuration files, and many other resources are stored in XML files that are re-usable across applications. Although XML is the preferred method for creating resource files, almost anything that can be specified in XML can also be specified using the XAF Java APIs.
Executable Specifications	The majority of the XAF features including dialogs (UI), SQL Statements, dynamic query rules and schema definitions, etc are done using what are called Executable Specifications. These executable specifications mean that most of the applications' resources double as both specifications (which can be extracted and automatically documented) and executable code. The same resource acts as the documentation and the testable application.
Value Sources	Value sources provide dynamic access to common business logic and may be considered a business rules library. Many of the classes in the XAF use value sources, which are simply Java classes that follow a specific interface, to provide values for captions, defaults, comparisons, conditionals, and many other types of variables. Value sources allow common business logic to be stored in reusable classes and then used either in XML or Java files where necessary. Value sources can be either single or multiple (list context) and are used in dialogs, fields, SQL statements, and many other places where dynamic data is required. The format of a value source is similar to a URL (name:params).
Centralized Configuration	The XAF favors XML storage of properties instead of using Java properties files. The <code>ConfigurationManager</code> class allows multiple properties to be defined in a single XML file, complete with variable replacements and the ability to create single-property or multiple property (list) items. Optionally, any property name could refer to value sources as part of the definition of a property so that the value of a property can become dynamic and be computed each time the property is used (in case the value of the property is based on a servlet request or session variable or some other application-defined business rule).
Advanced Forms/Dialogs	The XAF refers to HTML forms as "Dialogs" because it handles the two-way interaction between browsers and users completely; this includes data persistence, data validation, a sophisticated client-side JavaScript library and user interface skins. Dialogs are can be defined completely in XML, completely in Java, or a combination of the two. Even in XML, the entire Dialog including labels, captions, validation logic, conditional displays,

	and other advanced UI features can be easily defined. By keeping the entire definition in XML, non-programmers or junior engineers can create forms and more experienced developers can attach business logic as appropriate.
Datasources and Database Connectivity	The XAF 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.
SQL DML Generation	JSP custom tags and java classes are provided to automatically create SQL insert/update/remove DML (Data Manipulation Language) commands. By providing simple name/value pairs of data, XAF can automatically generate complex DML statements.
SQL Statement Libraries	To encourage reusability and encapsulation and reduce the amount of time spent creating "data beans", XAF allows all SQL statements and dynamic parameters used in a project to be specified in one or more SQL files. Once defined, a single or multiple SQL statements may be used in reports, dialogs (forms), Servlets, or JSP-pages. In many cases, SQL statement pooling completely replaces simple data-serving beans since data objects are automatically created for all SQL statements. Data can be easily aggregated from multiple data sources because each SQL statement in the statement pool can be specified (either in XML or JSP) to come from a variety of pre-defined or dynamic data sources.
Dynamic Queries	A powerful XML-based tool called Query Definitions allows developers to define tables, columns, joins, sort orders, and other important data through the use of Meta Information about data relationships. Once a developer creates a query definition, XAF allows end-users to use simple HTML-based forms to automatically generate SQL accurate and performance -tuned statements to create paged reports or export data to external sources.
Reports	XAF reports are defined completely in XML. This includes headings, banners, column types, calculations, grouping, sort order, etc. By keeping the entire definition in XML, non-programmers or junior engineers can create report definitions and more experienced developers can attach business logic.
Skins and Device Independence	XAF separates form/report presentation from form/report design and logic by automatically creating all HTML and DHTML in user-defined "skin" objects. The skins perform all drawing operations while the report/form objects manage all of the fields and validation. One immediate benefit of skins is the ability to design and describe a dialog once and execute it on mobile, small form- factors (handhelds), notebooks, and desktops or to

	different formats like PDF, comma-separated, and tab-delimited.
Security and Personalization	XML-based centralized Access Control Lists (ACL's) allow developers to restrict access to forms, reports, pages, and other resources based on user names, types, location, roles, capabilities, or other permissions. With security and other features, all XAF applications support personalization features that allow applications to respond differently to different users based on location, user type, or user names.

## Application Component Explorer

The Application Components Explorer (ACE) is a servlet that provides a browser-based administrative interface to the myriad of Sparx dynamic components and objects. ACE is automatically available to all Sparx-based applications during development and can be optionally turned off in production (for security).

### ACE Features

Automatic Implementation Documentation	Instead of having to create functional specifications and other implementation documentation manually, ACE automatically documents (using the XML definitions and XSLT style sheets) all the forms (web dialogs), SQL statements, schema objects, and other programming artifacts in a centralized browser-based interface. Developers concentrate on application creation while Sparx automatically documents their work. Managers can use this documentation in a real-time basis to track programmer work and productivity.
Application Unit Testing	While developers are working on forms and SQL statements, ACE automatically provides browser-based testing of the forms and statements. No servlets or JSPs need to be written for basic testing of forms, validations, and SQL statements. Once initial testing is completed and requirements are solidified, then the forms and statements can be aggregated to create interactive applications. End users can use the interactive testing tools to see code as it is being developed (supporting eXtreme Programming concepts).
Project Metrics	As developers create forms, SQL statements, query definitions, JSP, servlets, and other code, ACE automatically maintains basic application metrics. Metrics are an important part of every sophisticated software development process and Sparx can not only capture the metrics but store them in XML files so that they can be analyzed over time.
Application Performance Tracking and Logging	All mission-critical and sophisticated web applications need to be tuned for both database and application performance. XAF provides log output that ACE tracks for maintaining data about execution statistics for SQL statements, servlet and JSP pages, dialogs/forms, and security.



**Centralized Project Documentation**

ACE provides a centralized location for all project documentation for any application. Instead of storing application code and programmer documentation separately, ACE brings tag documentation, Javadocs, MS Office documents, and other project documents into a single easily accessible place. Managers will no longer need to hunt for documents.

## eXtensible Information Framework

The eXtensible Information Framework (XIF) is Sparx's Java and XML library consisting of dozens of reusable tables, columns, and indexes that are useful for almost any e-business application.

### XIF Benefits

- ✍ Database Programmers spend time on essential tables and schema elements significant to a specific application instead of rewriting common schema elements for each application.
- ✍ Allows for re-use of Schemas across applications and produces and maintains Schema documentation. XIF encourages the creation and re-use of a set of datatypes and tabletypes that define standard behaviors for columns and tables. Datatypes and tabletypes comprise the SchemaDoc database dictionary and can easily be inherited and extended.
- ✍ Almost all schema objects like tables, columns, data types, etc. are managed in a database-independent XML SchemaDoc. The entire schema is managed in XML as an XML document (a SchemaDoc) and SQL is generated through XSLT style sheets (the templates). The same SchemaDoc can be used to generate database-specific SQL DDL allowing a single XML source schema to work in a variety of SQL relational databases (like Oracle, SQL Server, MySQL, etc.).
- ✍ Database-specific SQL DDL is created by applying XSLT style sheets to a SchemaDoc. Experienced DBAs are not required to create consistent, high-quality SQL DDL during the design and construction phases of an application. Database-dependent objects like triggers and stored procedures are not managed by the XIF and are created using existing means.
- ✍ Database-independent Java Object-relational classes are created by applying XSLT style sheets to a SchemaDoc. This is called the Application DAL (Data Access Layer). XIF can automatically generate a Java Object-relational DAL (Data Access Layer) for an entire schema, automating the majority of SQL calls by providing strongly-typed Java wrappers for all tables and columns.

## Sparx Data Access Layer (DAL)

Using XSLT stylesheets, the XIF can generate a complete Java O-R map to every table in the schema. This Java O-R map is called the Sparx Data Access Layer, or DAL. The DAL requires a valid XML-based SchemaDoc be defined using the XIF tool. Once you have a valid XML SchemaDoc, you can generate the DAL using either a command-line based tool (recommended) or through ACE.

### DAL Benefits

- ✍ The DAL allows strongly-typed Java classes to be generated for each datatype, tabletype, and table in the schema.
- ✍ The entire schema becomes fully documented through the generation of JavaDoc documentation (the DAL generators generate JavaDoc comments automatically for all classes, members, and methods).
- ✍ Each datatype becomes a `Column` Java Class (which an actual table's column becomes an instance of).
- ✍ Each tabletype becomes a `Table Type` Java interface that each appropriate Table class implements.
- ✍ Each table generates specific classes. Assuming a table called Person,
  - ✍ A `Person` Interface is created
  - ✍ A `PersonRow` class that implements the `PersonDomain` is created
  - ✍ A `PersonRows` class that can hold a list of `PersonRow` objects is created
  - ✍ A `PersonTable` class that contains the column definitions (names, validation rules, foreign key constraints, etc) and SQL generation methods is created.

### DAL Features

DataAccessLayer Class	This is the primary class that programmers will use to access their schema through Java. It contains all of the table definitions for the Schema.
Column Classes	Each datatype becomes a Java class that extends the <code>com.xaf.db.schema.AbstractColumn</code> class and implements the <code>com.xaf.db.schema.Column</code> interface. For example, the "text" datatype becomes a <code>schema.column.TextColumn</code> class; the "integer" datatype becomes the <code>schema.column.IntegerColumn</code> class; etc.
Domain, Row and Rows Classes	Each table generates three different Java files which represent the data stored in the table.
Table Classes	Each table generates its own <code>Table</code> class. For example, assuming a Person table in the database, a <code>table.PersonTable</code> class is generated.

## Sparx: Starting The Fire

*Getting acquainted with the recommended directory structure of a Sparx project.*

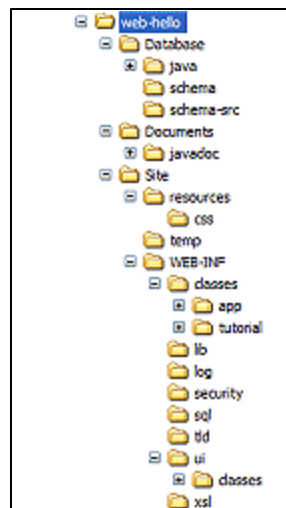
Every application that is created using the Sparx framework needs to conform to the standard servlet application directory structure<sup>1</sup>. This way, all the different components of the application are stored in predictable locations. Additionally, all application servers conformant to the Java servlet standard will be able to find and run the application components without a lot of reconfiguration. We will describe the different essential directories that need to be present in a Sparx application.

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<sup>2</sup> 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. Using the bundled `build` script, creating a new application is merely a matter of calling `build` with the appropriate parameters. This will create a skeleton directory structure suitable for starting a new application in. A developer wishing to experiment with Sparx can then rename and modify this directory structure with his own application data. We will go over this process in detail in the next chapter.

The screenshot shows a tree view for a sample Sparx application. The root directory for the application is highlighted. We shall explain the purpose of each directory in the screenshot in the following sections.

**Figure 1:**  
Tree View For A  
Sample Sparx  
Application.



<sup>1</sup> 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>

<sup>2</sup> You can learn more about Apache Ant at its official web site: <http://jakarta.apache.org/ant/index.html>

## ✍ Application Root

This directory contains every file that the application will need to run. All configuration files for the application server to read specific to this application will be located under this directory. All files that need to be processed to display on a web browser will be found under this directory. It needs to be noted that the directory structure of the application under this directory is a recommended best practices structure, not a requirement. There are just a few directories and files that are absolutely required by the Java Servlet specification and those will be mentioned as being required. All the rest are for the use of Sparx only.

## ✍ Database

This directory, located directly under the application root directory, contains an XML representation of the database schema used by the application. It contains two to three subdirectories, each containing variously processed forms of the same database schema.

The `schema-src` 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 complete database schema.

The `schema` directory contains the SQL generated after Sparx processes the `schema.xml` file. 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 `java` directory, which is created automatically after an application is built, contains the complete Object-Relation map for the database table structure given in the `schema.xml` file. This O-R map is

The `java` directory, which is created automatically after an application is built, contains Java equivalents of the tables defined in the `schema.xml` file. These Java equivalents treat each data type and each table as their own class. All these Java classes are part of the Sparx Data Access Layer (DAL) explained briefly earlier. Whereas the first type of processed schema is meant for use in the database server, this second type of processed schema is meant for use by an application developer in his programs to speed up development.

## ✍ Site

This directory, also located directly under the application root directory, contains all the browser accessible files for the application. 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 directories other than `WEB-INF` should be directly accessible through a browser. All files in this directory are also directly accessible through a web 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>.

### WEB-INF

This directory, located under the `Site` directory, is the only one required by the Java servlet specification. The specification requires a directory named `WEB-INF` to be located anywhere in the directory structure of the application. The preferred location of the `WEB-INF` directory in a Sparx application is directly under the `Site` directory. It contains all files private to the application. You will find configuration files for the application server under which the application is running, configuration files for Sparx and any other similar private data.

Sparx uses this directory to store quite a few different things. Each separate category of items has its own subdirectory. Each of these categories and corresponding sub-directories is listed here.

### UI

This directory contains one file named `dialogs.xml`. It may also contain other files, all of which are 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.

The first is a Java version of the dialog. Before we give an overview of how the two are related, you should know that all Sparx dialogs are 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.

The second sub-component is a Java data bean 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 binary form inside the `classes` directory under `UI`.

### SQL

This directory contains one file named `statements.xml`. It may also contain other files, all of which (like in the case of `dialogs.xml`) are 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.

### Classes

This directory 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 for each dialog are stored in the `classes` directory under `UI`. This is done mainly to avoid cluttering up the `classes` directory with autogenerated source and binary Java files.

These Java 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 in this directory. Any auxiliary Java classes that you might need should also be placed here.

### Lib

This directory 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 four files.

File	Description
<code>Sparx.jar</code>	This is the entire Sparx library compiled as a JAR file. This file is needed for all Sparx applications
<code>Oro.jar</code>	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
<code>Xalan.jar</code>	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.
<code>Xerces.jar</code>	The Xerces Java Parser 1.4.4 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
-----------	---

Out of these four files the first three are such that they can be placed in the application server's common lib directory where they will be available to all applications running under that server. Log4J.jar, however, is not such a library. Each application that needs it will be required to have its own local copy of it in the `WEB-INF/lib` directory.

### Security

This directory contains one file: `access-control.xml`. This is an XML definition of role-based security in 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.

### Log

This directory will contain all logs generated by Sparx. These logs are generated using the Log4J library mentioned earlier. The six different log files will contain the following information.

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.

### 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<sup>3</sup> 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 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. These Java classes needed to handle each tag should be located under the classes directory mentioned earlier; that way the classes will always be in the application's classpath.

## Application Configuration

Each Sparx application must have at least two files that are essential for it to work. The first, `web.xml`, is needed by the application server under which the application will run. The second, `configuration.xml`, is needed by Sparx to store configuration information related to the application.

### `web.xml`

This file, which is a requirement of the Java Servlet Specification<sup>4</sup>, 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.

### `configuration.xml`

This file, which is a requirement for all Sparx applications, contains a list of preferences necessary for a Sparx application to run. A default `configuration.xml` will contain information about all the paths and URLs required for Sparx to function with the specific application whose configuration file that is. As such this can also be used to store application-wide properties accessible to Sparx applications at runtime.

---

<sup>3</sup> To learn more about JSP Custom Tags, try the following URL:  
<http://java.sun.com/webservices/docs/ea1/tutorial/doc/JSPTags.html>



## Hello World

The “Hello World” example is a classic in nearly all texts dealing with programming. It is the simplest way to demonstrate the syntax of the language, 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. Whereas this example shows the simplicity of Sparx, it shall also serve as a stepping stone to a more complex example that will better illustrate the power of Sparx.

### 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.



**Figure 2:** A More Interactive Version With A Greeting Tailored To The Information You Provide

### 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 dialogs.xml file while the processors will be created by overriding the default Java dialogs with our own custom Java dialogs. 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 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 as suggested by the installer. We will also assume you have accepted the default port suggested by the Sparx evaluation kit installer. 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.

- ✍ Installation Path: `c:\Netspective`
- ✍ Sparx Install Path: `c:\Netspective\Sparx`
- ✍ Resin Install Path: `c:\Netspective\Resin`
- ✍ Hello World Install Path: `c:\Netspective\web-hello`
- ✍ Resin Web Server Port: 8089

Additionally, we will assume you are developing applications on 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 8089 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:8089/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.

## Setting up the Application

To minimize the startup time, we will use a shortcut built into Sparx. This will automatically create all appropriate directories for us along with blank files where they are needed. The procedure we will follow is detailed below.

1. Create a directory for the Hello World application. For this example, let's create a directory under the root directory and call it `web-hello`. Now all files related to the Hello World application will reside in the `c:\web-hello` directory.
2. Copy the application build batch file into the application root directory. This is accomplished using the following command:

```
copy c:\Sparx\tools\app-build.bat c:\web-hello\build.bat
```

The application build batch file is one of the most important components of your application. It is a convenient driver script for the Sparx Ant<sup>®</sup> build file. You will use this to compile your custom Java code as well as to generate and compile Sparx components related to your application. It will also serve as a quick sanity check for your application before trying to access it through a web browser.

3. Build the application directory structure and integrate the application into the application server by issuing the command:

```
build start-sparx-app
```

This will create all the directory structures that are necessary for the application as well as create empty files where they are needed.

4. Add a context section for this application in the application server configuration files. This would require opening up the Resin configuration file `server.xml` in a text editor such as Notepad. The full path of this configuration file is `c:\Netspective\Resin\conf\resin.conf`.

Once `server.xml` is open, look for the following text:

```
<web-app id='/examples/login'/>
```

This is a context declaration for the Resin manager directory. If you are familiar with XML notation, you will notice that this context is located within the default host as demonstrated by the presence of the `<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">` and `</Host>` tags that enclose this and other contexts.

Go to the end of this Host section (after all the Context sections are done but before the closing `</Host>` tag) and add the following line:

```
<web-app id='/hello' app-dir='c:\web-hello\Site'/>
```

Now save the `resin.conf` file and restart Resin so it can re-read its configuration and be aware of the new Hello World application.

Once you have restarted Resin with its configuration modified to know about our Hello World application, we can quickly check to ensure everything went smoothly by accessing the newly created application in a web browser. We can use two tests to check for a valid application setup:

5. 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://host:8080/hello`. If everything worked as it should, you will see a

web page rendered by Resin that will attempt to list all files and directories in your `c:\web-hello\Site` directory. Since the `WEB-INF` directory is meant for internal Resin (and Sparx) use, it will not be shown in the listing.

6. 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 relation to Sparx. In a web browser, then, we can go to the following URL: `http://host:8080/hello/ace`. 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.

Once you have assured yourself that the newly created application is working well so far, 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`. Once you're done getting a feel for the different options ACE has to offer, we can get started on the most visible aspect of this application: the user interface.

## Creating the User Interface

The user interface of the application is its most visible component, bar none. Creation of this interface is a three step process.

1. Create the XML definition of the interface in the `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. Adding 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.
3. Create the Java dialog 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.

### Creating the XML Dialog

To create the XML definition of the dialog, you will have to open up the application's main `dialogs.xml` file in a text editor such as notepad. 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 dialogs 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.

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. To get detailed descriptions of what each of these fields does, you can visit the Netspective web site at the following URL: <http://www.netspective.com>.

## Unit Testing with ACE

Now that you have finished adding the XML definition of the dialog to the `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.



**Figure 3:** Logging Into ACE. The Login is “ace” and the password is “ace”.

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> and

you will be presented with the ACE

login page. The default username for ACE is “ace” and the default password is “ace”. Use these defaults to login to 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.



**Figure 4:** Getting To The Application Dialogue Screen



**Figure 5:** List Of Application Dialogues.

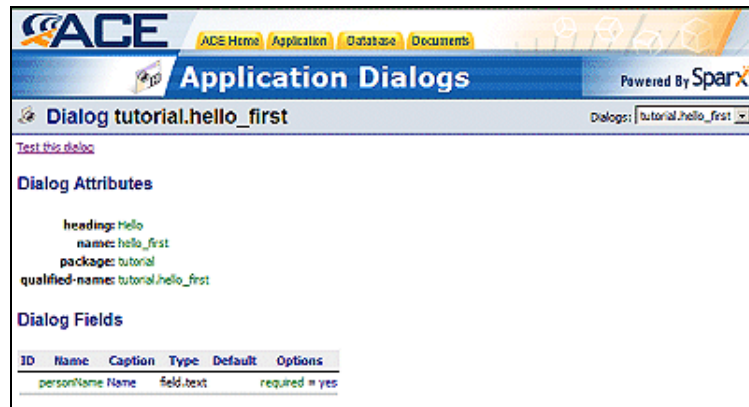
For each dialog, the Application Dialogs screen of ACE will give the following pieces of information.

- ✍ The option to run the dialog for testing purposes. 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.

- ✍ The option to see all options and components that comprise the dialog. This includes a complete field listing with per field options. This view is very helpful in verifying that a custom field was indeed recognized by Sparx and is active.
- ✍ The heading of the dialog as it will appear on the “title bar” of the dialog when displayed on a web page.
- ✍ 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.
- ✍ The number of fields that are a part of the dialog.
- ✍ 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.
- ✍ The custom Java dialog class that will be augmented by 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.
- ✍ The custom Java data bean (called a Dialog Context Bean) that allows seamless programmatic access to the dialog data and options. Similar to the default Java dialog, there is a default Java dialog context 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.
- ✍ 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.



**Figure 6:** tutorial.hello\_first Application Dialogue.

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.

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.



**Figure 7:** Testing The tutorial.hello\_first Dialogue

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`.

**Dialog: tutorial.hello\_first**

Need to add Dialog actions, provide listener, or override Dialog.execute(DialogContext).

<b>Dialog</b>	tutorial_hello_first
<b>Run Sequence</b>	2
<b>Active/Next Mode</b>	E -> V
<b>Validation Stage</b>	2
<b>Is Pending</b>	false
<b>Data Command</b>	[none]
<b>Populate Tasks</b>	none
<b>Execute Tasks</b>	none
<b>director</b>	null
<b>personName</b>	Bill
<b>XML Representation</b>	<pre>&lt;xaf&gt;   &lt;dialog-context name="hello_first" transaction="{B2ef2ada3}"&gt;     &lt;field name="personName" value-type="string"&gt;       &lt;value&gt;Bill&lt;/value&gt;     &lt;/field&gt;   &lt;/dialog-context&gt; &lt;/xaf&gt;</pre>



**Figure 8** Showing A Run Of The First Dialogue.

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 parameter that is a part of the dialog tag. Feel free to test the dialog as many times as you want.

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.

## 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 application's Site directory.

It should be pointed out that in the source shown below the most important line is the eighth line from the end which starts with the `<xaf: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.

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

<html>
  <head>
    <link rel='stylesheet'
      href='/hello/resources/css/main.css'>
    <title>Tutorial 1, Part 1: Hello World</title>
  </head>
  <body bgcolor='white' text='black' marginheight='0'
    marginwidth='0' topmargin=0 leftmargin=0>
    <table width='100%' border='0' cellpadding='0'
      cellspacing='0'>
      <tr>
        <td align='center' bgcolor='#660000' valign='center'
          width='750'>
          <font size='+3' color='white'>Tutorial I, Part I: Hello
            World</font></td>
      </tr>
      <tr>
        <td align='left' valign='top' width='750'>
          <table border='0' width='100%' cellpadding='5'
            cellspacing='0'>
            <tr>
              <td align='center' valign='top'>
                <xaf:dialog name="tutorial.hello_first"/></td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </body>
</html>
```

### Step by Step Explanation

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

The first line in the file is a JSP command (as signified by the `<%` and `%>` bracket style used in the tag) to include the Sparx Tag Library Descriptor 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.

```
<xaf: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 `xaf: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 embedding 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`.

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`.

Tutorial I, Part I: Hello World	
Need to add Dialog actions, provide listener, or override Dialog.execute(DialogContext).	
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	Bill
XML Representation	<pre>&lt;xaf&gt;   &lt;dialog-context name="hello_first" transaction="[B@16cd0da4"&gt;     &lt;field name="personName" value-type="string"&gt;       &lt;value&gt;Bill&lt;/value&gt;     &lt;/field&gt;   &lt;/dialog-context&gt; &lt;/xaf&gt;</pre>

**Figure 9:** Testing The Embedded Dialogue

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 tutorial;

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

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

public class HelloFirstDialog extends 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 in the `Site\WEB-INF\classes` directory to be included 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 `Site\WEB-INF\classes\tutorial\HelloFirstDialog.java` file for compilation and integration with the application.

#### Step by Step Explanation

```
package tutorial;
```

This line declares this class and all classes after this line to be a part of the “tutorial” 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 `tutorial` directory under the main `Site\WEB-INF\classes` 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 method declaration within the `HelloFirstDialog` class. The `execute` method, which is being overridden from the original `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 its 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 `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="tutorial.HelloFirstDialog">
```

This new line tells Sparx that the `hello_first` dialog will be handled by the Java class `tutorial.dialog.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.

## 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. However, if you are using Resin (the default application server that ships with Sparx) you will need to recompile your application. Fortunately, this process is not as cumbersome as it sounds. The build file that we used to create the directory structure for our application can be used to recompile our application too. In order to do this, you will need to go to the command line prompt and navigate to your application directory using the command

```
cd c:\web-hello\
```

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 etc. 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>.

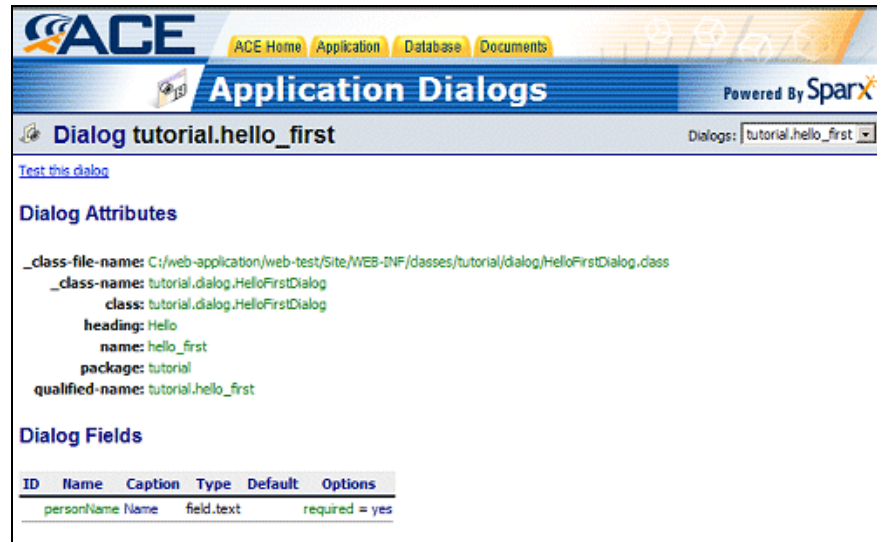


**Figure 10:** The Dialogue List With The Added class.

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 tutorial.dialog.HelloFirstDialog. We know now, therefore, that the



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

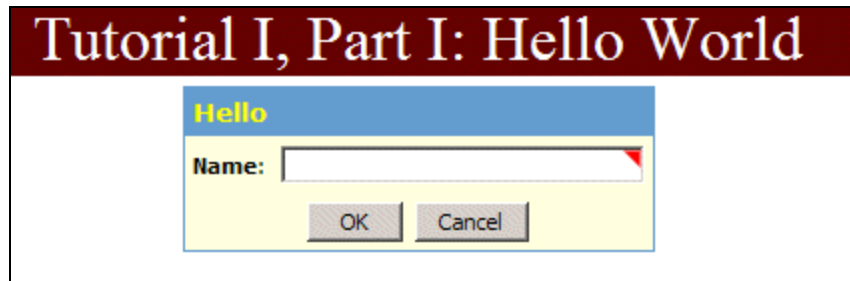


**Figure 11:** Field List With The Added Class

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` (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.



**Figure 12:** Final Dialogue Interface

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



**Figure 13:** Final Application Result

#### Success

With a total of 9 lines to define an XML dialog, 22 lines (excluding comments) to create a Java dialog handler class 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.

Now that you know how easy it is to create a full fledged application with Sparx and Resin, you can stop here and experiment with the setup yourself, referring back to this guide when necessary. Alternatively, you can continue on to the chapter on Advanced Dialogs. This chapter will cover some of the sample dialogs that are shipped with Sparx in a concise albeit complete manner.

## Advanced Dialogs

**T**he previous chapter 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 extremely 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.

### The 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 c:\Netspective\Sparx\test-dialogs.xml .
```

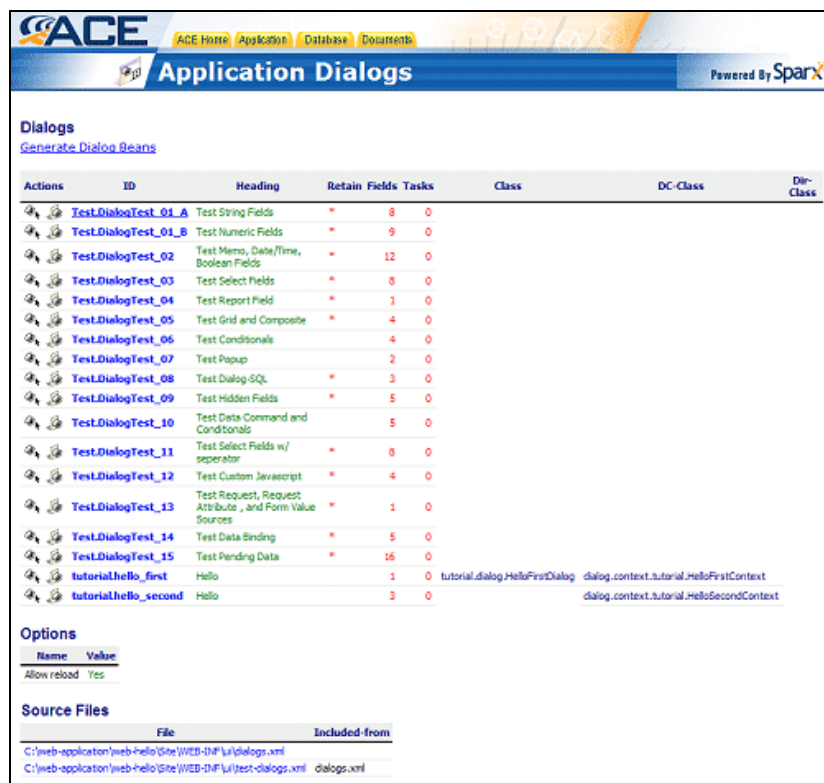
Your Sparx application's `Site\WEB-INF\ui` directory should now contain the `test-dialogs.xml` and `dialogs.xml` files in addition to any other files that might already be there. To automatically include all the test dialogs in your application, you need to add one line to your `dialogs.xml` file. Therefore 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 of its purpose. Let us go through each dialog and point out some of the important features.



The screenshot shows the ACE Application Dialogs interface. At the top, there's a navigation bar with 'ACE Home', 'Application', 'Database', and 'Documents'. Below this is a header 'Application Dialogs' with 'Powered By Sparx' on the right. The main section is titled 'Dialogs' and includes a link 'Generate Dialog Beans'. Below this is a table listing various dialogs with columns for Actions, ID, Heading, Retain, Fields, Tasks, Class, DC-Class, and Dir-Class. The table contains 15 test dialogs and 2 tutorial dialogs. Below the table, there's an 'Options' section with a table for 'Name' and 'Value', showing 'Allow reload' set to 'Yes'. At the bottom, there's a 'Source Files' section with a table showing the file paths for 'dialogs.xml' and 'test-dialogs.xml'.

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/ separator	*	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 14:** A List Of All new And Old Dialogues.

### 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**

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	requestid	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]+(\.[^\w+)]*)[A-Za-z0-9]+)\$/
email_field	Email		field.email		
		director			submit-caption = Submit

**Figure 15:** Test.Dialogue Test\_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**

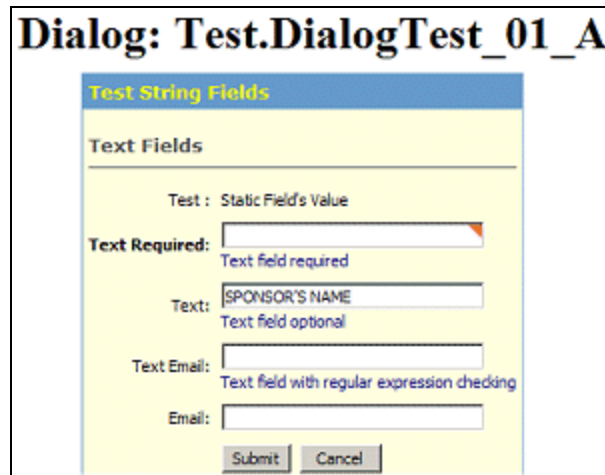
ID	Name	Caption	Type	Default	Options
			field.separator		heading = Text Fields
text_field_hidden			field.text	requestid	hidden = yes
static_field	Test		field.static	Static Field's Value	
text_field_required	Text Required		field.text		hint = Text field required required = yes
text_field	Text		field.text	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]+(\.[^\w+)]*)[A-Za-z0-9]+)\$/
email_field	Email		field.email		
		director			submit-caption = Submit

**Figure 16:** `text_field_hidden` and `text_field_email`

`text_field_hidden` is of interest primarily due to its 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.



**Figure 17:** The Dialogue Interface

#### 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 Test.DialogTest\_02**

Test this dialog

**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 18:** Test.Dialog Test\_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 19: 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.



**Dialog: Test.DialogTest\_02**

**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): 04-28-2002  
Format is MM-dd-yyyy

Date (Non-Strict Year): 04/29/2002  
Format is MM/dd/yyyy

Time: 12:14  
Format is HH:mm

**Boolean Fields**

Boolean Field (Radio): ☒ No ☐ Yes

Boolean Field (Alone): ☐

Boolean Field (Combo): No

Submit Cancel

**Figure 20:** Dialogue Interface

### 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`.


ACE Home
Application
Database
Documents



# Application Dialogs

Powered By Sparx


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: *
    
```

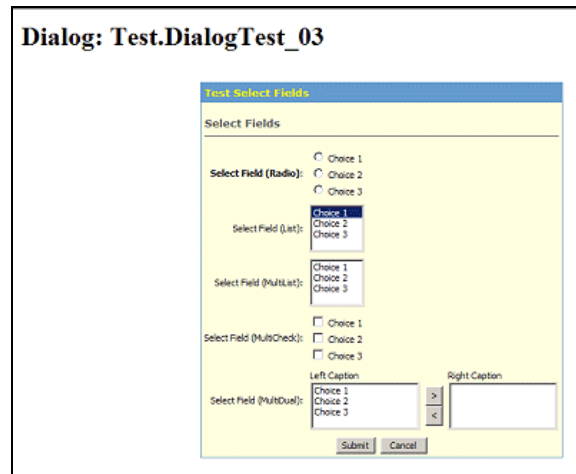
## Dialog Fields

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_multiselect	Select Field (MultiList)	field.select	B		choices = Choice 1=A;Choice 2=B;Choice 3=C size = 5 style = multiselect
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 21:** 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 22:** Dialog Interface

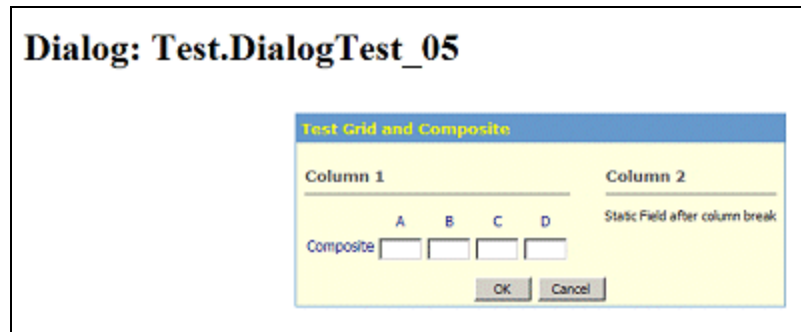
### Test Dialog 05

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.

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 23:** Dialogue Field List

Both grid and composite fields are a good way to create multi-widget fields can group lots of similar or 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 24:** Dialog Interface

### 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.

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 = checkbox
static_field2		field.static	conditional	Checkbox checked!	action = display js-expr = control.checked == true partner = checkbox_field

**Figure 25:** Dialog Field List

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.

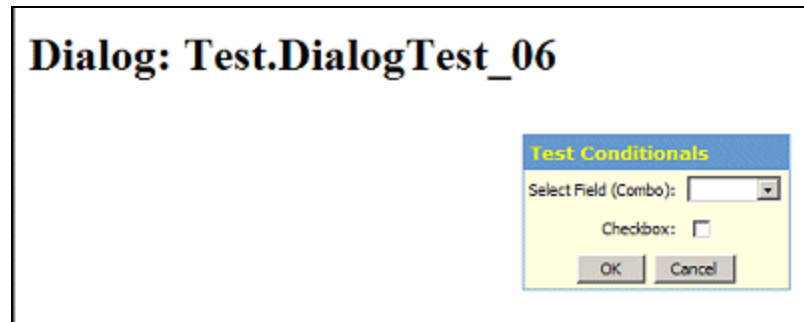


Figure 26: Dialog Interface

## 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.

## The Sparx Collection

**T**he Sparx Collection is a project meant to take you a few steps beyond the very 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 Collection will lead you through everything it takes to get an actual application up and running with special focus on developing the back end.

### Functionality

The Sparx Collection 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.

### Application Design

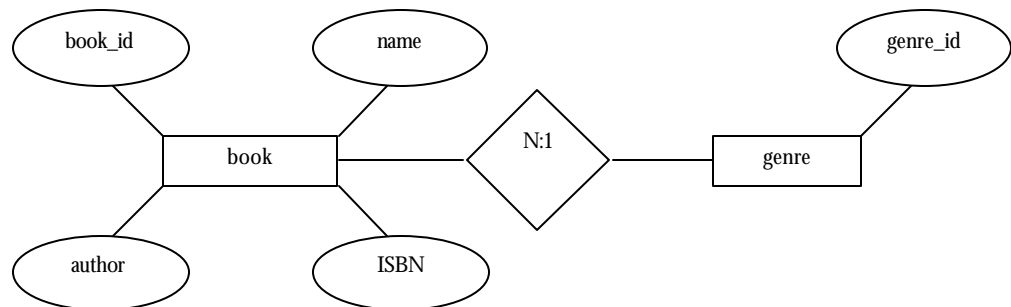
The Sparx Collection is designed around the basic Sparx components you have already seen and a few that you might not have worked with yet. The Collection's user interface will be a combination of XML dialogs for the front end data entry and validation while the back end will use Java dialogs for the 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 Collection is data storage. It is possible to use expensive and (in all certainty) overkill databases like Oracle for this application. Instead, the tool of choice is the completely Java-based embedded database that ships with the Sparx evaluation kit: HypersonicSQL. Sparx scales exceptionally well applications 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 Collection and setting the new database up with all existing data.

Application design will, of course, be dealt with in more detail when it is implemented later in this chapter.

### Database Design

The collection 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 Collection 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 has a one-to-many relationship<sup>4</sup> with the books.



The figure shows the entity-relationship diagram for the data we will be using. The database for the Collection 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.

### Implementation

When describing the implementation of The Sparx Collection, 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

---

<sup>4</sup> 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](http://www.devshed.com/Server_Side/MySQL/Normal/Normal1/print)

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 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.

- ✍ Installation Path: `c:\Netspective`
- ✍ Sparx Install Path: `c:\Netspective\Sparx`
- ✍ Resin Install Path: `c:\Netspective\Resin`
- ✍ Application Install Path: `c:\Netspective\web-library`
- ✍ Resin Web Server Port: 8089

## Starting the Application

Creation of a Sparx application involves the same basic steps that you went through when creating the Hello World application. It involves creating a directory for the application, copying over the build batch file and executing it. 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\web-library` to store all files related to the Sparx Collection. Additionally, you should ensure that the Sparx Collection is configured to have a context path of `/library` in the Resin `server.xml` file. 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 Collection: the database. There are two steps towards setting up the database.

1. 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 `Database` directory immediately under the Application Root.
2. 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.

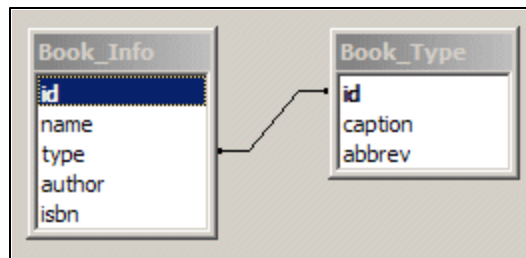


Once you have a data source configured, you can use ACE to verify that it is indeed recognized as a valid data source and that the JDBC driver for the source has been located and bound to the data source. To do this, open the Sparx Collection ACE in a web browser. After logging in, go to the Database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

If everything is configured properly, the Sparx Collection 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



**Figure 27:** Table Structure

necessary for the Sparx Collection. It is a very simple schema consisting of only two tables, one to store information about the different genres of books and the other to store all four attributes of the books in the library. The two tables are 1:n related by the genre\_id stored in both tables. This structure is shown below.

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. 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.

This XML schema is stored in the `Database\schema-src` directory of the Sparx Collection in the file called `schema.xml`. 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 Collection you should ensure that these files are created, with these exact contents, in the `Database\schema-src` directory of the Sparx Collection.

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"/>
  <column name="name" type="text" size="64" descr="Name of
the book"/>
  <column name="type" lookupref="Book_Type"/>
  <column name="author" type="text" size="64" descr="Name of
the author(s)"/>
  <column name="isbn" type="text" size="10" descr="The 10
digit ISBN number"/>
</table>
</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 Collection 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. These attributes are:

- ✍ `name`: This is a required attribute and specifies the name of the field in the table. This name is for use not just by Sparx but it is carried on and used as the name of the field in the DBMS itself.
- ✍ `type`: This is also a required attribute and specifies the type of field you want to create. Text fields are translated into the SQL `varchar`s. You can get an idea of all the fields available for use with this attribute by looking at the `datatypes.xml` file.
- ✍ `descr`: This is an optional attribute that helps tremendously in documenting the schema. This information is present in all automatically generated documentation for the schema.
- ✍ `size`: This attribute is an optional attribute that is required for text fields. It determines the length of the field as created in your DBMS.

Other attributes that appear in the `schema.xml` file are more specific. These are explained below.

- ✍ `primarykey`: This is an attribute that designates a field as the primary key for that table. Among other things, this implies that each record in the entire table will have a unique value for that field that will identify the record uniquely.
- ✍ `lookupref`: This is an attribute that packs a lot of power. Its purpose is to link the current table with the one specified as the value of the `lookupref` attribute in an n:1 relationship. The table specified in the

attribute has to be a special type of table known in Sparx applications as an Enumeration table. Enumerations will be explained in some detail a little further down. Suffice it to say that in this particular context, the `lookupref` attribute is saying that the value of the `type` field has to be one of the values stored in the `Book_Type` table (which stores information about the book genres).

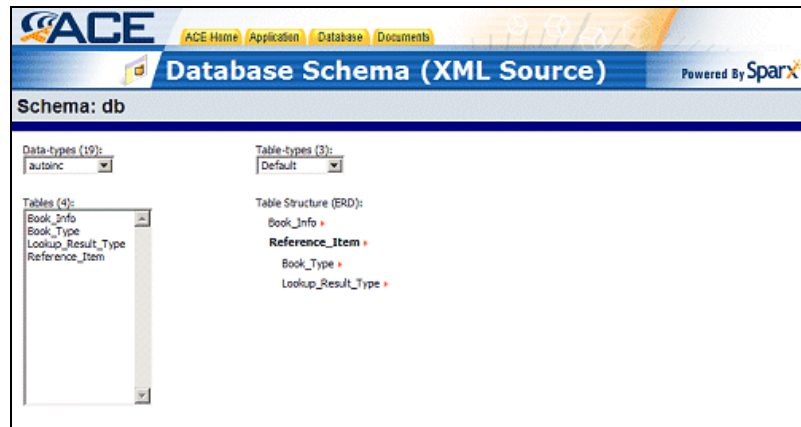
The second table in our schema, the `Book_Type` table, is used to store information about the different genres of books stored in the Sparx Collection. This table is defined as an Enumeration to Sparx and is conventionally stored in the file `enums.xml` that is then included in the `schema.xml`.

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.

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 Collection ACE in a web browser and, after logging in, choose the item “Schema (XML)” from the Database menu.



**Figure 28:** 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.

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

**Figure 29:** Book\_Info Table

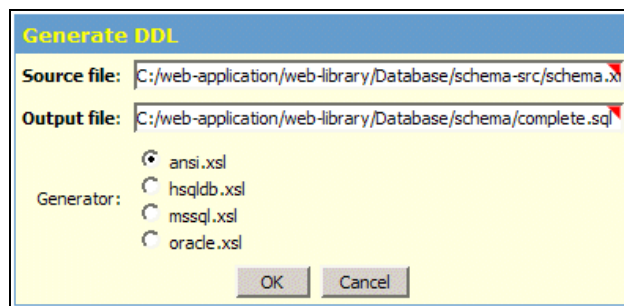
As you can see, the detailed view of the `Book_Info` table gives a lot of information about the table and the information stored in it. For each field in the table you can see its name (and description), the data type it was declared as (imported from `data-types.xml`), its default value (if any), the 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 Collection 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

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 what is known as a Database Policy to translate its internal representation of your database schema into the DDL appropriate for your DBMS. Thus it is able to maintain a platform independent XML representation for internal and developer use while retaining the ability to generate a platform specific representation as needed.



**Figure 30:** Generate DDL

To generate the DDL for the HypersonicSQL DBMS, get back into the Sparx Collection 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 `Database\schema` 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 Collection’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 Collection. 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.

## Creating the User Interface

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 Collection.

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

<xaf>
  <dialogs package="library">
    <dialog class="library.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 ValueSources 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 Collection 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 `ids` and `captions` 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 31:** 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 (`library.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 Collection. 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 Collection 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 Collection's `WEB-INF\classes` directory. A step by step explanation of the code will follow immediately after it.

```
package library;

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

import com.netspective.sparx.xif.dal.ConnectionContext;
import com.netspective.sparx.xaf.sql.DmlStatement;
import dal.DataAccessLayer;
import dal.table.BookInfoTable;
import dal.domain.row.BookInfoRow;

import dialog.context.library.BookInfoContext;

import javax.naming.NamingException;
```



```

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.sql.SQLException;
import java.util.Date;
import java.net.URL;
import java.io.Writer;
import java.io.IOException;

public class BookInfo extends Dialog {

    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()) {
            BookInfoContext 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 ...
                BookInfoRow 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();
            } catch (SQLException se) {
                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);

    // 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);

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

/**
 * Process the new data
 */
protected boolean processAddAction(Writer 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();
    } 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();
    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();
    } catch (NamingException ne) {
        ne.printStackTrace();
    } catch (SQLException se) {
        se.printStackTrace();
    }
}

return status;
}

/**
 * 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();

        // Create a new BookInfo record and insert it...
        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;
}
}

```

### Step by Step Explanation

There are five major parts to the source code of the `library.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.

- ✍ `populateValues`: This procedure 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 procedure 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 `library.BookInfo` class.
- ✍ `processAddAction`: This procedure is called from `execute` and is responsible for processing data that is meant to be added to the database.
- ✍ `processEditAction`: This procedure is called from `execute` and is responsible for processing a request to update an existing database record.

- ✍ `processDeleteAction`: This procedure is called from `execute` and is responsible for processing a request to delete a record from the database.

### Dissecting `populateValues`

```
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()) {
        BookInfoContext 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 ...
            BookInfoRow 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();
        } catch (SQLException se) {
            se.printStackTrace();
        }
    }
}
```

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.

### 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";

    // What to do if the dialog is in add mode ...
    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);

    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 Collection. In essence, then, it allows you to add, edit or delete books and, once you’re done, it redirects you to the Sparx Collection’s home page.

```

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

```

The first of these two lines instantiates a low-level `HttpServletRequest` object to gather information about the Sparx Collection. This information is provided to the `HttpServletRequest` object by the Resin application server. The second line uses that object to find out what the Collection uses as its context path. In this case, it is the `/library` prefix that is a part of all URLs belonging to the Sparx Collection 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();
    } catch (NamingException ne) {
        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.

```

        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 library\BookInfo.java file under the WEB-INF\classes directory of the Sparx Collection, 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 Collection. 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 Collection’s ACE. 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 32:** 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 ValueSource (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.

The dialog box has a blue header bar with the title 'Add Book Information' in yellow. Below the header, there are five input fields, each with a label and a red arrow icon on the right: 'Book ID:', 'Name:', 'Author:', 'Genre:', and 'ISBN:'. The 'Genre:' field is a dropdown menu. At the bottom of the dialog are two buttons: 'Save' and 'Cancel'.

**Figure 33:** The Dialog In Add Mode

- ✍ Notice that now the dialog heading reads “Add Book Information”. This implies that the create-data-cmd-header ValueSource has successfully detected the current mode the dialog was called in and added an appropriate prefix to the heading.
- ✍ Add a book to the Collection’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 Collection’s ACE test the only dialog listed in the Application Dialogs page.

The dialog box has a blue header bar with the title 'Edit Book Information' in yellow. Below the header, there are five input fields, each with a label and a red arrow icon on the right: 'Book ID:', 'Name:', 'Author:', 'Genre:', and 'ISBN:'. The fields are pre-filled with the following values: 'TESTBOOKID', 'Another Sample Book', 'A Sample Author', 'Chemistry' (in the dropdown menu), and '2131452341'. At the bottom of the dialog are two buttons: 'Save' and 'Cancel'.

**Figure 34:** Dialog In Edit Mode

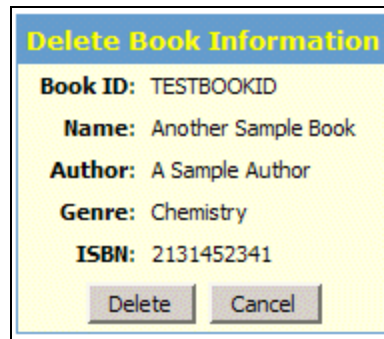
- ✍ In the new window that pops up (the test window) change the URL by appending `?data_cmd=edit&bookid=YOURBOOKID` to it. Make sure you replace the word `YOURBOOKID` with the Book ID you chose while testing the dialog’s Add mode above. Thus, if the book you added to the Collection earlier had a Book ID of `TESTBOOKID`, you will append

?data\_cmd=edit&bookid=TESTBOOKID to the URL shown in the test window.

- ✍ If your previous test to add a book to the Collection 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.

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 Collection’s ACE test the only dialog listed in the Application Dialogs page.



**Figure 35:** 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 Collection 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 a Static SQL Library

The Sparx Collection 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 two different forms of SQL: static and dynamic.

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 Collection. Static SQL libraries are stored in the `statements.xml` file in the `sql` directory of the Collection. The `sql` directory is directly under the `Site\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 Collection. Following this XML definition of the Collection'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
            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 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,
    book_info.id
  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;&gt;
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. 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>
  <column heading="column 0" />
  <column heading="column 1" />
  <column heading="column 2" />
  <column heading="column 5" index="5" />
  <column heading="column 6" />
</report>

```



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 fifth column 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.

```
<column align="center"
        heading=" "
        index="0"
        output="<a href=
        "bookInfo.jsp?data_cmd=edit&bookid=${0}">edit</a>" />
```

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

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




Similarly, this XML snippet from the fifth 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 0<sup>th</sup> column allows you to delete the record shown in that row of the table while the output format of the 5<sup>th</sup> 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 Collection and to search the Collection 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 Collection's `Site\WEB-INF\sql` directory, you will be able to see the statements in action by going to the Collection's ACE. 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 36:** 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.

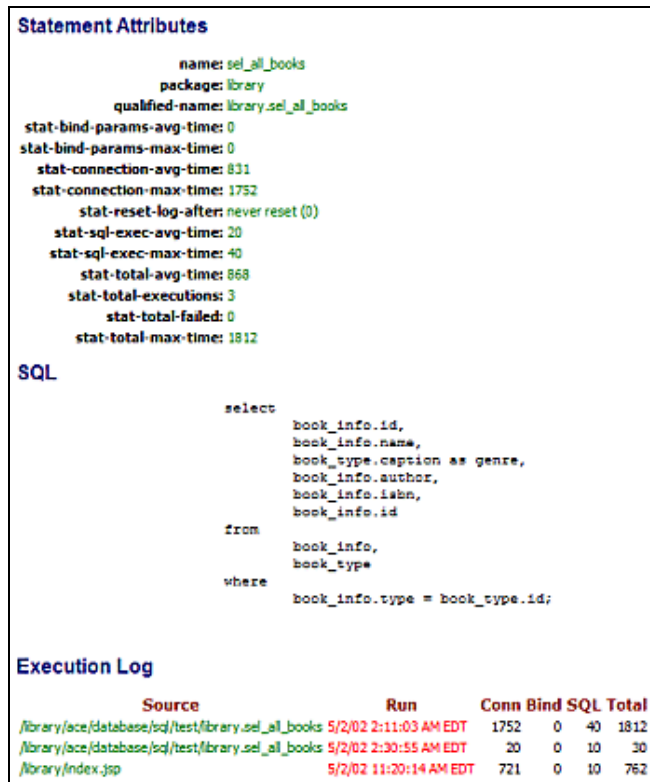


Figure 37: Statement Details

- ✍ Use your browser's back button to go back to the list of statements. Instead of clicking on the statement name, click on the icon on the extreme left of the table row containing the statement.

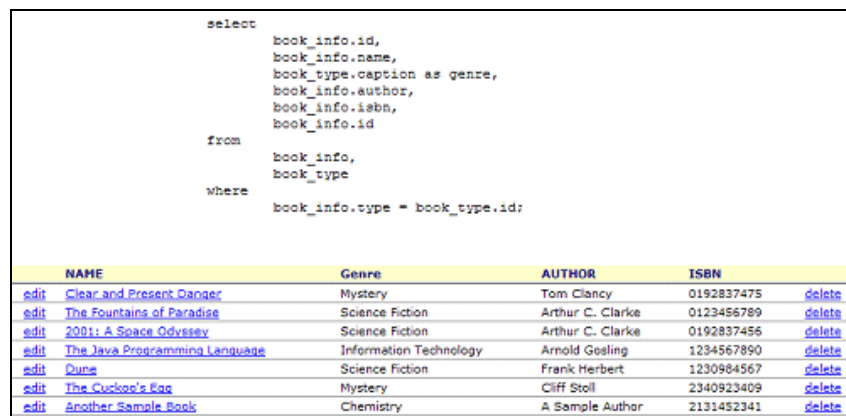


Figure 38: List Of Statements

- ✍ 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

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 Collection 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 Collection 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 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"/>
```

- ✍ **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"/>
  <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>

```

 **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">
    <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>

```

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.

```

<display field="book_id"/>
<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" value="form:book_author"
    connector="and"/>
    <condition 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 Collection, this is not needed. If set to “no”, which is the case for the Collection, 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 39:** Unit Testing In ACE

Now in a web browser open up the Collection’s ACE 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 Collection, 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 40: 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 Collection, 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.

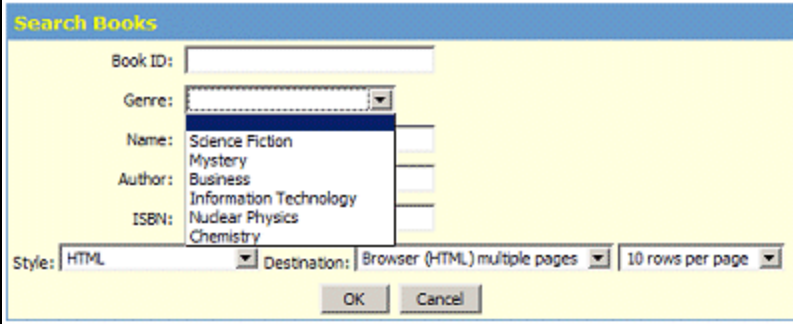


Figure 41: Select Dialog You Created

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

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

Search Books	
Book ID:	
Genre: Science Fiction	
Name:	
Author:	
ISBN:	
Done	

**Figure 42:** 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.

### The Collection Integrates

Now that the major pieces of the Sparx Collection 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.

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 Collection 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 Collection's

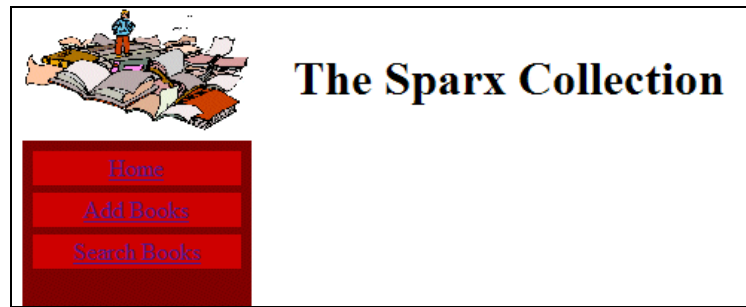
functionality is created using Sparx components, they need to be embedded in known JSP pages so you can create links to them in the JSP custom tag template.

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 Collection 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 Collection 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 43:** 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.

```
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
{
    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";

            out.println("<!DOCTYPE html PUBLIC \\"-//W3C//DTD HTML
4.01 Transitional//EN\>");
            out.println("<html>");
            out.println("<head>");
            out.println("    <title>" + getTitle() + "</title>");
            out.println("    <link rel='stylesheet' href='"+
resourcesUrl + "/css/library.css'>");
            out.println("");
            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\" width=\"100%\">");
            out.println("        <tbody>");
            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>");
            out.println("                <td valign=\"middle\"
align=\"center\">");
            out.println("                    <h1>" + getHeading() + "</h1>");
            out.println("                </td>");
            out.println("            </tr>");
            out.println("            <tr>");
            out.println("                <td valign=\"top\"
bgcolor=\"#800000\">");
            out.println("                    <table width=\"100%\" border=\"0\"
cellspacing=\"5\" cellpadding=\"2\">");
            out.println("                        <tbody>");
            out.println("                            <tr>");
            out.println("                                <td valign=\"middle\"
bgcolor=\"#cc0000\" align=\"center\"><font color=\"#ffffff\"><a
href=\"\" + rootPath + "/index.jsp\">Home</a></font><br>");
            out.println("                                    </td>");

```

```

        out.println("                </tr>");
        out.println("                <tr>");
        out.println("                    <td valign=\"middle\"
bgcolor=\"#cc0000\" align=\"center\"><font color=\"#ffffff\"><a
href=\"\" + rootPath + "/bookInfo.jsp?data_cmd=add\">Add
Books</a></font><br>");
        out.println("                </td>");
        out.println("                </tr>");
        out.println("                <tr>");
        out.println("                    <td valign=\"middle\"
bgcolor=\"#cc0000\" align=\"center\"><font color=\"#ffffff\"><a
href=\"\" + rootPath + "/search.jsp\">Search
Books</a></font><br>");
        out.println("                </td>");
        out.println("                </tr>");
        out.println("");
        out.println("            </tbody>");
        out.println("        </table>");
        out.println("        <br>");
        out.println("    </td>");
//        out.println("    <td valign=\"top\"
bgcolor=\"#800000\"><br>");
        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
        return EVAL_BODY_INCLUDE;
}

public int doEndTag() throws JspException
{
    JspWriter out = pageContext.getOut();
    try
    {
        out.println("    </td>");
        out.println("    </tr>");
        out.println("");
        out.println("    </tbody>");
        out.println("</table>");
        out.println("    <br>");
        out.println("    <br>");
        out.println("</body>");
        out.println("</html>");
    }
}

```

```

        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 Collection) 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 `Site\WEB-INF\classes\app\tag` directory so it is be available to all pages that require it.

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="xaf" 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">
    <xaf:query name="library.sel_all_books"/>
</app:page>

```

#### bookInfo.jsp

```

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

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

```

#### search.jsp

```

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

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

```

#### viewBook.jsp

```

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

```

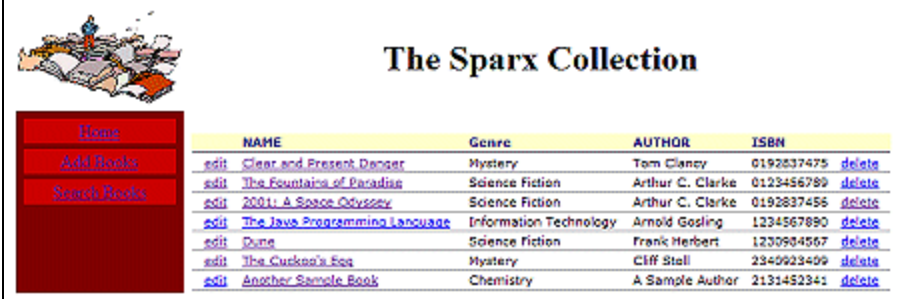


```

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

```

## Conclusion

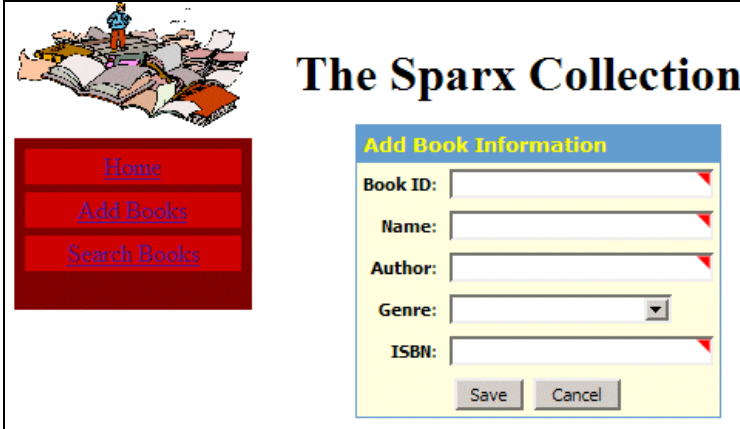


The screenshot shows the 'The Sparx Collection' web application. On the left is a navigation menu with 'Home', 'Add Books', and 'Search Books'. The main area displays a table of books with columns for NAME, Genre, AUTHOR, and ISBN. Each row includes 'edit' and 'delete' links.

NAME	Genre	AUTHOR	ISBN
<a href="#">edit</a> <a href="#">Clear and Present Danger</a>	Mystery	Tom Clancy	0192837475 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">The Fountains of Paradise</a>	Science Fiction	Arthur C. Clarke	0123456789 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">2001: A Space Odyssey</a>	Science Fiction	Arthur C. Clarke	0192837456 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">The Java Programming Language</a>	Information Technology	Arnold Gosling	1234567890 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">Dune</a>	Science Fiction	Frank Herbert	1230984567 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">The Cuckoo's Egg</a>	Mystery	Cliff Stoll	2345678901 <a href="#">delete</a>
<a href="#">edit</a> <a href="#">Another Sample Book</a>	Chemistry	A Sample Author	2131452341 <a href="#">delete</a>

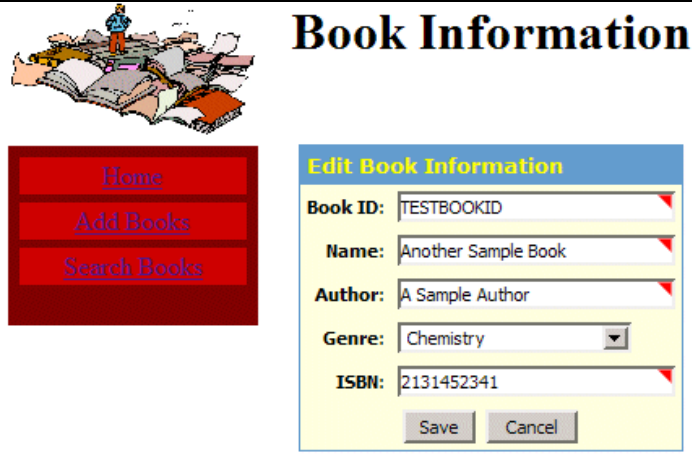
**Figure 44:** The Completed Sparx Collection

Congratulations! The Sparx Collection is complete. Now you should open up your browser window and go to the Collection's main page. 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.



The screenshot shows the 'The Sparx Collection' web application with the 'Add Book Information' form. The navigation menu on the left remains the same. The form on the right has fields for Book ID, Name, Author, Genre (a dropdown menu), and ISBN, each with a red arrow icon. At the bottom of the form are 'Save' and 'Cancel' buttons.

**Figure 45:** Adding A Book To The Collection



**Book Information**

Home  
Add Books  
Search Books

**Edit Book Information**

Book ID: TESTBOOKID

Name: Another Sample Book

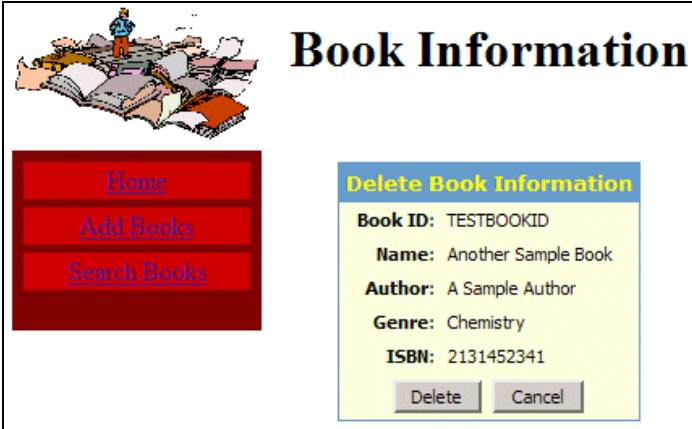
Author: A Sample Author

Genre: Chemistry

ISBN: 2131452341

Save Cancel

**Figure 46:** Editing A Book In the Collection



**Book Information**

Home  
Add Books  
Search Books

**Delete Book Information**

Book ID: TESTBOOKID

Name: Another Sample Book

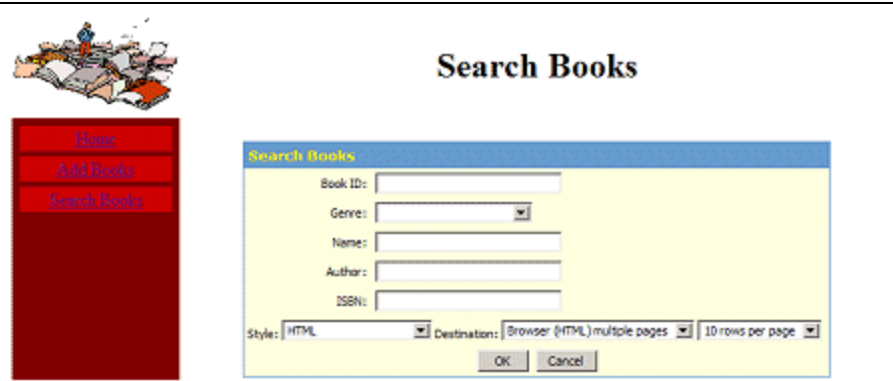
Author: A Sample Author

Genre: Chemistry

ISBN: 2131452341

Delete Cancel

**Figure 47:** Deleting a Book From The Collection



**Search Books**

Home  
Add Books  
Search Books

**Search Books**

Book ID:

Genre:

Name:

Author:

ISBN:

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

OK Cancel

**Figure 48:** Searching For Books In The Collection

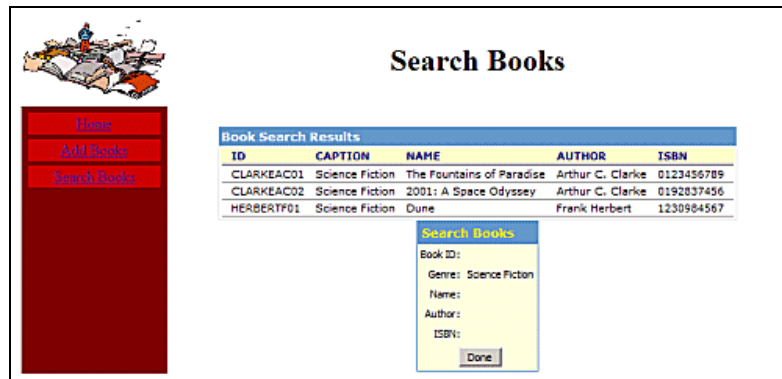


Figure 49: Search Books Results

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 Collection 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.

