Making Sparx Fly Higher

*Sparx Application Platform
Developer's Guide Volume II*

Netspective

Contents

The Cura Tutorial

Cura is the name of the third and largest application bundled with the Sparx evaluation kit. From the functionality standpoint, Cura is a project management application. It allows management of individual people, projects and clients. From the utility standpoint, however, Cura is an exceptional example application that demonstrates a large number of Sparx development Best Practices. It also demonstrates alternative development styles that serve two purposes.

♦   First, they show the variety of ways in which Sparx can be used to develop your own applications. They show the highest and lowest level of development possible and a few levels in between.

♦   Secondly, they show the immense difference in ease, readability and speed that is gained by using the higher level Sparx functionality to replace low level code. Indeed you are encouraged to rewrite any part of Cura, using a style different from the original, to show yourself how easy or difficult any alternatives are.

In short, Cura is a very usable full-blown application that shows what a few days of Sparx development can accomplish.

Setting up Cura

While Cura comes as part of the Sparx evaluation kit, its complex nature makes it difficult to be setup automatically by the evaluation kit installation program. Specifically, the database features that Cura requires to function properly are not available on the embedded HypersonicSQL database that is bundled with the other applications in the evaluation kit. Therefore, in order to use Cura, you are required to use an Oracle database. You can install Oracle on the same machine as the evaluation kit. Alternatively, you can use an existing installation of Oracle on another machine.

Having ensured that an Oracle database server is available for use, you only have two tasks to accomplish before you can start using Cura. Your first task is to setup an Oracle database with all the tables and data that the application needs. This is done in two steps.

♦   Use Cura's ACE to generate all the Oracle-compatible SQL for creating tables and inserting data into these tables.

♦   Execute this SQL inside Oracle as an administrator to automatically setup the database with all required tables and data.

The second task is to make Cura aware of the newly setup database and of how to use it. This task will involve modifying the web.xml file for web-cura. The relevant configuration directives should already be there but commented out.

Generating the SQL

When you get Cura as part of the Sparx evaluation kit, the complete SQL for Oracle is already generated and ready to be run on your Oracle database. However, in case you need to re-generate it at a later date, you can run through this brief procedure to do so.

To generate the complete SQL representing the database schema for any Sparx application, you need to first ensure that the application has been built using the build script. This build process ensures that the application is aware of all the changes that have been made in the schema or in the XML dialogs or statements and is able to auto-generate all components of the application that are dependent upon them. You can invoke the build script by dropping to a command prompt, changing to Cura's home directory and typing `build`.

Once this process terminates without an error, you are ready to generate the complete SQL for Cura. Open up a browser and access Cura's ACE. Typically, the URL to access the ACE would look similar to `http://localhost:8089/cura/ace` unless you have customized your installation of the evaluation kit or are accessing the installation from a remote location. In either case, substitute the appropriate values for the evaluation kit server and port in the above URL. If you are evaluating Sparx online, you should access Cura's ACE by pointing your web browser at the following URL: `http://developer.netspective.com:8089/cura/ace`.
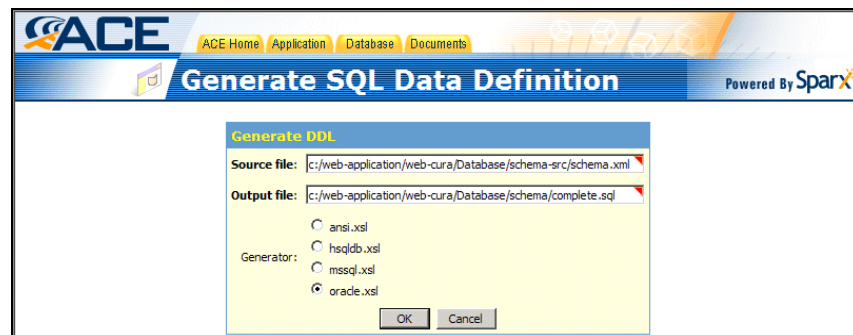
Figure 1: Generating The SQL Data Definition

Log into ACE using `ace` as the username and password. Choose the "Generate SQL DDL" option from the Database menu. This should bring up the screen that allows you to generate SQL targeted towards all the databases supported by Sparx. The two fields in the dialog presented allow you to specify the full path of the file used to store this generated SQL and the type of SQL that will be generated. While you are free to store the generated SQL anywhere on the filesystem, the preferred location is the `Database` directory inside your application's home directory. Further, for the purpose of this exercise, you should ask the SQL generator to write SQL conformant to Oracle's needs.
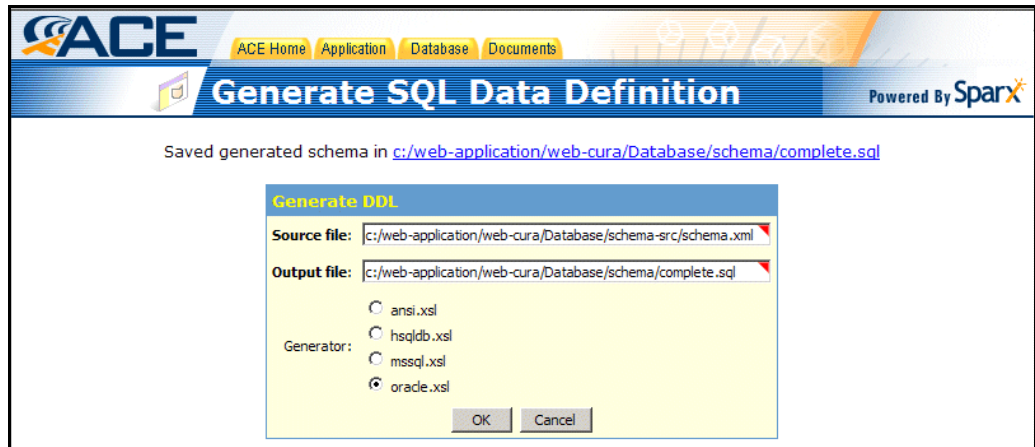
Figure 2: Click OK To Generate the DDL

Click on **OK** to generate the DDL and it should report that everything went ok.

The SQL below is taken from the complete.sql file that is created when Cura's ACE generates the complete SQL from the XML schema. It shows what is generated for the Person table. You should compare this to the Person table's XML source.

```
create table Person
(
  cr_stamp date DEFAULT sysdate,
  cr_person_id number(16),
  cr_org_id number(16),
  record_status_id number(8) DEFAULT 0,
  person_id number(16),
  name_prefix varchar(16),
  name_first varchar(32),
  name_middle varchar(32),
  name_last varchar(32),
  name_suffix varchar(16),
  short_name varchar(42),
  simple_name varchar(96),
  complete_name varchar(128),
  short_sortable_name varchar(42),
  complete_sortable_name varchar(128),
  ssn varchar(11),
  gender number(8) DEFAULT 0,
  marital_status number(8) DEFAULT 0,
  date_of_birth date,
  age number(8)
);
create index Per_ssn on Person(ssn);
alter table Person modify (person_id constraint Per_person_id_REQ NOT
        NULL);
alter table Person modify (name_first constraint Per_name_first_REQ
        NOT NULL);
alter table Person modify (name_last constraint Per_name_last_REQ NOT
        NULL);
alter table Person add (constraint Person_PK PRIMARY KEY
        (person_id));
```

As you can see, the generated SQL is very complete and is able to ensure an accurate translation of your XML schemas into the appropriate database SQL.

Installing the SQL in Oracle

With the SQL generated, you are already half way there. If you look into the `Database\schema` directory in the web-cura application directory you should notice three files. These files will be called `server.sql`, `complete.sql` and `test-data.sql`. Out of these, `complete.sql` contains the DDL generated by Cura's ACE. `test-data.sql` contains sample data provided with Cura that you can use to play with it. Finally, `server.sql` contains all the commands needed to create a new user named `cura` and run the `complete.sql` and `test-data.sql` files in that order.

In order to run these three SQL scripts, you need to move them to a computer that has the `sqlplus` client that comes with Oracle. Start `sqlplus` and log into Oracle as an administrator. Once you are logged in, ensure that all three files mentioned above are in the same directory as where you started `sqlplus` from. At the `sql>` prompt type `start server` and press enter. This should run the `server.sql` file to create a user named `cura` and create all necessary tables as well as populate them with sample data. You are now ready to let Cura know of this newly created datasource.

Configuring Data Sources in Cura

Having setup Oracle for Cura, you are now ready to configure Cura for your Oracle database. This is done by editing the Cura's `web.xml` file (located in the `WEB-INF` directory) to add a data source for the application. The relevant lines should already be a part of the configuration file but should be commented out. The commented out version of this configuration is shown below. You should know that the lines shown below are used only when Cura is being run using the Resin application server bundled with the Sparx evaluation kit. You can find out how to create datasources for other application servers by referring to the documentation that comes with the evaluation kit for alternate application servers.

```
<!--
<context-param>
  <param-name>default-data-source</param-name>
  <param-value>jdbc/cura</param-value>
</context-param>

<resource-ref>
  <res-ref-name>jdbc/cura</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <init-param driver-name="oracle.jdbc.driver.OracleDriver"/>
  <init-param url="jdbc:oracle:thin:@127.0.0.1:1521:oracle"/>
  <init-param user="cura"/>
  <init-param password="cura"/>
  <init-param max-connections="20"/>
  <init-param enable-transaction="false"/>
</resource-ref>
-->
```

Uncomment this block of the `web.xml` file by removing the `<!--` and `-->` in the first and last lines. Most of the datasource configuration will remain identical. However, the line shown below will need to be changed to point to the IP, port and SID of your Oracle installation. A brief explanation of these configuration directives is given below.

```
<context-param>
```

```
    <param-name>default-data-source</param-name>
    <param-value>jdbc/cura</param-value>
</context-param>
```

This section declares the default data source for an application. In this case, the default data source for Cura will be the one named `jdbc/cura`. A datasource is, in essence, a conduit for retrieval of data. It could be a proper database (most commonly) or a flat-file database or just a set of classes that will return data when queried. This data can then be used in the application. While an application can have many different datasources, each one dealing with a separate store of data, there can be only one default datasource in an application. It is generally a good idea to have a default datasource so you can fall back on it when all other datasources fail.

```
<resource-ref>
  <res-ref-name>jdbc/cura</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <init-param driver-name="oracle.jdbc.driver.OracleDriver"/>
  <init-param url="jdbc:oracle:thin:@127.0.0.1:1521:oracle"/>
  <init-param user="cura"/>
  <init-param password="cura"/>
  <init-param max-connections="20"/>
  <init-param enable-transaction="false"/>
</resource-ref>
```

This section actually declares a datasource, in this case the one name `jdbc/cura`. The main parameters that are used to define the datasource are:

♦ driver-name: This is the name of the Java class that serves as a JDBC driver for that database. This class needs to be in the classpath of your application (or your application server) and needs to be specified as a fully qualified name here.

♦ url: This is the string used by the JDBC driver specified above to connect to the database server. This string is different for each JDBC driver. The format that you should use for the url string for Oracle databases is: jdbc:oracle:thin:@<server.ip>:<server.port>:<server.sid>.

♦ user: The username to be used to connect to the database.

♦ password: The password for the username that will be used to connect to the database.

The rest of the parameters used in defining the data source are database-specific parameters that can, for now, be taken as is.
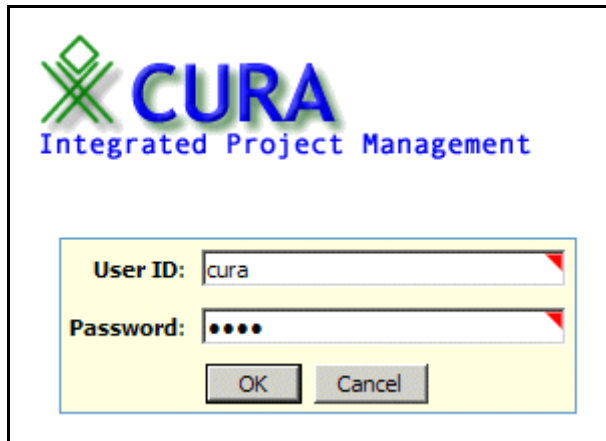
Having configured Cura for the data source, you are now ready to explore the application in detail.

Cura's Functionality

Cura, as a project management application, is fairly complete and usable for most small to medium scale projects. Cura can be used by multiple users each working for multiple clients. Further, a user can track many different projects for each of his clients making Cura the one stop shop for all his project management needs.

If you test-drive Cura for a few minutes you will notice a few key elements that make Cura so useful. The login for the default user in Cura is cura and the default password for this user is cura.
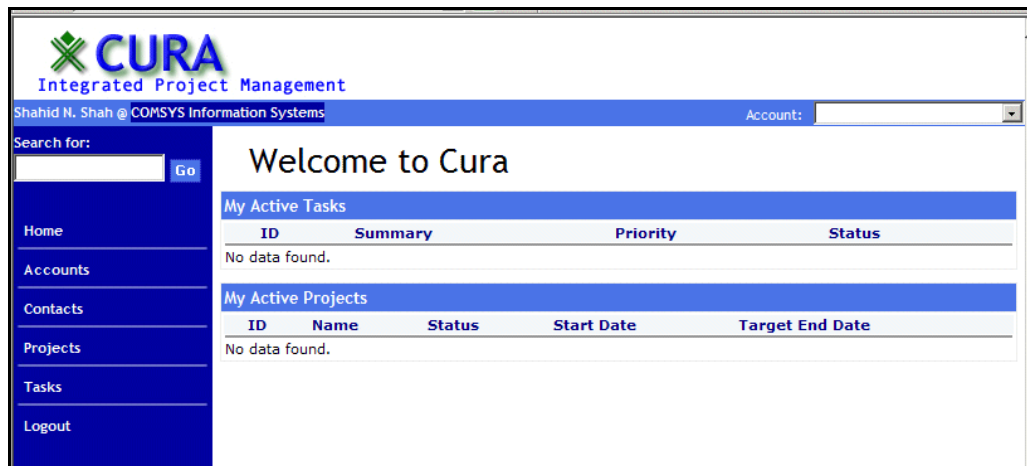
The first thing you will notice is the fact that there is a login page at all. The Sparx Collection project that forms the basis of the advanced tutorial in Volume 2 does not contain such a feature. By enforcing users to login to Cura, the application gets the ability to track users by their username and therefore allow multiple users to store and access their own data in the application.

The introduction of such a capability also raises the specter of security considerations within the application. After all, if a user is storing data related to his work in the database along with the data of many other users, he would not want any of those other users to access or modify that data. This is where the security features of Sparx come into play as we will see in the implementation of Cura.

After logging into Cura, the application shows you a very informative summary page that lists not only all your active projects and tasks but also brief information about yourself. From here you can access all the features of the application.

Figure 4: CURA Summary Page

The bar on the top – that displays information about you – and the menu on the left are a part of every page displayed by the application. Therefore, you can access any part of the application from anywhere. Additionally, you will notice that on the extreme right side of the top bar, there is a combo box that allows you to choose between your different clients. This allows you a lot of flexibility since you can just change the client name from the top bar and immediately watch the information displayed on the current page change to become relevant to the client you selected.

Going through the menu bar on the left, you can see that clicking on each link takes you to an important part of the application. The details of each link are listed below.

♦ The "Home" link will take you to the summary page displayed when you first login.
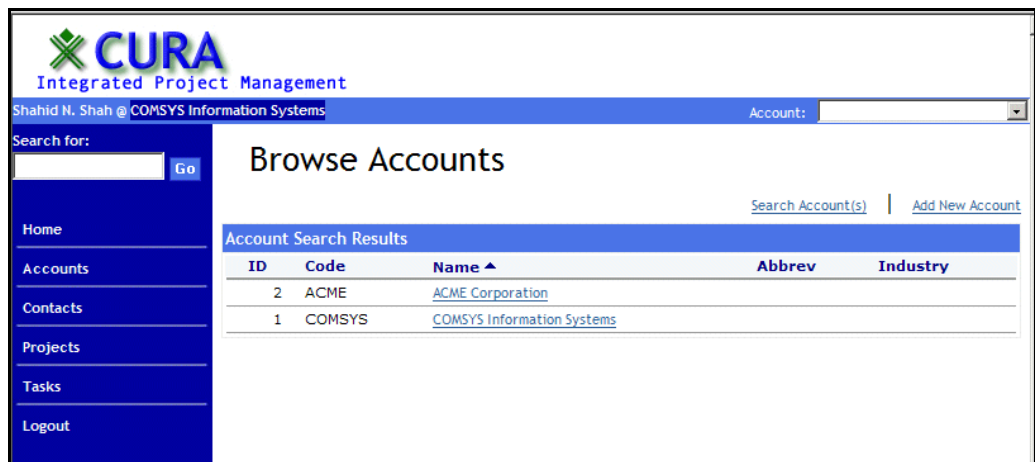
Figure 5: Browsing Accounts

♦ The "Accounts" link will take you to a list of your current clients along with giving you the opportunity to get extended information about them or add a new client if you choose.
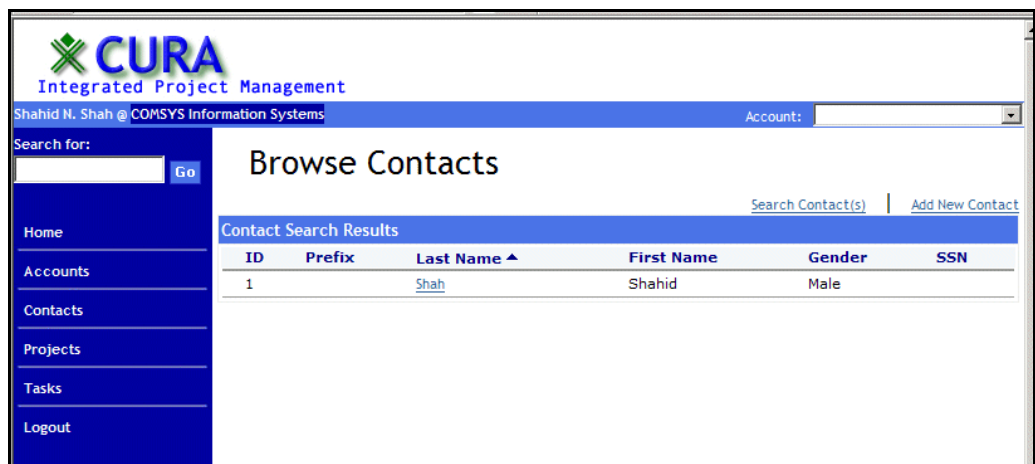
Figure 6: Browsing Contacts

♦ The "Contacts" menu will take you to a list of your contacts in the Cura database. Here you can add new contacts, see extended information (including client association) for a contact or edit existing contacts.
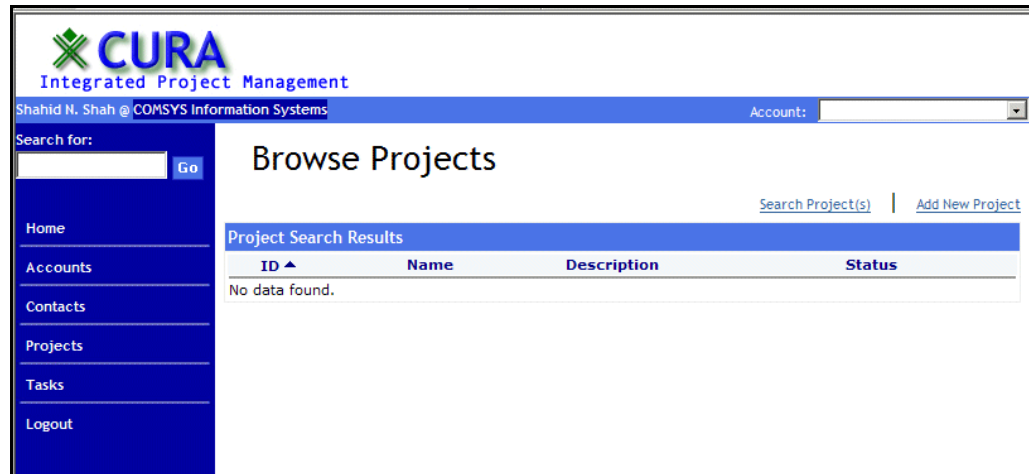
Figure 7: Browsing Projects

♦ The "Projects" link takes you to a summary list of all the projects you have stored in the Cura database. This list is similar to the list shown in the summary page when you login to Cura. Here you can view details about a project, edit a project's details, add or remove contacts from a project and add, remove or edit tasks associated with the project.
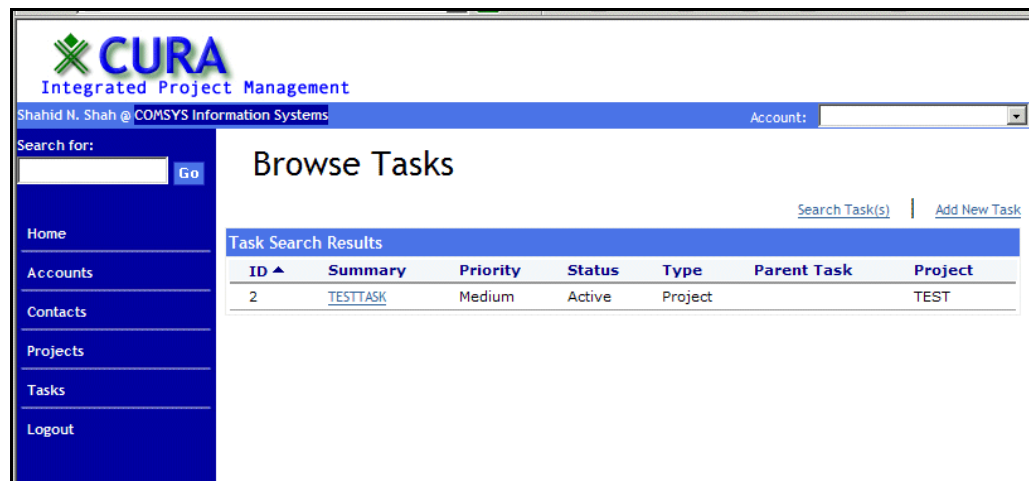
Figure 8: Browsing Tasks

♦ The "Tasks" link takes you to a summary list of tasks that are visible to you. This list is similar to the task list in the summary page shown when you first login to Cura. Here you can view extended details of a task, edit these details and add/remove members and subtasks to a task.

The "Logout" link logs you out of Cura and brings you back to the login page where you can log back in.

Cura's Database Design

As you can see, Cura is a complex application that manages a large amount of data. This data pertains not only to individual objects such as tasks and projects, but also to relationships between objects such as contacts, child tasks etc. The interface for all this data is clean and functional and has been discussed briefly above. As you saw while creating the Sparx Collection, creating such a user interface is straightforward. Sparx provides you with all the tools to create a quick template (such as the one used for the left sidebar menu and the top information bar) and dialogs. Using these tools, putting up a skeleton for Cura is an uncomplicated task.

The underlying database design is equally clean and effective. This design can be seen by opening up the XML definition of the entire database schema or by accessing Cura's ACE and selecting "Schema (XML)" from the Database menu. You will see a clean hierarchical Entity-Relationship diagram on the right of the page. The left side of the page will contain a list box with all the tables in the schema listed.

If you select any one of these tables, you will get a detailed view of that table with important information about each field in the selected table. A detailed description of all the fields shown in this view is provided in Volume 2 of the Sparx Documentation.

The overview of the schema and detailed view of any table is a functionality provided automatically by Sparx and needs no work on your part. It is a very good way to study the database schema used for Cura or any other Sparx application, especially one that you did not write.

Entities and Relationships

The few basic entities required for Cura are shown clearly bolded in the application's ACE[1]. These are the entities used to represent a person, a client organization, a project and a task. The other bolded entities shown in the application's ACE are auxiliary groupings of similar tables that are used throughout Cura.

As you can see, based on the application that we saw, the database schema is a very sensible mix of flexibility of entity relations and rigidity of structure. Each major entity is represented as a table containing nothing but references to child tables containing details about specific attributes of the parent entity. Similarly, relationships between major entities are handled by relationship tables that allow the flexibility of 1:n or n:n relationships between the two major entities involved. All limited value

---

[1] Choose Schema (XML) from the Database menu to see the ERD for Cura.

attributes are handled by 1:n relationships between major entities and lookup reference tables implemented as enumerations in the XML database schema.
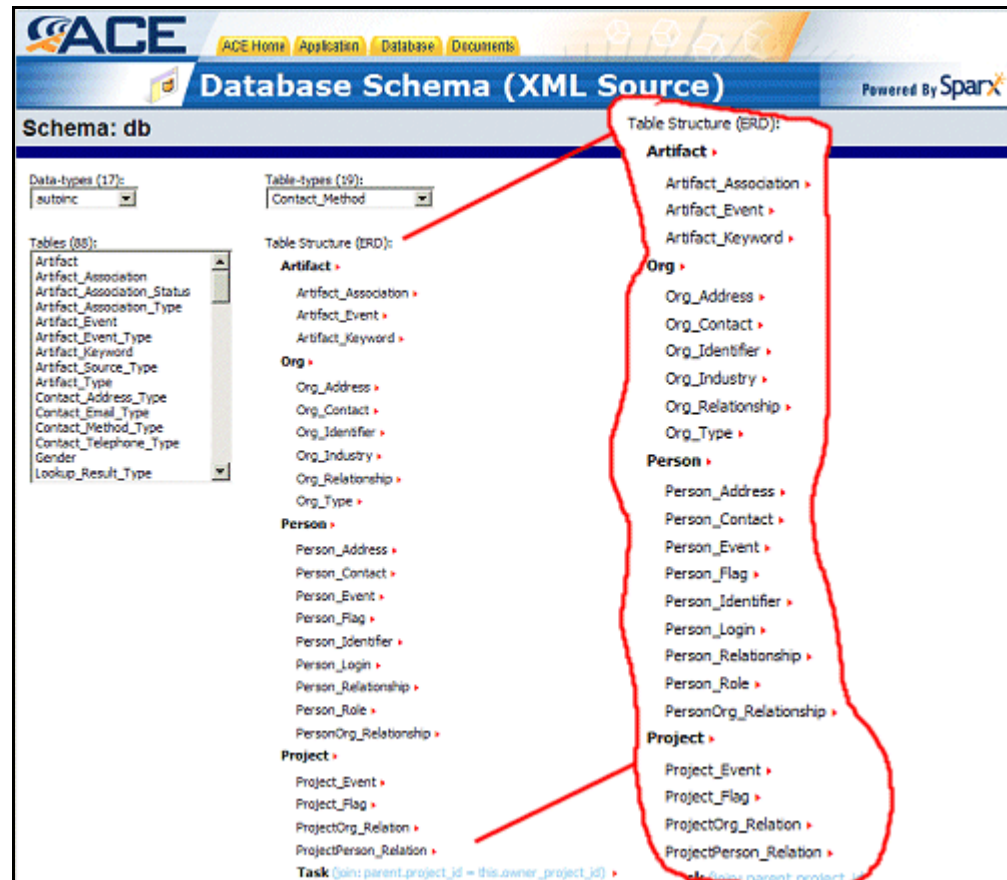
Figure 9: Viewing the database schema using ACE

If you look at the database schema, you will notice that while it is conceptually a very clean one, it can start looking complicated very fast. To make sense of all the tables involved in the Cura database schema, you can refer back to the Schema (XML) page in the application's ACE. This time, however, either click on one of the little red arrows to the right of a table's name or find and click on the table name in the list box on the left of the page.

Figure 10: Details of the Person table

This will take you to a detailed view of the table you selected. This is a completely auto-generated page that is created by Sparx by reading the XML schema stored in the Database directory of your application. As you can see, this view shows not only important details about every field in the selected table but also information about the relationships between this table and others. Specifically all tables that are children of the currently selected table are listed right at the top of the page. Such child tables are generally used to store attributes of the entity represented by the currently selected table.

You should also notice the icons to the extreme left of each field. These icons show whether this field is a foreign key, primary key, required field or a regular field. Clicking on a field name will show you additional details about the field. The most important of these details is the list of other tables which refer to this field. In essence, this is a list of all the tables that are related to the current table through the field you selected. Additionally, in the list of fields for the table, you will notice the References column which is filled for lookups. Clicking on a lookup will bring up that table so you can see exactly what values are being looked up. Finally, the Java Type column shows you what data type to use in your Java classes when reading the field from the database.

Relationships in the XML Schema

Relationships between tables are relatively easy to visualize using an application's ACE. In fact, when you examine each table in a relationship closely, you will notice that the documentation for different types of relationship is different: it is more reflective of the nature of the relationship between those tables. Similarly, different types of relationships between tables are represented differently in the Sparx DAL for

the application's database schema. Keeping in mind that the documentation and the DAL are generated automatically by Sparx, there must be an actual difference in the specification of the relationships in XML that results in the differences in their documentation and their implementation in DAL.

The four different types of relationships that can be defined between tables in the XML database schema are listed below with brief explanations of the purpose of each.

♦ **lookupref**. A lookup reference allows a field in a table to reference a field in another table. When translated to SQL it results in the creation of a foreign key relationship between the two tables. The general syntax follows the pattern `<column lookupref="theTable.theField"/>`

♦ **parentref**. A parentref reference allows the creation of a logical parent/child relationship between two tables. The target of such a reference is the field that will be used to create a 1:n foreign key relationship between the two tables involved. The general syntax of such a reference is `<column parentref="theTable.theField"/>`

♦ **selfref**. A selfref is a way to create a foreign key relationship within a table. The target of such a reference is the field that will be used to create a self-referential 1:n relationship in the table. This allows logical structures such as hierarchies. The general syntax of such a self-referential relationship is `<column selfref="theTable.theField"/>`

♦ **usetype**. This is a weak reference to the target field in the target table. Its sole purpose is to ensure that the source column always has a definition that is identical to the target column in the target table. The usetype relationship is mostly discarded in favor of creating a custom data-type that can then be used in all the concerned tables while eliminating such weak references between them. The general syntax for such a weak reference is `<column usetype="theTable.theField"/>`

You should know that when creating references between tables in the XML schema, it is not necessary to copy all the details of the referenced field into the referencing field. These details are automatically taken care of by Sparx. This is specially important for the most commonly used references since it speeds up the work of creating an XML database schema for an application and gets developers started on their work sooner.

Entity Inheritance

Finally the last important item to look for is the Inherits column in the table field list. This column's value demonstrates a very powerful feature of Sparx database schemas: inheritance. As an example, look at the field list for the tables `Person_Address`, which stores addresses for people, and `Org_Address`, which stores addresses for client organizations. You will notice that except for the first four fields (which are common to all tables) the rest of the fields all have the same names, SQL and Java data types. Further, all these fields (in both tables) have `Physical_Address` as the value of the Inherits column.

In this particular case, if you examine the **schema.xml** file for Cura, you will see that the `Physical_Address` table is a base table created to contain addresses of any entity.

The `Org_Address` and `Person_Address` tables, however, are derived from the `Physical_Address` table and therefore both contain the same fields as the `Physical_Address` table.

```xml
<tabletype name="Physical_Address" type="Default">
  <description>An address of a member of a certain entity, such as a
      person or org.</description>
  <column name="system_id" type="autoinc" primarykey="yes"
      descr="Unique identifier for foreign-key and update purposes"/>
  <column name="parent_id" parentref="$parentref$" indexed="yes"
      descr="The owner of the address"/>
  <column name="address_name" indexed="yes" type="text" size="128"
      descr="The address name"/>

  <column name="mailing" type="boolean" descr="True if this is the
      mailing address"/>

  <column name="line1" type="text" size="256" descr="The first
      address line"/>
  <column name="line2" type="text" size="256" descr="The second
      address line"/>
  <column name="city" type="text" size="128" descr="The city of the
      state"/>
  <column name="county" type="text" size="128" descr="The county of
      the state"/>
  <column name="state" type="text" size="128" descr="The state of the
      union"/>
  <column name="zip" type="text" size="128" descr="The postal code
      (+4 format)"/>
  <column name="country" type="text" size="128" descr="The country of
      the address"/>

  <index name="$tbl_abbrev$_unq" type="unique"
      columns="parent_id,address_name"/>
</tabletype>
```

```xml
<table name="Person_Address" abbrev="PerAddr"
      type="Physical_Address">
  <param name="parentref">Person.person_id</param>
</table>
```

```xml
<table name="Org_Address" abbrev="OrgAdr" type="Physical_Address">
  <param name="parentref">Org.org_id</param>
</table>
```

This inheritance behavior is shown in the source code above. The first bit of code shows the `Physical_Address` table being declared in the `table_types.xml` file. The second and third bits of code show the `Person_Address` and `Org_Address` tables being declared in the `person.xml` and `org.xml` files. Notice that the Person/Org Address tables needed to just specify that the tables were of type `Physical_Address` and follow that by the value of `parentref` needed by each table. Furthermore, changing the `Physical_Address` table and regenerating the schema will automatically result in modifying all tables that are derived from it.

This type of inheritance mechanism, provided by Sparx, allows the application data modeler to write less XML to define the schema while achieving the desired result. In addition, an application's ACE allows a data modeler to examine each table in detail so that he can verify that the desired results are indeed being achieved.

Enumerations and Lookups

When designing a database schema for an application, data modelers often have to balance the degree of normalization in the schema with the ease of use of the resulting schema from a developer's perspective. Thus, for example, when creating a model for a person's record and deciding on how to implement the `marital_status` field, you might settle for creating an integer field that stores `0` for `single` and `1` for `married`. This would require developers to be aware of this mapping and add to their work by forcing them to translate the numbers to the appropriate strings for display purposes.

Such a method has many limitations, the least of which is that any additions or changes to the code (i.e. a new value `2` that means `divorced` and another value `3` that stands for `separated`) would involve developers looking at all their code and adding interpretations for those new values.

An alternate method would be to store all the different marital status values in a table and create a one to many relationship between the `Marital_Status` and `Person` tables. Such a method would involve a different way of thinking on the part of developers who would now have one more relationship to worry about in their code.

Enter Sparx with its built-in enumerations and lookups. They are a cross between the two types of data models described above. They provide the ease of development that comes with the first code-based method and the flexibility of the second method all with minimal overhead. Developers are able to use enumerations as easily as they would the code-based method described earlier and would not need to worry about any additions or changes to the enumerated data since their code will automatically reflect the changes.

```
<table name="Marital_Status" abbrev="MarStat" type="Enumeration">
  <enum>Unknown</enum>
  <enum>Single</enum>
  <enum>Married</enum>
  <enum>Partner</enum>
  <enum>Legally Separated</enum>
  <enum>Divorced</enum>
  <enum>Widowed</enum>
  <enum>Not applicable</enum>
</table>
```

```
<table name="Person" abbrev="Per" type="Default">
  <description>Any person (client, staff member, etc.)</description>
  <column name="person_id" primarykey="yes" type="autoinc"
      descr="Auto-generated Unique ID for a person within the entire
      system"/>
  <column name="name_prefix" type="text" size="16"/>
  <column name="name_first" type="text" required="yes" size="32"/>
  <column name="name_middle" type="text" size="32"/>
  <column name="name_last" type="text" required="yes" size="32"/>
  <column name="name_suffix" type="text" size="16"/>
  <column name="short_name" type="text" required="dbms" size="42"
      descr="[first letter of name_first] [name_last]"/>
  <column name="simple_name" type="text" required="dbms" size="96"
      descr="[name_first] [name_last]"/>
```

```
    <column name="complete_name" type="text" required="dbms" size="128"
        descr="[name_prefix] [name_first] [name_middle] [name_last]
        [name_suffix]"/>
    <column name="short_sortable_name" type="text" required="dbms"
        size="42" descr="[name_last], [first letter of name_first]"/>
    <column name="complete_sortable_name" type="text" required="dbms"
        size="128" descr="[name_last], [name_first] [name_middle]"/>
    <column name="ssn" indexed="yes" type="text" size="11"
        descr="Social Security number"/>
    <column name="gender" lookupref="Gender" default="0"
        descr="Gender"/>
    <column name="marital_status" lookupref="Marital_Status"
        default="0" descr="Marital status"/>
    <column name="date_of_birth" type="date" descr="Date of Birth"/>
    <column name="age" type="integer" descr="Actual age (only required
        if date_of_birth is NULL or not known)"/>
</table>
```

To illustrate the ease of use for data modelers, you should look at the XML shown above. This is taken directly from the Cura's XML schema source and shows how the marital status of a person is modeled. Using the enumeration is almost intuitive and, if the data in the enumeration changes, it will automatically be reflected in the `Person` table.

```
create table Marital_Status
(
        id number(8),
        caption varchar(96),
        abbrev varchar(32)
);
create unique index MarStat_abbrev_unq on Marital_Status(abbrev);
alter table Marital_Status modify (id constraint MarStat_id_REQ NOT
        NULL);
alter table Marital_Status modify (caption constraint
        MarStat_caption_REQ NOT NULL);
alter table Marital_Status add (constraint Marital_Status_PK PRIMARY
        KEY (id));
create sequence Per_person_id_SEQ increment by 1 start with 1
        nomaxvalue nocache nocycle;
```

```
create table Person
(
        cr_stamp date DEFAULT sysdate,
        cr_person_id number(16),
        cr_org_id number(16),
        record_status_id number(8) DEFAULT 0,
        person_id number(16),
        name_prefix varchar(16),
        name_first varchar(32),
        name_middle varchar(32),
        name_last varchar(32),
        name_suffix varchar(16),
        short_name varchar(42),
        simple_name varchar(96),
        complete_name varchar(128),
        short_sortable_name varchar(42),
        complete_sortable_name varchar(128),
        ssn varchar(11),
        gender number(8) DEFAULT 0,
        marital_status number(8) DEFAULT 0,
        date_of_birth date,
        age number(8)
);
create index Per_ssn on Person(ssn);
alter table Person modify (person_id constraint Per_person_id_REQ NOT
        NULL);
```

```
alter table Person modify (name_first constraint Per_name_first_REQ
        NOT NULL);
alter table Person modify (name_last constraint Per_name_last_REQ NOT
        NULL);
alter table Person add (constraint Person_PK PRIMARY KEY
        (person_id));
```

```
alter table Person add (constraint Per_cr_person_id_FK FOREIGN KEY
        (cr_person_id) REFERENCES Person(person_id));
alter table Person add (constraint Per_cr_org_id_FK FOREIGN KEY
        (cr_org_id) REFERENCES Org(org_id));
alter table Person add (constraint Per_record_status_id_FK FOREIGN
        KEY (record_status_id) REFERENCES Record_Status(id));
alter table Person add (constraint Per_gender_FK FOREIGN KEY (gender)
        REFERENCES Gender(id));
alter table Person add (constraint Per_marital_status_FK FOREIGN KEY
        (marital_status) REFERENCES Marital_Status(id));
```

```
insert into Marital_Status(id, caption) values (0, 'Unknown');
insert into Marital_Status(id, caption) values (1, 'Single');
insert into Marital_Status(id, caption) values (2, 'Married');
insert into Marital_Status(id, caption) values (3, 'Partner');
insert into Marital_Status(id, caption) values (4, 'Legally
        Separated');
insert into Marital_Status(id, caption) values (5, 'Divorced');
insert into Marital_Status(id, caption) values (6, 'Widowed');
insert into Marital_Status(id, caption) values (7, 'Not applicable');
```

The three code segments shown above reflect all the SQL that is generated as a result of Sparx processing the XML database schema. Specifically, these are all the SQL generated for the `Person` table and the `Marital_Status` enumeration and their relationship. As you can see, the SQL is properly defined and the bolded part even shows the relationship being created between the two tables.

More Database Schemas

Whereas the database schema you saw in the previous section was geared towards Cura, it can be used as a pattern for future applications that require similar data relationships to those that exist between the major entities in Cura. This allows for a high degree of reusability for schemas between applications. Ease of reuse, in turn, means that any investment you make in designing a good database schema for an application will be able to pay dividends in not only the original application but also in all the applications that use it.

Cura's Presentation Design

The main elements of the Cura user interface are the page template and the data input and display elements used on the various pages of the application. Of these, a simple page template was covered in the Sparx Collection tutorial using JSP custom tags. However, since Cura adds login based security to the mix, the `app.page` template deserves a second look. Most of the dialogs are relatively trivial to understand using the information from the previous tutorials concerning the `dialogs.xml` file. The exceptions are where a dialog's source displays a particular Sparx usage pattern. Displaying fields in a table grid is also covered in previous tutorials using SQL statements on a JSP page. In addition to basic reports, however, this document illustrates a few Sparx usage patterns that are used in Cura and come in handy during the development of most applications.

The Page Template

As you saw in the Sparx Collection example, a page template is a JSP custom tag[2] that is used to start and end every JSP page in a Sparx application. This is not necessary but eases implementation of security and a consistent page design. The Java class that handles the JSP custom tag is called `app.tag.PageTag`. You have already seen a version of the `PageTag` class that implements a consistent page template in the Sparx Collection example. The process of implementing a page template and using it to create a page can be summarized as follows:

♦ Separate your page design into three sections: everything preceding the content, the content itself, and everything after the content.

♦ In your `PageTag` class, use the `doStartTag` method to output everything preceding the content and the `doEndTag` method to output everything after the content.

♦ When writing the JSP for a page, put the actual content of that page between the begin and end <app:page> tags.

The end result is that when your application server encounters the opening <app:page> tag, it executes the `doStartTag` method of the `PageTag` class. This causes everything preceding the content of the page to be output to the browser. Everything from the opening <app:page> tag to the corresponding ending tag will be the content and will be interpreted and output to the browser as necessary. When the application server encounters the ending <app:page> tag, it executes the `doEndTag` method of the `PageTag` class which outputs everything in the design that should come after the content to the browser.

---

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

The advantage of knowing these details of the inner workings of a custom JSP tag, specially the <app:page> tag, is that it should give you a better idea of the level in the application at which security is activated. When implemented, security should be added to the doStart method of your PageTag class. This enables your application to ensure that a user is permitted to view the currently requested page before showing him any part of it. Instead of the content of the page, it can directly display a dialog box, asking the user to verify his credentials by logging in to the application again.

```
if (doLogin(servletContext, (Servlet) pageContext.getPage(), req, resp))
  return SKIP_BODY;

if(! hasPermission())
{
  out.print(req.getAttribute(PAGE_SECURITY_MESSAGE_ATTRNAME));
  return SKIP_BODY;
}

String rootPath = req.getContextPath();
String resourcesUrl = rootPath + "/resources";

AuthenticatedUser user = (AuthenticatedUser) session.getAttribute
(com.netspective.sparx.xaf.security.LoginDialog.DEFAULT_ATTRNAME_USERINFO
);
Map personRegistration = (Map) user.getAttribute ("registration");
BigDecimal personId = (BigDecimal) personRegistration.get ("person_id");

// get the list of organizations the user belongs to
Map memberOrgs = (Map) user.getAttribute("member-orgs");
int orgCount = memberOrgs.size();
// get the current organization selected
String currOrgId = (String)session.getAttribute("organization");
String currOrgName = (String)memberOrgs.get(currOrgId);
```

The source code above is taken directly from (and edited slightly to remove non-essential lines) the PageTag class[3] that is a part of the Cura application. As you can see, these lines are distinct from the ones that are used to output the page template. These lines are also executed prior to any output to the web browser. All output is conditional upon the status returned by these security checks: if they return a failure, the user is asked to verify his credentials. Otherwise, the user is shown the page he originally requested. The screenshot below shows what happens when a user's session ends or he logs out and tries to use the Back button to get back into the site.

---

[3] The PageTag class has a fully qualified name of app.page.PageTag and is located in the classes/app/page directory under Cura's WEB-INF
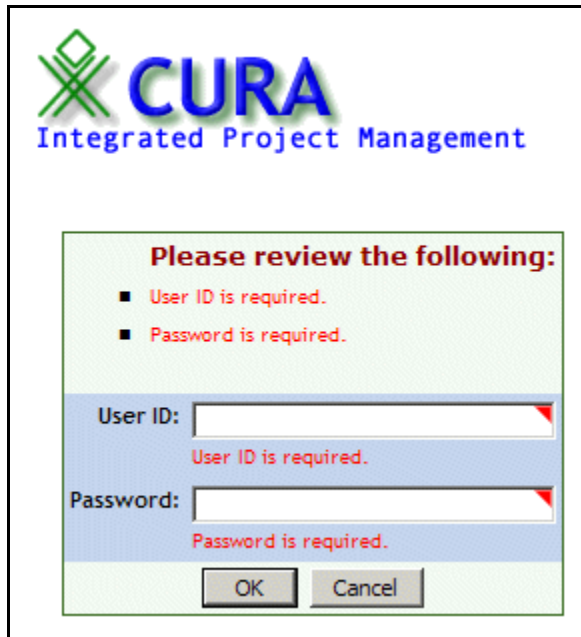
Figure 11: Results of a user's ending session, a user's logging out or a user using the Back button.

The two key statements in this code are the `if` statements at the top. If either of those statements' conditions is satisfied, the application server is instructed to skip the body of the tags and instead display appropriate error messages. The end result is that the content of the page (assuming it's an embedded Sparx dialog or query) is executed only when a user with valid credentials is accessing the page. This allows a developer to assume this when writing, say, the Java dialog handler class corresponding to a dialog embedded in a page secured in this manner.

Security and Permissions

If you want to take security for your application to the next level, you can institute permissions that are assigned to each user and are checked before each page is displayed. A user who does not have the permissions required to view a page will, under this new scheme, not be able to see that page.

It is easy to create a hierarchical role-based security model in Sparx merely by editing the `access-control.xml` file located in your application's `Site\WEB-INF\security` directory. The hierarchical nature of security allows you to have as fine grained security as is needed by your application without any additional coding.

```
<app:page title="Cura" heading="Welcome to Cura">
```

The line above is taken from the `index.jsp` file for Cura. The `app:page` tag, as used, allows anyone to view the contents of the `index.jsp` page. However, if the tag is modified to look like the one below, it will start limiting the types of users who can see the `index.jsp` page.

```
<app:page title="Cura" heading="Welcome to Cura" permission="normal-
user">
```

The line above will cause the `PageTag` class (which is responsible for handling the `app:page` tag) to verify that the user attempting to view the `index.jsp` page is permitted to view it. If the user does not have the `normal-user` permission, he will not be allowed to see the contents of `index.jsp`. Instead, he will get a message informing him that he is not allowed to see the page.

To give you an idea of how simple it is to add permissions to the access-control.xml file, the contents of the file (as it comes with Cura) are shown below. You should notice the hierarchy of permissions as well as the pseudo-roles created using the same permissions based system. A role, in this case, is basically a collection of permissions declared earlier. The `super-user` role, for example, is allowed any operations on orders while a `normal-user` is allowed everything but deleting orders.

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

<xaf>
  <access-control name="default">
  <permission name="app">
    <permission name="orders">
      <permission name="order_list"/>
      <permission name="create_order"/>
      <permission name="edit_order"/>
      <permission name="view_order"/>
      <permission name="delete_order"/>
    </permission>
  </permission>

  <permission name="role">
    <permission name="super-user">
      <grant permission="/app/orders"/>
    </permission>

    <permission name="normal-user">
      <grant permission="/app/orders"/>
      <revoke permission="/app/orders/delete_order"/>
    </permission>
  </permission>
</access-control>
</xaf>
```

As you can see, adding security to your applications is a relatively simple process. All you need to do is to ensure that your application's `PageTag` class contains a security check like the one shown above. In fact, the process is standard enough that it has been included in Volume 4 as a standard Sparx Usage Pattern. These Sparx Usage Patterns (SUPs) can be thought of as recipes containing generic instructions on how to accomplish something while developing your application. The SUP Guide (Volume 4 of the Sparx Documentation) contains many more such recipes that you can use in your own applications.

Security and Java

By this section, you should have a clear idea of how to secure pages against people who do not have the permissions to view them. This includes not only people who haven't logged into the application but also people who are just not allowed to see secure pages. There is one part of this mechanism that has not been covered yet: how to assign permissions to users who log into the application?

Assigning permissions to users is a deceptively simple operation in that it raises a whole new layer of security in front of your applications with just a couple of lines of Java. The code snippet shown below is taken from the source of the `AppLoginDialog` class located in the `Site/WEB-INF/classes/app/security` directory of your Cura home.

```
AuthenticatedUser user = new BasicAuthenticatedUser (dc.getValue
("user_id"), (String) personRegistration.get ("complete_name"));
user.setAttribute("person-id", personId);
dc.getSession().setAttribute("person_id", personId);
user.setAttribute("registration", personRegistration);
user.setAttribute("member-orgs", memberOrgs);
```

These lines are part of the `createUserData` method that returns an `AuthenticatedUser` object if a user was able to successfully login. You can add further code to this section that sets the roles that this authenticated user can have. Normally you might want to do that on a per-username basis but in this case the code shown below will be indiscriminately assigning the super-user role to everyone who is able to successfully log into the application.

```
ServletContext svc = dc.getServletContext ();
AccessControlList acl = AccessControlListFactory.getACL (svc);
user.setRoles (acl, { "super-user" });
```

These three lines first get the servlet context for the current application using the dialog context. Next, they get the list of ACLs defined in the `access-control.xml` file or the appropriate file as defined in your application's `configuration.xml` file. Finally, the last line assigns as many roles to the `AuthenticatedUser` object `user` as needed. In this case, just the `super-user` role is needed.

That is all there is to it! As you can see, working with security in Sparx is simple whether you use Java or XML. The key is finding out what level of security you need and then implementing it in a few lines of XML and/or Java.

Conditional Fields

Having seen how security is implemented in the page template, you can now get an inside look at another interesting SUP used in Cura: Conditional fields. These are fields that behave differently based on criteria set by the developer. These criteria can range from the value input into another field within a dialog, the mode of the dialog (add, edit or delete) or anything in between.

To see a first-hand example of a conditional field in use, you can open up a browser and log into Cura. Now use the left sidebar to go to the list of client Accounts. You should see a list in the content area to the right along with links to search for an account or to add a new account. Since the conditional field in the Accounts section is based on the mode of the dialog and not other criteria, it is important to see the appropriate dialog in multiple modes of action so the behavior of the field becomes apparent. To do this, follow the steps outlined below.

Figure 12: The ACME Corporation Account

♦   Click on an existing Account to view details about it. Then, click on the Edit link directly underneath the section titled Overview. This should take you to a dialog pre-populated with information about the account you selected. This dialog is in edit mode as shown by the title which reads "Edit Account"[4]. The fields present in this dialog, listed in order of their appearance, are: Account code, Name, Abbreviation, Ownership, Ticker symbol, Employees, and Timezone.

Figure 13: Editing The Account

---

[4] As further evidence of the dialog mode, open up the `edit.jsp` file in your `web-cura\account` directory and you will see that before the dialog is embedded in the JSP, the `data_cmd` attribute is set to edit, causing the dialog to go into edit mode. The alternate way, which you saw in the Sparx Collection, was to call edit.jsp with the URL parameters `data_cmd=edit`.

Figure 14: Adding an Account

♦ Now, to observe the difference, go back to the list of accounts and, this time, choose to add a new account. You will notice that in addition to all the fields shown in the edit mode of the dialog, there are three additional fields, namely: Type, Industry and Parent organization.

Looking at the XML source of the dialog, this can be clarified immediately. All three extra fields are declared the same way in the XML. As an example, the XML for the Type field is shown below.

```
<field.select name="org_type" caption="Type" choices="schema-
enum:Org_Type_Enum">
  <conditional action="apply-flag" flag="invisible" data-
cmd="edit,delete"/>
</field.select>
```

The only thing that separates this `field.select` is the addition of the `<conditional>` tag inside the `<field.select>` tag. This tag, which can be added to any other field type as well, tells Sparx to apply the invisibility flag on this field (in other words, make this field invisible) when the dialog is in edit or delete mode. You can verify this behavior by going back to Cura, selecting an account and clicking on delete. You should not see the Type, Industry or Parent organization fields listed in the resulting dialog.

Conditional fields are extremely helpful and intuitive to use directly in the XML declaration of a dialog. However, by making them a part of the XML, a developer can exert only coarse control over them. You can achieve much finer control by embedding this behavior in a custom Java dialog handler class written for the dialog. Such conditional behavior would go into the `makeStateChanges` method of your class, which would override the identically named method in the Sparx Dialog class.

Conditional Fields with Security

As an interesting tie-in to the previous section regarding security and permissions, if you have looked at the extended documentation[5] available for each Sparx field, you might have noticed the `has-permission` and `lack-permission` attributes for the `conditional`[6] tag. These attributes, in conjunction with the action attribute of a conditional, can be used to alter the behavior of a field when it is being viewed by someone with a particular set of permissions.

```
<field.select name="org_type" caption="Type" choices="schema-
enum:Org_Type_Enum">
  <conditional action="apply-flag" flag="invisible" has-
permission="normal-user"/>
</field.select>
```

The code above is exactly the same as the one shown in the previous section but with a different `conditional` tag. This version of the tag makes the `org_type` field invisible only when a user with `normal-user` permissions tries to view the dialog. Similarly, the `lack-permission` attribute can be used, for example, to show the `org_type` field only if the user attempting to view the dialog does not have the `super-user` permission.

Having learned how to use conditional fields in a Sparx dialog, you should also know that the syntax presented in the source code above is listed as a standard Sparx Usage Pattern in the SUP Guide. For other types of conditional fields that are not used in Cura, you should refer to the SUP Guide, which lists a few additional types of conditional fields.

Reports

The third important type of UI element used in Cura is the report. Reports are the tabular presentation of information used to display the summary for each section of Cura. They are generated directly from SQL statements which can be declared inline or in the `sql-statements.xml` file for your application. As simple a control as they might seem, do not be deceived by them. They are extremely flexible in how you can use them.

The SUP Guide contains an entry describing how to create a report that has, as an example, edit and delete links next to each row of the report. Such a report would allow the user to have one click access to the three major editing functions for every record displayed in the report: add, edit and delete. The edit and delete would exist for every row while the add functionality would be implemented as a link somewhere near the table.

---

[5] Extended developer documentation is available online at the following URL:
`http://developer.netspective.com`.

[6] Conditional fields are documented online at
`http://developer.netspective.com/xaf/dialogs/conditionals.html`
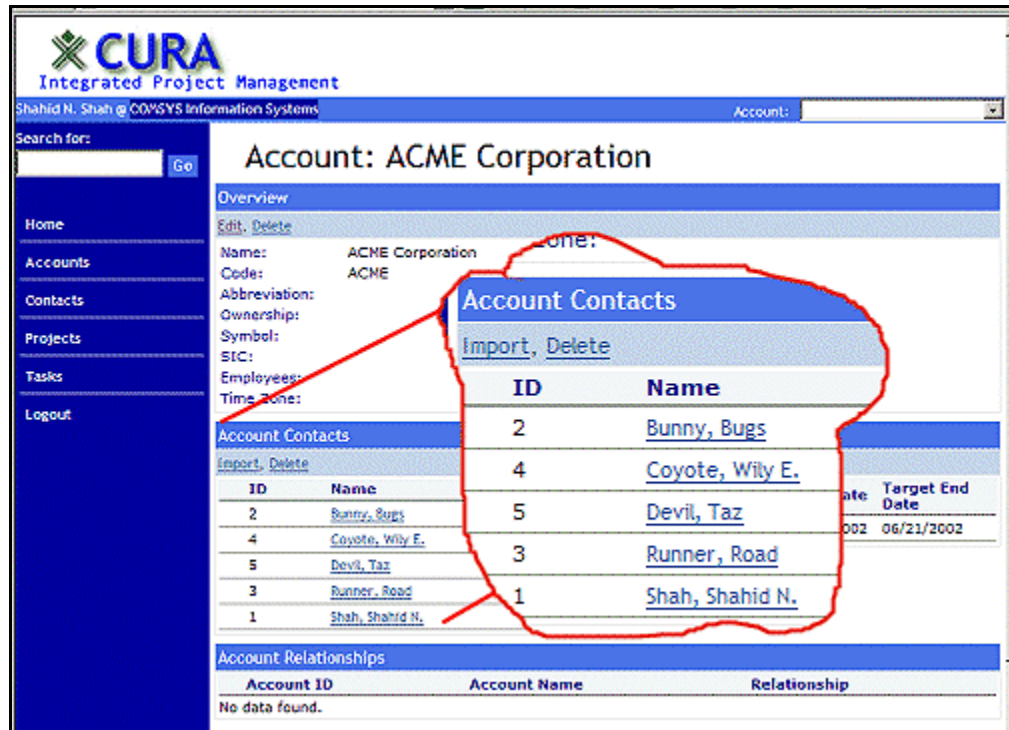
Figure 15: Account Contacts

The example below is taken from the `app-org-sql.xml` file located in the `WEB-INF\sql` directory of Cura. To see this report in action you should log into the Cura application using a web browser. Click on the Accounts link in the left sidebar menu to go to the list of accounts. Now click on any client account's name. In the detailed view of that account that appears, this report is towards the left side in the middle row. The report is titled "Account Contacts".

```
<statement name="contact-list">
  select
    person.person_id,
    person.complete_sortable_name,
    person.complete_name,
    org.org_name
  from
    personorg_relationship por, person, org
  where
    por.rel_org_id = ? and
    org.org_id = por.rel_org_id and
    por.parent_id = person.person_id
  order by
    person.complete_sortable_name

  <params>
    <param value="request:org_id"/>
  </params>

  <report heading="Account Contacts">
    <banner style="horizontal">
      <item caption="Import" url="config-
expr:add_member.jsp?org_id=${request:org_id}&amp;org_name=${request:org_n
ame}"/>
      <item caption="Delete" url="config-
expr:delete_member.jsp?org_id=${request:org_id}&amp;org_name=${request:or
g_name}"/>
    </banner>
```

```
    <column heading="ID" align="left"/>
    <column heading="Name" url="create-app-
url:/contact/home.jsp?person_id=${0}&amp;person_name=%{2}"/>
    <column display="no"/>
    <column heading="Organization" display="no"/>
  </report>

</statement>
```

The initial part of this SQL statement is the actual SQL, followed by details of the pluggable parameters it needs. The only parameter this statement requires is shown in the code snippet below as a question mark.

```
    por.rel_org_id = ? and
```

Details of how to find the value of the pluggable parameter are given in the `params` section of the statement. This section, shown below, says that the only parameter needed by the statement can be obtained using the value source `request:org_id`. The word to the left of the colon determines the type of value source that this is. The word to the right of the colon determines the name of the value (in the context of that type of value source) that the value source should fetch. In this case, the word `request` means that this is a value source that will fetch its data from URL encoded parameters passed into whichever page this query is embedded (directly or indirectly). Similarly, the `org_id` indicates that this value source will fetch the value of the variable named `org_id` from the list of URL encoded parameters that are passed into the page.

Keep in mind, however, that when calling a statement from Java at runtime, any parameter's value can be overridden. Thus the `org_id` parameter, which the statement expects as a URL encoded value, can be filled in automatically by the Java code that invoke the statement. One major advantage this capability provides developers is ease in debugging SQL statements. If all SQL statement parameters accept URL encoded values, you can use specially crafted URLs to test out the statement for debugging purposes. In a production environment, however, you can override these parameters directly in the Java.

```
  <params>
    <param value="request:org_id"/>
  </params>
```

Address | http://localhost:8080/cura/account/home.jsp?org_id=2&org_name=ACME+Corporation | Go
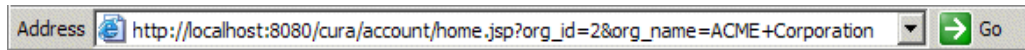
Figure 16: URL showing URL encoded parameters

Value sources

You have seen briefly what a value source is in this section as well as in previous tutorials. However, they are such an important part of the Sparx architecture that they deserve a good introduction here. A value source is a way to give input to a Sparx component. Instead of hard coding this input, developers have the ability to provide this input using a variety of methods. Examples of such methods include

direct user input through a dialog, session variables, URL encoded parameters and input created using a SQL query against a database.

The two main types of value sources are single value sources and list value sources. A single value source, by definition, can provide only a single value at a time. Therefore, when a developer embeds a single value source (such as `request:org_id`) into a Sparx component, he should be confident that at execution time, that value source will be replaced with one and only one value. List value sources provide multiple values at a time in the form of an array. Sparx components that expect multiple values are able to interpret the data provided by a list value source; other components are unable to do so. Therefore it is important to know what kind of value source to use in your code.

The power of the concept of value sources can be seen clearly when a developer wants to switch from, say, URL encoded parameters to session variables. The change is a simple matter of changing all occurrences of `request:` to `session:` and, of course, ensuring that these session variables are also updated in all the Java supporting classes. The actual process of obtaining the values, parsing them into an interpretable format and passing them to the Sparx components is opaque to the developer using them.

However, from a developer's perspective, the real power of value sources is revealed when you embed business logic into one. As an example, in a shopping cart that is used in a large online store, you might want to allow special discounts for customers based on the number of orders they have placed over the past year. You can create a custom value source called `discount-rate` that when asked to return a percentage discount rate for a customer can automatically look up that user's record and return a discount rate that is scaled to the customer's importance. Encapsulating all that business logic into a value source, you are free to develop the rest of your application always assuming you will get the right discount rate for each customer instead of always having to calculate the rate per customer yourself.

As with all Sparx components, while there is an XML version of value sources, there is also a corresponding Java version of each pre-defined value source. These Java versions can be used in all the Java dialog handlers and other supporting classes seamlessly. The other major advantage of the Java versions of the pre-defined value sources is that you can derive completely new value source classes for them. These custom value sources can then be registered in XML dialogs or SQL statements and used like a pre-defined value source.

While the pre-defined value sources that come with Sparx are responsible for just fetching data and passing it to your application, your custom value sources do not have to limit themselves in this way. They can manipulate this data before passing it on to your Sparx components. This allows for value sources with some business logic incorporated in them that ensures all data that is passed to your application conforms to business rules. All in all, value sources can easily be considered among the most powerful of the features of Sparx.

Below you will find the complete list of value sources that come built into Sparx. After looking over this list there should be little doubt in your mind that value sources can be one of the most important tools in your Sparx toolbox. Add to that the Java-based extensibility that is inherent in value sources and you have a fundamentally new way of thinking when writing your applications. For complete instructions on how to use the value sources, please visit please visit http://developer.netspective.com/xaf/value-sources.html.

NAME	DESCRIPTION
config	Provides access to configuration variables in the default configuration file (configuration.xml). If no source-name is provided the property-name requested is read from the default configuration element of the default configuration file. If a source-name is provided, then the property-name is read from the configuration named source-name in the default configuration file.
config-expr	Evaluates the given expression as a property value that would be created had the given expression been specified in the default configuration file (configuration.xml). If no source-name is provided the expr requested is evaluted based on the the default configuration element of the default configuration file. If a source-name is provided, then the expr is evaluated based on the configuration named source-name in the default configuration file.
create-app-url	Generates a URL based on the current application by automatically prepending the default servlet URL to the expression provided. There are four styles that can be used: **/** is used when you want to refer to the root path of your application The identifier **active-servlet** is used if you want to refer to the actively running servlet Any URL starting with a slash is considered a URL that is relative to the root path of your application Any URL not starting with a slash is considered a URL that is relative to the current servlet
create-data-cmd-heading	Returns the current dialog data_cmd identifier plus the text provided that would be suitable for use as the heading of a multi-purpose dialog (a dialog that can be used for adding, updating, and deleting). For example, if `Person` is the text, and the current dialog's data_cmd is `add` then this SVS would return `Add Person`.
custom-sql	This SVS is used in SQL DML tasks and custom tags when, instead of a java value or expression, you want the DML processing to take an actual SQL expression that should be evaluated in the database. For example, if `sysdate` is the sql-expr, then this SVS would return `sysdate` without evaluating in Java or treating it as a Java string. If this value source is used in any context other than a DML (SQL insert or update or delete) then it returns just the sql-expr itself.
dialog-field-types	Returns a list of all the dialog field types (<field.*>) that can be used in dialogs
dialogs	Returns a list of all the dialogs in the default dialogs file that match the Perl5 reg-ex provided.

NAME	DESCRIPTION
filesystem-entries	Provides list of files contained in a directory (either all files or by filter). If only a path is provided then this LVS returns a list of all the files in the given path. If a regular expression is provided (filter-reg-ex) then it must be a Perl5 regular expression that will be used to match the files that should be included in the list.
form	Provides access to a specific field of a dialog.
formOrRequest	Provides access to a specific field of a dialog. If the field-name refers to a dialog field whose value is null, then this value source will return the value of a request parameter named field-name.
formOrRequestAttr	Provides access to a specific field of a dialog. If the field-name refers to a dialog field whose value is null, then this value source will return the value of a request attribute named field-name.
generate-id	Returns a unique value each time the value source is called. The unique value is computed as a MD5 message digest hash based on the md5-seed provided with current date/time appended.
query	Executes a query and returns the results of the query as rows. If just a statement-name is provided, then the statement must exist in the default statement manager. If a statement manager file name is provided then the statement must exist in the provided file. Optionally, a list of bind parameters may be supplied as single value source specification of the form `vs:params`. Basically, any value source may be used in the bind parameters. If the query is being used to supply a select field with choices, then the first column is expected to be the caption to display and the second column, if any, will be used as the id column.
query-cols	Executes a query and returns the results of the query as columns. If just a statement-name is provided, then the statement must exist in the default statement manager. If a statement manager file name is provided then the statement must exist in the provided file. If the query is being used to supply a select field with choices, then the column name becomes the caption to display and the ResultSet's first row's column's value becomes the id column.
request	Provides access to HTTP servlet request parameters. All parameter values are returned as String objects.
request-attr	Provides access to ServletRequest attributes; intelligently handles object of types String, String[], List, and Map.
request-param	Provides access to HTTP servlet request parameters. All parameter values are returned as String objects.
session	Provides access to HTTP session attributes. Intelligently handles object of types String, String[], and List
static	Used to explicitly specify a value instead of obtaining the value from a parameter name. For example, if the `static-string` is `myfile.xml` then this string is literally taken as the value instead of searching for a parameter.
string	Used to explicitly specify a value instead of obtaining the value from a parameter name. For example, if the `static-string` is `myfile.xml` then this string is literally taken as the value instead of searching for a parameter.

NAME	DESCRIPTION
system-property	Provides access to the system property indicated by the specified key.

Report Structure

Having looked in detail at the SQL statements and, more importantly, value sources, it is time to look at exactly how a report is created from such a SQL statement. The source code shown below contains just the report section from the statement above. Keep in mind that for all SQL statements declared in the `WEB-INF\sql` directory of your application, a report section is purely optional. If you do not define it, Sparx will auto-generate a default report style that will be used if you embed such a SQL statement into a JSP page.

```
<report heading="Account Contacts">

   <banner style="horizontal">
     <item caption="Import" url="config-
expr:add_member.jsp?org_id=${request:org_id}&amp;org_name=${request:org_n
ame}"/>
     <item caption="Delete" url="config-
expr:delete_member.jsp?org_id=${request:org_id}&amp;org_name=${request:or
g_name}"/>
   </banner>

   <column heading="ID" align="left"/>
   <column heading="Name" url="create-app-
url:/contact/home.jsp?person_id=${0}&amp;person_name=%{2}"/>
   <column display="no"/>
   <column heading="Organization" display="no"/>

</report>
```

You should be able to see that this report consists of two main parts: a banner and the actual tabular report. Both of these parts of a report are dissected in the paragraphs below.

The banner is an area just above the column headings for the tabular report but below the report title. Here you can choose to add links to various table-wide actions, such as adding a new record.

```
    <banner style="horizontal">
```

This line starts the banner section. The value of the style parameter determines how items within the banner are aligned.

```
     <item caption="Import" url="config-
expr:add_member.jsp?org_id=${request:org_id}&amp;org_name=${request:org_n
ame}"/>
```

This is the first of two items that are part of the report banner. The item tag has two parameters: `caption` and `url`. Each item is rendered as a link where the link text is the `caption` and the link target is the `url`. In this particular case, the `url` is a value source as identified by its familiar syntax (`protocol:valueName`). However, this is a special value source in that the "value name" is in fact a string with two instances of request values embedded in it.

To start at the top, the `url` parameter is a value source of type `config-expr`. In its current instantiation, this type of value source will prefix its valueName with the full URL prefix for the application's home. This will result in the url becoming something along the lines of `http://localhost:8089/add_member.jsp` and so on. Further, the `${request:org_id}` is an instance of another value source, of a type different from the main value source. This one would, as discussed previously, fetch the value of the URL encoded parameter named `org_id` and embed that in it's location within the URL. Similarly, the `${request:org_name}` value source (located further in the URL) will be replaced by the value of the `org_name` parameter after the value source has extracted it from its URL encoded form. Assuming, then, that you clicked on the contact named Bugs Bunny, the URL corresponding to the Import caption will be similar to `http://localhost:8089/cura/account/add_member.jsp?org_id=2` and so on.

This facility of nested value sources, provided by Sparx, is another example of the sheer flexibility that value sources offer developers. Without these value sources you would be forced to use a full URL as opposed to a relative URL, not to mention the amount of code it would take to fetch and embed the values of `org_id` and `org_name` into the URL.

```
    <item caption="Delete" url="config-
expr:delete_member.jsp?org_id=${request:org_id}&amp;org_name=${request:or
g_name}"/>
```

This second banner item is very similar to the first one. It contains the same three value sources and should, therefore, be trivial to understand.

```
    </banner>
```

This line ends the banner section. The next section of the report is where each column of the tabular report is declared. If this section were omitted, Sparx is intelligent enough to auto-generate a default table.

The actual report table itself is defined using the following lines of code.

```
    <column heading="ID" align="left"/>
    <column heading="Name" url="create-app-
url:/contact/home.jsp?person_id=${0}&amp;person_name=%{2}"/>
    <column display="no"/>
    <column heading="Organization" display="no"/>
```

You have seen a report defined this way in the Sparx Tutorial so this is nothing new. However, for the sake of better understanding, these four lines will be briefly explained here.

Each column tag in the above four lines refers to a column representation of a field from the select statement declared earlier in the code. Knowing that, you can mentally see that the first three column tags represent the `person_id`, `complete_sortable_name` and `complete_name` fields of the person table. Further, you can see that the fourth column tag represents the `org_name` field in the `org` table.

Looking at the attributes for each column, the least interesting but most obviously noticeable in the final output is the `display` attribute which is used in the last two columns. In both these columns, its value is set to `no` and, as can be seen in the output of this report, those two columns do not appear in the output.

The most interesting attribute is the `url` attribute used in the second column. This column, titled Name, shows names of contacts in the output. However, these names are not plain text like the values in the ID column; instead, they are links which lead to detailed information about each contact. The url attribute in the second column tag is used to create the target URLs for each of the names shown in the output.

As you can see, the value of the `url` attribute is a value source, specifically a `create-app-url` value source. This type of value source pre-pends the base URL of the application to its target (in this case, the `/contact/home.jsp?` and so on). The two noteworthy items in this URL are:

♦ First, the part of the URL that says `person_id=${0}`. Here the `${0}` specifies that Sparx should replace this `${0}` with the value of the 0th column in each record. The 0th column in this report is the `person_id` column. Therefore, this would cause the SQL statement to look like `person_id=0001` if the value of the `person_id` field for a record were 0001.

♦ Second, the part of the url that says `person_name=%{2}`. Here the `%{2}` is a again a variable substitution similar to `${0}`. However, in this case not only does it substitute `%{2}` with the value in column 2 of the report for each record, but it also URL encodes that value before substituting it in.

Conditional Fields in Reports

You saw earlier how to create dialogs where some fields were conditionals: they reacted to the state of the application and the user who was viewing them. That same concept can be applied to reports with very good results.

As an example of such a situation, consider a payroll application which has administrator level users as well as normal users. Normal users are allowed to schedule pay checks or bonus checks for employees in the system but can not see the amount of any such check. It takes an administrator level user to add or delete users and employees from the system. In addition, only an administrator level user can view or modify the salary for employees to, say, reflect a promotion.

Consider, further, that as soon as a user logs into the application, the first screen he sees is a report of all the checks scheduled to go out today with details of the recipient employee and the amount of each check. In such a case, a conditional report would be very useful in hiding the amount of each check from everyone but administrator level users.

```
<report heading="Account Contacts">
  <column heading="Employee ID" align="right"/>
  <column heading="Name" align="left"/>
  <column heading="Check Number"/>
  <column heading="Amount">
    <conditional action="apply-flag" flag="invisible" lack-
permission="administrator"/>
  </column>
```

```
</report>
```

The XML snippet above is a sample report that could be used in the hypothetical application described above. Notice that the last column has a `conditional` tag nested inside it that tells Sparx to make it invisible if the user who is attempting to view it lacks the `administrator` permissions. With this conditional in place, the same report can be used for both normal users as well as administrators of the application. With one less report to write, a developer can concentrate his efforts on other parts of the application.

You should now be able to see how a simple SQL statement, augmented with a report definition, can create a complex looking table complete with links to allow record editing. The method of achieving this end result is relatively uncomplicated. However the technology behind it is very complex and powerful.

Query Definitions – Reports on Fire

While SQL statements are static statements that return a set of results based on pre-determined criteria, they are not the most powerful database query tool provided by Sparx. That title rests with Query Definitions.

Query definitions are frameworks that define the relationships between a set of tables, based on which developers can allow users to create and use any number of dynamically generated SQL queries for data. In practical terms, this implies that once a developer has created a query definition, users of the application can enter their own criteria to search the tables defined in the query definition for data.

Query definitions have been covered at some length in Volume I of the Sparx documentation but deserve a brief overview here. There are three main parts of a query definition.

♦ Definitions of all the fields accessible by the user. In this section you are responsible for defining all the fields that end users can use to create dynamic queries using the query definition.

♦ Definition of table relationships. In this section you should declare all the tables that this query definition encompasses (directly or indirectly) as well as the relationships between all of them. This section is extremely important as Sparx will use the rules defined here to create appropriate relationships between tables when it generates a dynamic SQL query.

♦ Definitions of UI. In this section you need to create at least one user interface for end users. This is the UI that will be presented to them when they need to create a dynamic query.

The code that follows is the XML definition of a query definition used in Cura. For a detailed explanation of how to write query definitions, which would also explain this query definition considerably, please refer to Volume 2 of the Sparx documentation.

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

<xaf>
  <query-defn id="Person">
```

```xml
    <!-- Fields -->
    <field id="person_id" caption="Contact ID" join="person"
column="person_id">
      <report heading="ID"  align="left"/>
    </field>
    <field id="name_prefix" caption="Prefix" join="person"
column="name_prefix">
      <report heading="Prefix" />
    </field>
    <field id="gender" caption="Gender" join="person" column="gender">
      <report heading="Gender"/>
    </field>
        <field id="gender_name" caption="Gender Type" join="gender"
column="caption">
            <report heading="Gender"/>
        </field>
    <field id="name_last" caption="Last Name" join="person"
column="name_last" where-expr="upper(person.name_last)">
      <report heading="Last Name" url="create-app-
url:/contact/home.jsp?person_id=${0}&amp;person_name=%{4}"/>
    </field>
    <field id="name_first" caption="First Name" join="person"
column="name_first" where-expr="upper(person.name_first)">
      <report heading="First Name"/>
    </field>
    <field id="complete_name" caption="Full Name" join="person"
column="complete_name" where-expr="upper(person.complete_name)">
            <report heading="Full Name"  display="no" />
    </field>
    <field id="ssn" caption="SSN" join="person" column="ssn" column-
expr="person.ssn">
      <report heading="SSN"/>
    </field>
        <field id="dob" caption="Date of Birth" join="person"
column="dob" where-expr="to_char(dob, 'MM/DD/YYYY')">
            <report heading="Date of Birth"/>
        </field>
        <field id="marital_status" caption="Marital Status" join="person"
column="marital_status" >
            <report heading="Marital Status"/>
        </field>
        <field id="org_id" caption="Organization ID" join="org"
column="org_id" where-expr="org.org_id">
            <report heading="Organization ID"/>
        </field>
    <field id="org_name" caption="Organization Name" join="org"
column="org_name" where-expr="upper(org.org_name)">
      <report heading="Organization Name"/>
    </field>

    <!-- Joins -->
    <join id="person" table="person" auto-include="yes"/>
    <join id="org" table="org"  condition="org.org_id =
personorg_relationship.rel_org_id" imply-join="personorg_relationship"/>
        <join id="personorg_relationship" table="personorg_relationship"
condition="personorg_relationship.parent_id = person.person_id"/>
        <join id="gender" table="gender" condition="gender.id =
person.gender" />

    <select-dialog name="person_search" allow-debug="yes" show-output-
dests="yes" hide-readonly-hints="yes" heading="Search Contacts">
      <field.text query-field="person_id"/>
      <field.text query-field="name_last"/>
      <field.text query-field="name_first"/>
      <field.select query-field="gender" choices="query:person.gender-
list" prepend-blank="yes"/>
      <field.select query-field="org_id" choices="query:person.active-
org-memberships-name-id"  default="session:organization" prepend-
blank="yes"/>
            <field.select name="sort_order"
          caption="Sort By"
```

```
            style="combo"
            choices="Last Name=name_last;Gender=gender;SSN=ssn"
                  default="name_last"/>

    <select heading="Contact Search Results">
      <display field="person_id"/>
            <display field="name_prefix"/>
            <display field="name_last"/>
            <display field="name_first"/>
            <display field="complete_name"/>
      <display field="gender_name"/>
      <display field="ssn"/>

      <order-by field="form:sort_order"/>

      <condition field="person_id" allow-null="no" comparison="starts-
with" value="form:person_id" connector="and" />
      <condition field="name_last" allow-null="no" comparison="starts-
with" value="form:name_last" connector="and" bind-expr="upper(?)"/>
      <condition field="name_first" allow-null="no" comparison="starts-
with" value="form:name_first" connector="and" bind-expr="upper(?)"/>
      <condition field="gender" allow-null="no" comparison="equals"
value="form:gender" connector="and"/>
            <condition field="ssn" allow-null="no"
comparison="equals" value="form:ssn" connector="and"/>
            <condition field="org_id" allow-null="no"
comparison="equals" value="form:org_id" connector="and" />
    </select>
    <director cancel-url="index.jsp" submit-caption="Search"/>
  </select-dialog>
  </query-defn>
</xaf>
```

You can get a good idea of what each part of this query definition is by observing it using Cura's ACE. Get to the main list of query definitions by choosing SQL Query Definitions from the Database menu in Cura's ACE.

Figure 17: Query Definitions

This screenshot shows the list of all the query definitions that are a part of Cura. You can test out any of these query definitions by clicking on their names and in the resulting screen clicking on the Run icon for their user interfaces.
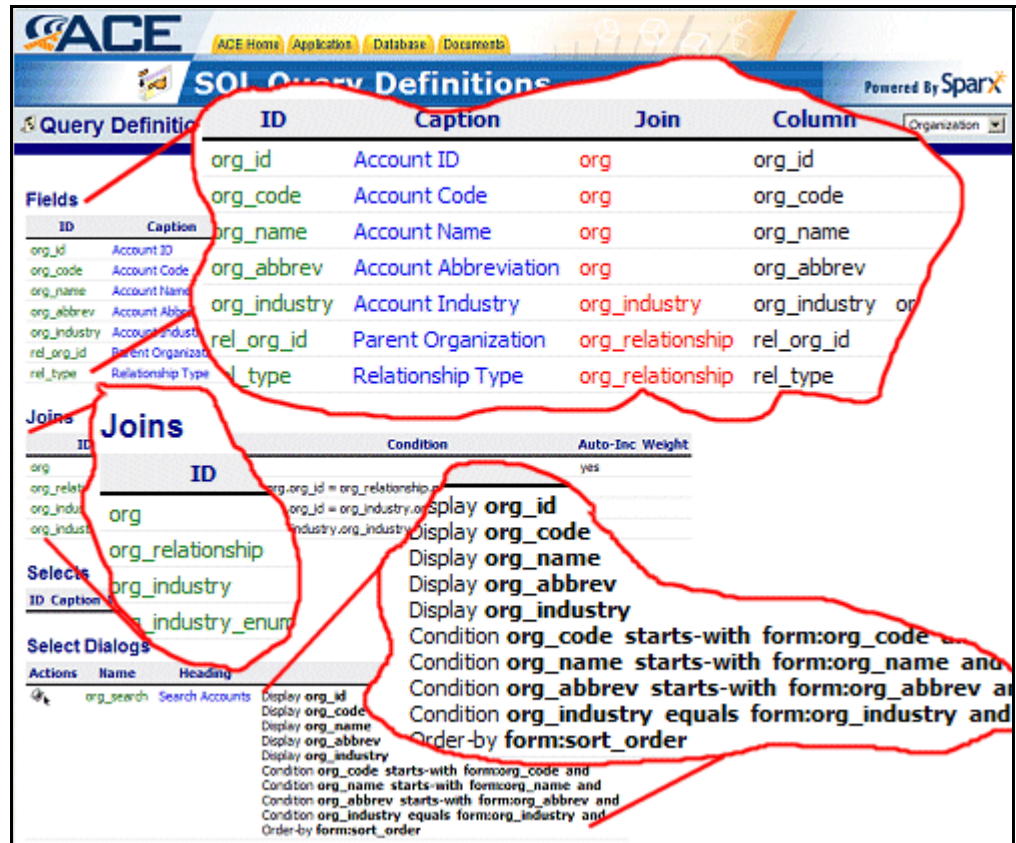
Figure 18: Details Of the Organization Query Definition in ACE

This screenshot shows the details of the `Organization` query definition. You can clearly see the three major sections in the query definition: the fields, the table relationships (referred to as joins) and the user interfaces (referred to as select dialogs). Click on the icon in the left-most column of the `org_search` select dialog to try out this query definition.
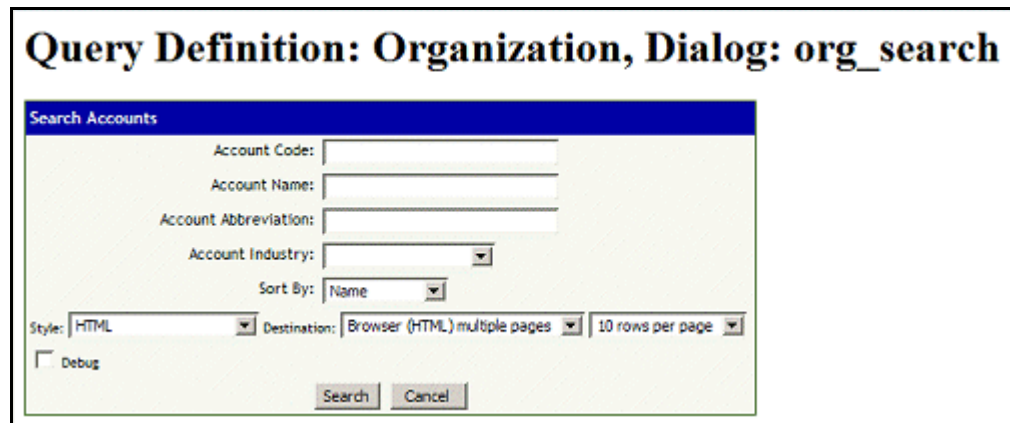
Figure 19: The org_search select dialog

This screenshot shows the `org_search` select dialog. As you can see, it is meant to allow users to search for an organization using a variety of criteria. The criteria are

defined in the fields section of the query definition. If you just press the `Search` button while all the fields are empty, you should get back a report containing all the organizations stored in the `org` table. An example of such a report is shown below.

Figure 20: The org_search Results

As you can see, this report looks like a Sparx-generated report from data gathered from any static SQL query. In fact, as seen in the screenshot, you can create reports much the same way as you would from static SQL albeit with knowledge of the fact that all your queries will be automatically generated.

This marks the end of your journey into the highlights of Cura's UI. The rest of the UI should be self-explanatory, assuming the knowledge gained in Volume 1 and 2 of the Sparx tutorials. It is a very good example of how a complex application can be built using simple components grouped together in a meaningful whole. Cura's interface is compact, informative and accessible and all it takes are a few well chosen dialogs and reports sewn together with a good page template.

Cura's Data Layer

The data layer functionality built into Sparx comprises most (if not all) of the range of the data manipulation functions needed by the average database driven application. There are two main areas where the data layer plays a part. The first is in communicating information from the UI of the application to the data processing modules. The vehicle responsible for this communication is the Dialog context. The second area, where the data layer plays a huge role, is the communication of information between the data processing modules and the database. This communication is handled by the use of the Sparx DAL. However, as might already be apparent, the most important players in the data layer are the actual data processing modules. These are usually implemented as parts of the Java dialog handler classes and are responsible for processing all data incoming from the UI and outgoing from the application. These data processing modules demonstrate not only complex data validation but also compile-time validation of data access code.

All three important parts of a Sparx application's data layer will be explained in detail in the sections to follow, starting with dialog contexts.

Dialog Contexts

Dialog context classes are, in essence, Java beans that are used to access data that has been submitted using a Sparx dialog. There are two types of dialog contexts. The first is a generic default dialog context that is built into Sparx. This dialog context can be used with any dialog and has a generic set of procedures to get and set data in that dialog. The other is a custom dialog context. This can either be a hand-coded Java class that is derived from the Sparx `DialogContext` class or an auto-generated class that is created when you build a Sparx application.

A custom dialog context class will usually be designed solely for use with a particular dialog and none others. Since a dialog context is a bean, it will consist mainly of get and set methods that allow a developer to access data stored in that dialog context or alter data stored there. For a custom dialog context, there will be specific get and set methods for each field in the dialog to allow explicit data interaction.

Examples of how generic and custom dialog contexts are used in real-world applications follow in the sections below. They are intentionally bundled with the examples for Sparx's different data access styles since dialog contexts are used most often in conjunction with data access.

Java Dialog Handlers

A skeleton of a Java dialog handler class is shown below. All the important methods that a Java dialog handler override in the default dialog handler are shown. You can see how this skeleton is used in Cura's dialogs by examining the Java source for all the dialog handlers it comes with.

```java
public class DialogHandler extends Dialog
{
  public void populateValues(DialogContext dc, int i) {
    super.populateValues(dc, i);
  }

  public void makeStateChanges(DialogContext dc, int stage) {
    super.makeStateChanges(dc, stage);
  }

  public boolean isValid(DialogContext dc) {
    return super.isValid(dc);
  }

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

The four methods shown in this skeleton class have been discussed briefly before. Being as important to a Sparx application as they are, however, they deserve some elaboration here.

The populateValues method

The `populateValues` method, as its name suggests, is used to populate the controls (such as text boxes, list boxes, etc) of a dialog with values prior to rendering it on a client browser. In this skeleton class, this method is overriding the method of the same name present in its parent Dialog class. This parent class is also the default dialog handler for all XML dialogs that do not have corresponding Java dialog handlers.

By default, the `populateValues` method ensures that all fields in a dialog are empty. You can, however, override this method in your own class and, based on your own criteria, determine what values you want to fill each field with. These values can be the fields from a row of a database table, the key to which you can pass to the dialog as URL encoded parameters. Alternatively, you could dynamically scan a directory for all new files and populate a pull-down list box with these filenames. In essence, then, how you use the `populateValues` method is up to you. Remember, however, that you wield a lot of power with it.

In the case of the example above, the sole contents of the populate Values method is a call to the `populateValues` method in its parent class. This is to ensure that the default dialog population logic has a chance to execute. All code present in this method can and will modify the results of that default behavior. If you find that the default behavior is all you need, you can simply choose not to define the `populateValues` method at all. Alternatively, you can leave the `populateValues` method as it is and only the default behavior will execute.

```
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()) {
    String orgId = dc.getRequest().getParameter("org_id");
    dc.populateValuesFromStatement("org.registration", new Object[] {new
Long(orgId)});
  }

  if (dc.deletingData()) {
    String orgId = dc.getRequest().getParameter("org_id");
    dc.populateValuesFromStatement("org.registration", new Object[] {new
Long(orgId)});
  }
}
```

The method shown above is taken directly from the `OrgDialog.java` file in Cura. As you can see, this method has a very simple way of populating an entire dialog.

The makeStateChanges method

The `makeStateChanges` method is another one with a name that gives away its purpose. This method is called right after `populateValues` but right before the dialog is rendered. Its purpose is to make any changes of state that are necessary to the dialog and its fields before rendering it. As you can see, this method also ensures that the default behavior takes place by calling the `makeStateChanges` method for its parent class.

The default behavior of the `makeStateChanges` method can be noticed in almost every dialog when it is in delete mode: all the fields become read only. For the other modes of execution that a dialog has, the `makeStateChanges` method does not touch the field states. Examples of alternate behavior that an overridden `makeStateChanges` method makes possible include hiding a field when a particular record is being displayed in the dialog or showing the value of an internal variable when an administrator-level user is using a dialog. As with the `populateValues` method earlier, the possibilities are limited only to your imagination and the requirements of the application.

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

The method shown above is taken directly from the `OrgDialog.java` file in Cura. As you can see, this method is nothing but a call to the `makeStateChanges` method in the parent `Dialog` class. This ensures that the `makeStateChanges` method will carry on the default behavior of a dialog.

A more elaborate example, however, might be of a dialog that contains a field that is visible only during business hours (0900 to 1700 on weekdays only). If that dialog is executed at any other time, that field would not be rendered.

```java
public void makeStateChanges (DialogContext dc, int stage) {
  super.makeStateChanges (dc, stage);

  Calendar rightNow = Calendar.getInstance ();
  int theHour = rightNow.get (Calendar.HOUR_OF_DAY);

  if ((theHour >= 9) || (theHour <= 17)) {
    // Dont let people misuse this program during business hours
    TextField _personName = (TextField) dc.getField ("personName");
    _personName.setFlag (DialogField.FLDFLAG_INVISIBLE);
  }
}
```

The code snippet above is one way of hiding a field during work hours. In this case, it is a field named personName that is being made invisible. Of course, making a field invisible is not the only state change that is possible with Sparx. You should refer to the complete Sparx XAF API (JavaDoc) to see what other field flags are available in the DialogField class.

The isValid method

The isValid method is probably one of the most important methods in a dialog handler class. Its function is to validate all data that is input using a dialog. It should be easy to see that this method gets executed immediately after the dialog is submitted. If, for whatever reason, this method returns a failure code, the dialog is re-populated with the data input by the user (courtesy of the default behavior of the default populateValues method) and re-rendered with the offending fields or data types marked. This loop of validating the data and re-rendering the dialog upon failure continues until the user enters data that can be validated successfully.

The default behavior of the isValid method is to call individually the validation methods of each field that comprises the dialog. By default, therefore, the isValid method of a dialog handler class will only verify that each field passes its own sanity checks (such as a date field holding an impossible date) and leave it at that. In most applications, however, a dialog's data will need to be validated not only against a database but also against the rest of the data input in the same dialog. This means that for the majority of cases, the isValid method of a dialog handler class will need to be overridden to allow the addition of this validation logic.

A classic example of validation logic that is required by most applications but is not handled by the default isValid method is that of choosing a username for registering at an online forum. The application needs to make certain that no two patrons have the same username before allowing a new patron's record to be created. The isValid method for the dialog that handles registration will then have to, among other things, access the database and verify that the username entered by the new patron is not already in use. If the application finds that the username is in use, the isValid method of the dialog can mark the username field as an error and give an informative message to the new patron, allowing him to fix the error and try again.

```
public boolean isValid(DialogContext dc) {
  return super.isValid(dc);
}
```

The method shown above is taken directly from the OrgDialog.java file in Cura. As you can see, this method is nothing but a call to the isValid method in the parent Dialog class. This ensures that the isValid method will carry on the default behavior of a dialog which is to validate each field individually and, if every field validates successfully, to indicate a successful validation of the whole dialog.

```
public boolean isValid(DialogContext dc) {
  boolean status = true;

  if (!super.isValid(dc)) {
    dc.addErrorMessage ("There was an error validating this dialog!");
    status = false;
  }

  return status;
}
```

The version of the isValid method shown above is an improvement over the default isValid that is used in the OrgDialog.java file in Cura. This version first checks to see whether the default isValid method returns a true or false. In case it returns a false, it proceeds to add a dialog level error message (using the line shown below) that shows up at the top of the dialog to let the user know that there was an error validating his input. Further, it returns the false status back to the dialog so it knows something went wrong.

```
    dc.addErrorMessage ("There was an error validating this dialog!");
```

A more elaborate example, however, that will also provide insight into the scope of the dialog-level isValid method, is when you want to ensure that a user enters two unique numbers into a dialog. Now you are faced not only with the task of ensuring that you collect two *numbers* (field-level validation) but also that they should not be the same (dialog-level validation). While a default isValid method would be fine for the first part, you need to add instructions to compare the two numbers for the second part.

```
public boolean isValid(DialogContext dc) {
  boolean status = super.isValid(dc);

  if (status) {
    if (parseInt(dc.getValue("number1")) !=
parseInt(dc.getValue("number2"))) {
      dc.addErrorMessage ("number2", "Please change the value of number2
so it is different from the value of number1");
      status = false;
    }
  }

  return status;
}
```

As you can see, while this method called the default isValid method to ensure that each field is validated individually, it did not stop there. The rest of the method contains logic to ensure that the two fields entered do not match. If they do, the second field (named number2) will have an error message attached to it that lets the user know to change the value of this field to comply with the dialog's validation

rules. Further, the dialog will be rendered invalid even though on a per-field basis, it is perfectly valid.

The execute method

The **execute** method is the core of a dialog handler class. This method is executed after the input data has been successfully validated. It is the code written in this method that is responsible for determining what the input is from the dialog and what is to be done with it. Since the data is already validated prior to being exposed to the **execute** method, developers have the benefit of knowing that all the data they will get will be valid. This way, developers do not need to worry about malformed data or illegal input. Instead they can focus on coding the processing needed for the data after it has been input. Once this input has been processed, the code in the **execute** method usually outputs a status or a message to inform the user of the result. After submitting a dialog with input, a user will not see the dialog again. Instead he will see, in its place, the output from the **execute** method. You can observe this behavior in Volumes 1 and 2.

If a dialog handler class does not contain an **execute** method, the dialog's execution will be handled by the default **execute** method. Depending, then, on how you have implemented the dialog, the default **execute** method can do one of three things.

The first case is when you have used the **<execute-task>** tag in your XML dialog. The **<execute-task>** tag allows a developer to use pure XML to define all the processing that needs to be done when a dialog is executed. The type of processing is, of course, not very complex. However, a few well chosen lines of XML can easily replace an entire custom Java dialog handler class. Ultimately, however, in the absence of a custom dialog handler class, it is the **execute** method of the default dialog handler class that interprets the XML task definitions inside an **<execute-task>** tag and executes them. Therefore, if a custom Java dialog handler class has no **execute** method, the default **execute** method will take over and, assuming there are tasks defined in the XML dialog, will execute all the tasks defined in the dialog.

The second case is when the XML definition of the dialog does not have any special tags that specify what tasks to execute. In this case, the default execute method behaves just like in the ACE: It dumps a list of all the values input by the dialog and re-renders the dialog at the end of this dump, waiting for further user input.

The third and final case is when a Sparx dialog is instantiated *and* defined in a JSP file as shown in the JSP code snippet above. In this case, everything between the beginning and ending **<xaf:dialog>** tags is considered to be the execute method of that dialog. Further, such a dialog inherits the **populateValues**, **isValid** and **makeStateChanges** methods from the default Java dialog handler.

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

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

```
<xaf:dialog name="test.hello_first">
  Hi there <%= dialogContext.getValue("personName") %>
</xaf:dialog>
```

The two code snippets shown above demonstrate exactly how a dialog can have its **execute** method defined completely in JSP. The first code snippet shows an XML definition of a dialog which, in this case, only contains the one field named **personName**. The second snippet shows the contents of a JSP file that uses the default **dialogContext** object provided within the **xaf:dialog** tags to fetch the value of the **personName** field and display it on the user's browser.

As you can clearly see, the **execute** method of a dialog handler class is the central element in the class, around which everything revolves. It is here that a developer would insert validated data into a database or update a table in the database based on input from the dialog. It is also here that a developer can make changes to a filesystem or instruct a process to stop running based on the input from the dialog. In short, the code written in this method can do anything dictated by the input data.

Examples of the typical usage of the **execute** method are shown later in this document. Most of these examples use the **execute** method as a dispatching routine which calls an appropriate (non-inherited) method of the dialog to handle each separate dialog mode. This pattern will become apparent later in this document when you see the **execute** methods of various dialogs as they process dialog data.

```java
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 url = "";
  HttpServletRequest request = (HttpServletRequest)dc.getRequest();
  try {
    if (dc.addingData()) {
      // dialog is in the add data command mode
      this.processAddData(dc);
      url = request.getContextPath() + "/account/home.jsp?org_id=" +
request.getAttribute("org_id") +
          "&org_name=" +
URLEncoder.encode((String)request.getAttribute("org_name"));
    }

    if (dc.editingData()) {
      // dialog is in the edit data command mode
      this.processEditData(dc);
      url = request.getContextPath() + "/account/home.jsp?org_id=" +
request.getAttribute("org_id") +
          "&org_name=" +
URLEncoder.encode((String)request.getAttribute("org_name"));
    }

    if (dc.deletingData()) {
      // dialog is in the edit data command mode
      this.processDeleteData(dc);
    }
  } catch (Exception e) {
    e.printStackTrace();
  }

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

```
      e.printStackTrace();
   }
}
```

Data Access in Cura

Having seen the full power of a Java dialog handler and how it lets you implement the data processing part of the data layer of your application, you are now ready to learn about how Sparx makes actual data access as easy and intuitive as the UI of an application. Sparx provides developers with extremely easy to use programmatic tools to manipulate date at the data layer of an application. This feature is highlighted in Cura by the use of all the different methods for database interaction that Sparx provides. This includes high level abstractions of database tables, rows and columns (courtesy of the Sparx DAL) all the way to the lowest level of standard JDBC communications. The reasoning behind using such a variety of database interaction methods is relatively simple. By comparing the different ways of accomplishing essentially the same task, you can determine which method is closest to your liking in terms of the ease, flexibility and control that it offers.

Data Access Styles

Each method of database interaction can be used to accomplish any task that needs to be done. However, one method might be superior to others under special circumstances. In this case, the one superior method will take fewer instructions or use more intuitive syntax for accomplishing the task than the other methods would. The other methods would work just as well but might be more cumbersome to use. These different methods are referred to as data access styles since all of them accomplish the same task but go about it slightly differently.

These different styles of accessing data within a Sparx application are not mutually exclusive: You can use as many styles in an application as possible. The only requirement is to be consistent when using them. You should not, for example, start accessing data using a high level style and intersperse that with low level SQL commands to the database.

The styles that are an integral part of Sparx aim to fulfill the requirements of both novices and experts. The high level abstractions are easy to work with since they allow easier visualization from a data flow perspective. You do not need to be concerned with the individual idiosyncrasies of the DBMS you are working with, nor do you need to be concerned with piecing together individual SQL statements and getting their results in obscure data structures. The lower level data access styles, however, give you as much ease as you want while giving you as much control over the small details as you can handle.

You can get an idea of how the different data access styles fit with each other by referring to the table below. It lists all the styles in order of decreasing level. Each style has a small description next to it.

DATA ACCESS STYLE	DESCRIPTION
Data Access Layer (DAL)	The Sparx DAL is the highest level abstraction of all the tables in a database and all the rows in those tables. Very readable and easy to debug. It can, however, only be used in a Java dialog handler.
XML Data Access (DML)	The Sparx DML is an XML representation of all the transactions possible with a database. It serves as a Java-free way to prototype an application allowing even junior level developers to develop sophisticated database applications. It does have limitations like lack of commit/rollback processing and can get cumbersome fairly rapidly.
Encapsulated SQL	Sparx also provides wrappers for standard SQL commands like insert, update and delete. These wrappers are intuitive ways of generating appropriate statements for execution on a database. While using these wrappers is complicated due to the required understanding of underlying database principles, it provides developers with the most flexibility.

In this section you will find examples of each type of data access style mentioned above. These examples are taken directly from Cura and will help to demonstrate the capabilities of each style. You need to understand, however, that just because a data access style is used in a particular area of Cura code, that doesn't make it the best style for that area. Cura's data layer was written with the express purpose of providing as many examples of real life code as possible.

Data Access Layer (DAL)

As mentioned before, this style is the highest level of data access available through Sparx. It utilizes Java objects to represent each table in the database. Further, each row and field in a table is represented by its own Java class. These classes are generated automatically from the XML database schema whenever an application is built. When programming data access using the DAL, you can always refer to the automatically generated API documentation using your application's ACE. The DAL API for an application along with its documentation allows a developer to quickly get up to speed on how to access data in any table of a database without going through the complicated setup, execution and result retrieval phases of JDBC programming. The whole process is painless and the end result is very human-readable Java code.

You can look at the code below as an example of how DAL is used for accessing data in a database. This code is taken directly from the `OrgDialog.java` file that is located in the `WEB-INF\classes\app\form` directory under the Cura home directory. This class, once compiled, takes care of processing the registration of every new client organization that is added to the system. Therefore, this is not a lightweight class that touches one table and is done. Even this method, which deals solely with deleting an organization from the system, touches multiple tables. More of this class will be explained in order of complexity.

```
protected void processDeleteData(DialogContext dc) throws SQLException,
NamingException {
```

```
    dialog.context.org.RegistrationContext rc =
(dialog.context.org.RegistrationContext) dc;
  ConnectionContext cc = dc.getConnectionContext();
  String org_id = dc.getRequest().getParameter("org_id");
  cc.beginTransaction();

  OrgTypeTable orgTypeTable =
dal.DataAccessLayer.instance.getOrgTypeTable();
  orgTypeTable.deleteOrgTypeRowsUsingOrgId(cc, new Long(org_id));
  OrgIndustryTable orgIndustryTable =
dal.DataAccessLayer.instance.getOrgIndustryTable();
  orgIndustryTable.deleteOrgIndustryRowsUsingOrgId(cc, new Long(org_id));
  OrgTable orgTable = dal.DataAccessLayer.instance.getOrgTable();
  OrgRow orgRow = orgTable.getOrgByOrgId(cc, new Long(org_id));
  orgTable.delete(cc, orgRow);
  cc.commitTransaction();

}
```

Dissecting this method is not non-trivial. However, the high level nature of the Sparx DAL coupled with its usage in the above section of code might make such an analysis unnecessary. For the sake of completion, however, and to ensure that everything in the above code is understood properly, it is analyzed at some length here.

```
protected void processDeleteData(DialogContext dc) throws SQLException,
NamingException {

dialog.context.org.RegistrationContext rc =
(dialog.context.org.RegistrationContext) dc;
…

}
```

The first and last lines in the code snippet above mark the beginning and end (respectively) of the processDeleteData method. As you can see, the only parameter that this method accepts is the dialog context for the currently submitted dialog. This context, passed as the generic Sparx DialogContext class, allows this method to access all the data submitted using the dialog. Since this is a very generic method, in the interests of gearing this code specifically towards this dialog, line 2 immediately declares an object of type dialog.context.org.RegistrationContext and proceeds to set its default value by casting the generic DialogContext object to the RegistrationContext type. Since all dialog context (even custom ones) are derived directly or indirectly from the generic Sprax DialogContext class, this is a valid operation. The end result is that the same data that was previously accessible using generic get and set methods, is now accessible using specific methods for each field of the dialog allowing the developer to trap any errors in the dialog context at compile time instead of runtime.

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

This line instantiates an object of type ConnectionContext and initializes it with the current database connection context from the dialog context object. This connection context object stores all the attributes needed to establish a connection to a database that is previously declared in a Sparx application's configuration files. It is also intelligent enough to re-use existing database connections (connection pooling) so there is no overhead for constantly creating and destroying database connections. With a connection context in hand, the developer can now access the database.

```
  String org_id = dc.getRequest().getParameter("org_id");
```

This line shows how to access parameters that are passed to a dialog using URL encoding. This line gets the value of the `org_id` parameter that should have been passed through the URL and stores that as a string for future use.

```
  cc.beginTransaction();
…
  cc.commitTransaction();
```

This line calls the `beginTransaction` method of the connection context obtained previously. This starts a database transaction which does not end until the `commitTransaction` method of the same connection context is called. A transaction, as must be known to DBAs and database programmers, is an atomic block of instructions which, should any one of the sub-instructions within the block fail, can be rolled back or undone without leaving any traces of data in the database. The corresponding `commitTransaction` is in the last line of this method. This means that all the data access code that is executed from now until the end of this method will be part of an atomic block of data access.

```
  OrgTypeTable orgTypeTable =
dal.DataAccessLayer.instance.getOrgTypeTable();
```

This line instantiates an object of type `OrgTypeTable`. This is the heart of the Sparx DAL. The `OrgTypeTable` class is a completely Java representation of the `org_type` table in the database and this line of code shows the typical way to instantiate such a DAL representation of a database table. This is the highest level abstraction of that database table and, as such, promotes extremely easy to use and intuitive syntax when dealing with this table in the database. This ease of use and intuitiveness will become apparent in later lines.

```
  orgTypeTable.deleteOrgTypeRowsUsingOrgId(cc, new Long(org_id));
```

This line, as might be understandable from just reading it, is deleting all records in the `org_type` table that match the given `org_id`. In this case, the `org_id` is provided as a URL encoded parameter (that was parsed earlier in this code) to the dialog.

```
  OrgIndustryTable orgIndustryTable =
dal.DataAccessLayer.instance.getOrgIndustryTable();
  orgIndustryTable.deleteOrgIndustryRowsUsingOrgId(cc, new Long(org_id));
```

These lines instantiate another Sparx DAL representation of a database table. This time it is a representation of the `org_industry` table. Further, after instantiating the table, they go on to delete all rows in this table that match the given `org_id`.

```
  OrgTable orgTable = dal.DataAccessLayer.instance.getOrgTable();
  OrgRow orgRow = orgTable.getOrgByOrgId(cc, new Long(org_id));
  orgTable.delete(cc, orgRow);
```

Finally, these lines use a slightly different way of deleting an organization from the main org table. The first line instantiates a DAL representation of the `org` database table. The second line instantiates a DAL representation of a row in this `org` database table and initializes it to the row whose `org_id` is equal to the value passed in to the dialog as a URL encoded parameter. This initialization is done by asking the `org` table to get the row for which the value stored in its `org_id` field is the same as the value of the `org_id` variable declared and defined earlier in this method. Lastly, it tells the `org` table to delete this row that it just found.

Since this is an atomic transaction, all three of the deletes have to be successful for the operation to be successful. Therefore, you will never be left with a record whose information is deleted from the `org` table but has orphan information left in the `org_industry` or `org_type` tables.

As you can see, even in this relatively uncomplicated example, the Sparx DAL shows its power by letting you write very high level human-readable code that not only works but works well. A more complex example of the Sparx DAL is shown below. This example might need less explanation due to the fact that the basics of working with DAL were already explained in the example above. Nevertheless, an explanation of additional DAL usages is at the end of this code snippet.

This section of code is also from the `OrgDialog.java` file and therefore it deals with the client account registration form. However, as the name suggests, this method handles all the data processing needed to add a record to the database. As with the previous example method from this file, this method also only requires the dialog context to be passed in as an argument. However, there are a few important differences between this method and the previous. First, this method accesses security information in the application by way of the username and password used to log into the system. Second, this method is dealing with insertion of data as opposed to deletion and is therefore slightly more involved. The length of this method is only reflective of the length of the process of insertion and not of the complexity of the same process. After reading through the code and, hopefully, gaining some insight into its working, you can proceed to the explanation of selected bits of this method after the code blocks.

```java
protected void processAddData(DialogContext dc) {
  HttpSession session = dc.getSession();
  AuthenticatedUser user = (AuthenticatedUser)
session.getAttribute(com.netspective.sparx.xaf.security.LoginDialog.DEFAU
LT_ATTRNAME_USERINFO);
  Map personMap = (Map) user.getAttribute("registration");
  BigDecimal personId = (BigDecimal) personMap.get("person_id");
  ConnectionContext cc = null;

  try {
    dialog.context.org.RegistrationContext rc = (RegistrationContext)dc;
    cc = rc.getConnectionContext();

    cc.beginTransaction();
    OrgTable orgTable = dal.DataAccessLayer.instance.getOrgTable();
    OrgRow orgRow = orgTable.createOrgRow();
    // populate the row with values from the dialog fields
    orgRow.populateDataByNames(rc);
    orgTable.insert(cc, orgRow);

    OrgIndustryTable orgIndustryTable =
dal.DataAccessLayer.instance.getOrgIndustryTable();
    OrgIndustryRow orgIndustryRow =
orgIndustryTable.createOrgIndustryRow();
    orgIndustryRow.setCrOrgId(rc.getCrOrgId());
    orgIndustryRow.setCrPersonId(rc.getCrPersonId());
    orgIndustryRow.setCrStampSqlExpr("sysdate");
    orgIndustryRow.setOrgIndustry(new Integer(rc.getOrgIndustry()));
    orgIndustryRow.setOrgId(orgRow.getOrgIdLong());
    orgIndustryTable.insert(cc, orgIndustryRow);

    OrgTypeTable orgTypeTable =
dal.DataAccessLayer.instance.getOrgTypeTable();
    OrgTypeRow orgTypeRow = orgTypeTable.createOrgTypeRow();
```

```
        orgTypeRow.setCrOrgId(rc.getCrOrgId());
        orgTypeRow.setCrPersonId(rc.getCrPersonId());
        orgTypeRow.setCrStampSqlExpr("sysdate");
        orgTypeRow.setOrgId(orgRow.getOrgIdLong());
        orgTypeRow.setOrgType(new Integer(rc.getOrgType()));
        orgTypeTable.insert(cc, orgTypeRow);

        if (rc.getParentOrgId() != null && rc.getParentOrgId().length() > 0)
{
            // add org relationships
            OrgRelationshipTable orgRelTable =
dal.DataAccessLayer.instance.getOrgRelationshipTable();
            // add a relationship row for this new org with the parent org
            OrgRelationshipRow orgRelRow =
orgRelTable.createOrgRelationshipRow();
            orgRelRow.setCrOrgId(rc.getCrOrgId());
            orgRelRow.setCrPersonId(rc.getCrPersonId());
            orgRelRow.setCrStampSqlExpr("sysdate");
            orgRelRow.setParentId(orgRow.getOrgIdLong());
            orgRelRow.setRelOrgId(new Long(rc.getParentOrgId()));
            orgRelRow.setRelType(
dal.table.OrgRelationshipTypeTable.EnumeratedItem.PARENT_OF_ORG);
            orgRelTable.insert(cc, orgRelRow);

            // add a relationship row for the parent org with the new row as
its child
            OrgRelationshipRow parentOrgRow =
orgRelTable.createOrgRelationshipRow();
            parentOrgRow.setCrOrgId(rc.getCrOrgId());
            parentOrgRow.setCrPersonId(rc.getCrPersonId());
            parentOrgRow.setCrStampSqlExpr("sysdate");
            parentOrgRow.setParentId(new Long(rc.getParentOrgId()));
            parentOrgRow.setRelOrgId(orgRow.getOrgId());
            parentOrgRow.setRelType(
dal.table.OrgRelationshipTypeTable.EnumeratedItem.CHILD_OF_ORG);
            orgRelTable.insert(cc, parentOrgRow);


            // get all the relationships of the parent org
            OrgRelationshipRows orgRelRows =
orgRelTable.getOrgRelationshipRowsByParentId(cc, new
Long(rc.getParentOrgId()));
            for (int i=0; i < orgRelRows.size(); i++) {
                OrgRelationshipRow tmpRow = (OrgRelationshipRow)
orgRelRows.get(i);

                if (tmpRow.getRelTypeEnum() ==
dal.table.OrgRelationshipTypeTable.EnumeratedItem.PARENT_OF_ORG) {
                    // parent of the new org's parent should be added as the new
org's ancestor
                    OrgRelationshipRow orgAncestorRow =
orgRelTable.createOrgRelationshipRow();
                    orgAncestorRow.setCrOrgId(rc.getCrOrgId());
                    orgAncestorRow.setCrPersonId(rc.getCrPersonId());
                    orgAncestorRow.setCrStampSqlExpr("sysdate");
                    orgAncestorRow.setParentId(orgRow.getOrgIdLong());
                    orgAncestorRow.setRelOrgId(tmpRow.getRelOrgId());
                    orgAncestorRow.setRelType(
dal.table.OrgRelationshipTypeTable.EnumeratedItem.ANCESTOR_OF_ORG);
                    orgRelTable.insert(cc, orgAncestorRow);

                    // the new org should be added as the descendent of the parent
of the new org's parent
                    OrgRelationshipRow ancestorOrgRow =
orgRelTable.createOrgRelationshipRow();
                    ancestorOrgRow.setCrOrgId(rc.getCrOrgId());
                    ancestorOrgRow.setCrPersonId(rc.getCrPersonId());
                    ancestorOrgRow.setCrStampSqlExpr("sysdate");
                    ancestorOrgRow.setParentId(tmpRow.getRelOrgId());
                    ancestorOrgRow.setRelOrgId(orgRow.getOrgIdLong());
                    ancestorOrgRow.setRelType(
dal.table.OrgRelationshipTypeTable.EnumeratedItem.DESCENDENT_OF_ORG);
```

```
            orgRelTable.insert(cc, ancestorOrgRow);

      }
    }

  }

  cc.commitTransaction();
  dc.getRequest().setAttribute("org_id", orgRow.getOrgId());
  dc.getRequest().setAttribute("org_name", orgRow.getOrgName());
} catch (Exception e) {
  e.printStackTrace();
  if (cc != null) {
    try {
      cc.rollbackTransaction();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
  }
}
}
```

Having explained the previous method in considerable detail, this method should not need much explanation. However, since there are newer coding methods dealt with here, it is essential that they be touched upon at least slightly.

```
HttpSession session = dc.getSession();
AuthenticatedUser user = (AuthenticatedUser)
session.getAttribute(com.netspective.sparx.xaf.security.LoginDialog.DEFAU
LT_ATTRNAME_USERINFO);
Map personMap = (Map) user.getAttribute("registration");
BigDecimal personId = (BigDecimal) personMap.get("person_id");
ConnectionContext cc = null;
```

These first few lines help setup all the data needed for the rest of the method. The first line stores the current session in the session variable. The facility of sessions is provided by the application server itself and is available to all applications, not just Sparx applications. The next few lines are exclusive to Sparx, however and are used to get the information for the currently authenticated user (i.e. the one who logged into the system) and specifically, his person_id. This will be used later on in the method. The last line is the familiar connection context being instantiated. It will be initialized later since it is essential for all database access and cannot be used uninitialized.

```
try {
   dialog.context.org.RegistrationContext rc = (RegistrationContext)dc;
   cc = rc.getConnectionContext();

   cc.beginTransaction();
   OrgTable orgTable = dal.DataAccessLayer.instance.getOrgTable();
   OrgRow orgRow = orgTable.createOrgRow();
   // populate the row with values from the dialog fields
   orgRow.populateDataByNames(rc);
   orgTable.insert(cc, orgRow);
```

These next few lines are the beginning of all the processing done by this method. Everything involving database access is enclosed in a try-catch block that is present to ensure any and all exceptions are caught and dealt with properly. The first two lines inside the try block are setting up the specific RegistrationContext object and using it to initialize the connection context.

Once the transaction is begun, the fun starts. The `org` table is instantiated using the `OrgTable` DAL class and immediately a row of the `org` table is instantiated using the `OrgRow` class. The next line is special since it automates a lot of the work involved in populating a row of a table prior to insertion in the database. In this case, the DAL allows a developer to ask the row to populate itself based on the various fields available in the dialog context. All fields that are named the same as a field in the `OrgRow` have their values copied into the `OrgRow`, ready for insertion. Of course if the `OrgRow` cannot find a field in the dialog context, it will remain empty and it is the developer's responsibility to fill that field in manually before inserting the row into the database. As expected, this `populateDataByNames` method only requires the dialog context (or, in this case, the specific Registration context) to do its job.

The final line does the inevitable and actually inserts the row into the database using the current connection context. These few lines are identical in process to how any data is inserted into the database. You will see this pattern repeated throughout this method for the different pieces of data necessary to support a record in the `org` table.

```
    OrgIndustryTable orgIndustryTable =
dal.DataAccessLayer.instance.getOrgIndustryTable();
    OrgIndustryRow orgIndustryRow =
orgIndustryTable.createOrgIndustryRow();
    orgIndustryRow.setCrOrgId(rc.getCrOrgId());
    orgIndustryRow.setCrPersonId(rc.getCrPersonId());
    orgIndustryRow.setCrStampSqlExpr("sysdate");
    orgIndustryRow.setOrgIndustry(new Integer(rc.getOrgIndustry()));
    orgIndustryRow.setOrgId(orgRow.getOrgIdLong());
    orgIndustryTable.insert(cc, orgIndustryRow);
```

These lines illustrate the similarities.

XML Data Access

XML-based data access is one of the more interesting aspects of a Sparx application. This data access method is referred to by Sparx as the Data Manipulation Language or DML. While not as powerful as Java for database interaction, it is a full-fledged solution for those people (developers or otherwise) who do not wish to code in Java. As you will shortly see in an example, you can accomplish extremely complex tasks with DML while retaining the ease of use of XML coupled with the advantage of error checking against a DTD for compile-time debugging.

A picture is, indeed, worth a thousand words or more. Therefore, instead of explaining DML in detail, you should look at the DML sample shown below and compare it with the source code for the Java dialog handler for the same class. This DML is taken from the `org.registration` dialog located in the `app-org-dialogs.xml` file (which is located in Cura's `ui` directory). This DML corresponds to the Java code from `OrgDialog.java` that is explained in detail earlier in this document. If you look inside the `app-org-dialogs.xml` file, you will notice that this DML is commented out since it has been superceded by the custom Java dialog handler code in `OrgDialog.java`. Without the custom Java dialog handler code, this DML is executed every time the dialog is submitted, provided the DML is uncommented.

Of course, the key player in all this is the execute method of the default Java dialog handler class. If that method is ever executed, it automatically runs all the DML it finds in a dialog. Theoretically, therefore, DML can co-exist with a custom Java dialog handler class and be used in conjunction with it. In practice, however, if a custom Java dialog handler comes into the picture, the DML usually becomes redundant and is removed from the dialog.

```
<populate-tasks data-cmd="edit,delete">
  <exec-statement report="none" store-type="row-fields" store="form:*">
    select * from org where org_id = ?
    <params>
      <param value="request:org_id"/>
    </params>
  </exec-statement>
</populate-tasks>

<execute-tasks data-cmd="add">
  <exec-transaction command="begin"/>

    <exec-dml command="insert" table="org" auto-
inc="org_id,org_org_id_seq" auto-inc-store="request-attr:org_id"
fields="org_code,org_name,org_abbrev,ownership,ticker_symbol,employees,ti
me_zone"/>
    <exec-dml command="insert" table="org_industry"
columns="org_id=request-attr:org_id,system_id=custom-
sql:oind_system_id_seq.nextval" fields="org_industry"/>
    <exec-dml command="insert" table="org_type" columns="org_id=request-
attr:org_id,system_id=custom-sql:otyp_system_id_seq.nextval"
fields="org_type"/>
      <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=rel_org_id"
        columns="parent_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1010">
        <conditional action="execute" has-value="form:parent_org_id"/>
      </exec-dml>

      <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=parent_id"
        columns="rel_org_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1030">
        <conditional action="execute" has-value="form:parent_org_id"/>
      </exec-dml>

      <exec-statement stmt-src="org.ancestor-list" report="none" store-
type="row-fields" store="request-attr:ancestor-list"/>

      <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=rel_org_id"
        columns="parent_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1010">
        <conditional action="execute" has-value="form:parent_org_id"/>
      </exec-dml>

  <exec-transaction command="end"/>

  <exec-redirect url="config-expr:${create-app-
url:/account/home.jsp}?org_id=${request-
attr:org_id}&amp;org_name=${form:org_name}"/>

</execute-tasks>

<execute-tasks data-cmd="edit">
```

```
    <exec-dml command="update" table="org"
fields="org_code,org_name,org_abbrev,ownership,ticker_symbol,employees,ti
me_zone" where="org_id = ?" where-bind="request:org_id"/>
  <exec-redirect url="config-expr:${create-app-
url:/account/home.jsp}?org_id=${request:org_id}"/>
</execute-tasks>

<execute-tasks data-cmd="delete">
  <exec-transaction command="begin"/>
    <exec-dml command="delete" table="org_industry" where="org_id = ?"
where-bind="request:org_id"/>
    <exec-dml command="delete" table="org_type" where="org_id = ?" where-
bind="request:org_id"/>
    <exec-dml command="delete" table="org_relationship" where="parent_id
= ?" where-bind="request:org_id"/>
    <exec-dml command="delete" table="org_relationship" where="rel_org_id
= ?" where-bind="request:org_id"/>
    <exec-dml command="delete" table="org" where="org_id = ?" where-
bind="request:org_id"/>
  <exec-transaction command="end"/>
  <exec-redirect url="create-app-url:/account"/>
</execute-tasks>
```

As you might be able to see, there are two main types of DML code in the XML shown above. These are signified by the two different top level XML tags in the source above.

♦ **populate-values:** This tag is used to populate a dialog prior to being rendered. It is the DML analog to the `populateValues` method in a custom Java dialog handler class

♦ **execute-tasks:** This tag is used to contain DML statements that are executed when a dialog is submitted. It is the DML analog to the execute method in a custom Java dialog handler class.

```
<populate-tasks data-cmd="edit,delete">
  <exec-statement report="none" store-type="row-fields" store="form:*">
    select * from org where org_id = ?
    <params>
      <param value="request:org_id"/>
    </params>
  </exec-statement>
</populate-tasks>
```

The DML snippet above contains the entire populate-tasks code for the `org.registration` dialog. As you might be able to guess, this tag and the DML inside it handles dialog population only if the data command is edit or delete. Therefore, the DML inside this populate-tasks tag will execute only if you use the dialog to edit or delete a record.

The only statement inside the populate-tasks section uses the **exec-statement** tag. This statement executes a SQL statement (stored in the **statements.xml** file in Cura) and automatically populates all dialog fields that are named the same as the fields returned by the SQL statement. Therefore, one simple statement takes care of all the population issues that this dialog might need to face.

```
<execute-tasks data-cmd="add">
  <exec-transaction command="begin"/>

    <exec-dml command="insert" table="org" auto-
inc="org_id,org_org_id_seq" auto-inc-store="request-attr:org_id"
fields="org_code,org_name,org_abbrev,ownership,ticker_symbol,employees,ti
me_zone"/>
```

```
        <exec-dml command="insert" table="org_industry"
columns="org_id=request-attr:org_id,system_id=custom-
sql:oind_system_id_seq.nextval" fields="org_industry"/>
        <exec-dml command="insert" table="org_type" columns="org_id=request-
attr:org_id,system_id=custom-sql:otyp_system_id_seq.nextval"
fields="org_type"/>
        <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=rel_org_id"
        columns="parent_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1010">
        <conditional action="execute" has-value="form:parent_org_id"/>
        </exec-dml>

        <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=parent_id"
        columns="rel_org_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1030">
        <conditional action="execute" has-value="form:parent_org_id"/>
        </exec-dml>

        <exec-statement stmt-src="org.ancestor-list" report="none" store-
type="row-fields" store="request-attr:ancestor-list"/>

        <exec-dml command="insert" table="org_relationship" auto-
inc="system_id,OrgRel_system_id_SEQ"
        fields="parent_org_id=rel_org_id"
        columns="parent_id=request-attr:org_id,cr_stamp=custom-
sql:sysdate,cr_org_id=session:organization,cr_person_id=session:person_id
,rel_begin=custom-sql:sysdate,rel_type=custom-sql:1010">
        <conditional action="execute" has-value="form:parent_org_id"/>
        </exec-dml>

    <exec-transaction command="end"/>

    <exec-redirect url="config-expr:${create-app-
url:/account/home.jsp}?org_id=${request-
attr:org_id}&amp;org_name=${form:org_name}"/>

</execute-tasks>
```

The `execute-tasks` tag used in the DML snippet above contains all the statements that are executed when the dialog is submitted in the add mode. As you might be able to guess, the beginning and ending `exec-transaction` tags bundle all the statements inside them as an atomic transaction, thus ensuring either the success or failure of all the statements inside. Therefore, you will never be left with a record whose information is partially inserted into a few tables but which could not be inserted into the rest of the supporting tables.

You should keep in mind, however, that DML does not explicitly allow you to commit or rollback transactions. If a transaction fails, the DML will cause an error message to appear in place of the dialog in your application. In most circumstances, this behavior is highly undesirable. However, the Sparx DAL, when used in conjunction with a Java dialog handler, allows a developer to commit a transaction or roll it back depending on whether it was executed successfully. This lets a developer choose what to do in either circumstance and is usually the better approach when dealing with transaction processing in databases.

Each `exec-dml` tag is responsible for direct database interaction. Depending on the value of the `command` attribute (insert, update or delete) and the `table` attribute, an `exec-dml` statement is parsed to generate SQL targeting the specified operation on the specified table using the rest of the data in the tag.

```
    <exec-dml command="insert" table="org" auto-
inc="org_id,org_org_id_seq" auto-inc-store="request-attr:org_id"
fields="org_code,org_name,org_abbrev,ownership,ticker_symbol,employees,ti
me_zone"/>
```

The above `exec-dml` statement, for example, inserts into the `org` table the values of the dialog fields specified in the `fields` attribute. The `auto-inc` attribute specifies which Oracle sequence to use to get the next value of `org_id` while the `auto-inc-store` attribute says that the resulting value of `org_id` should be stored as a request attribute once the statement is executed. When a columns attribute is specified instead of a fields attribute, the values that are inserted into the table can be custom values taken from environment variables or static strings or even the results of other statements.

```
   <exec-redirect url="config-expr:${create-app-
url:/account/home.jsp}?org_id=${request-
attr:org_id}&amp;org_name=${form:org_name}"/>
```

Once all the `exec-dml` statements are done, the last type of tag you observe in the code snippet is the `exec-redirect` tag. This tag accepts one attribute: the URL to redirect the user to once the previous task has been completed.

In conclusion, then, you can see that with relatively easy to understand XML, anyone can leverage the power of Sparx to create full-fledged applications without writing a single line of Java. Developers wishing to prototype an application quickly or not willing to delve into a lot of Java to accomplish simple record addition, editing or deletion are delighted to use a non-Java way of manipulating data in a database. Further, the inherent build-time debugging capabilities of DML make it a boon for non-Java developers by reducing the time spent hunting for the cause of errors and putting that time towards fixing any syntax errors that might exist with the DML.

Encapsulated SQL

Encapsulated SQL is at once low level and simple, if you know SQL well. It is a method of database interaction that can be used in a Java dialog handler. Instead of providing you with high level abstractions of database tables and fields, it gives you only low level abstractions of the different types of SQL statements. Provided you know SQL well, you can execute arbitrary SQL in your database to accomplish the desired goals. The three types of SQL statements that are abstracted are insert, update and delete. For each statement, all supporting clauses (such as where, order by, etc) are supported.

```
protected void processAddAction(DialogContext dc) {
  try {
    String cr_person_id =
(String)dc.getRequest().getAttribute("personId");
    // begin the transaction
    dc.beginSqlTransaction();
```

```
    StatementManager sm = dc.getStatementManager();
    DatabaseContext dbContext =
DatabaseContextFactory.getContext(dc.getRequest(),
dc.getServletContext());

    String fullName = dc.getValue("name_first") + " " +
dc.getValue("name_middle") + " " + dc.getValue("name_last");

    // fields represent a mapping for dialog fields to database column
names
    String fields = "name_last,name_first,name_middle,name_prefix," +
       "ssn,gender";
    // columns represent a mapping of database column names to literal
values?
    String columns = "cr_stamp=custom-sql:sysdate,complete_name=custom-
sql:'" + fullName + "'";
    String autoinc = "person_id,per_person_id_seq";
    String autoincStore = "request-attr:new_person_id";
    dc.executeSqlInsert("person", fields, columns, autoinc,
autoincStore);
    BigDecimal person_id =
(BigDecimal)dc.getRequest().getAttribute("new_person_id");

    // associate this new contact with an organization
    fields = "organization=rel_org_id,organization_relation=rel_type";
    columns = "cr_stamp=custom-sql:sysdate,cr_person_id=custom-sql:" +
cr_person_id + ",parent_id=custom-sql:" + person_id +
       ",rel_begin=custom-sql:sysdate,record_status_id=custom-sql:1" ;
    autoinc = "system_id,PerRel_system_id_seq";
    autoincStore = "request-attr:new_personorg_rel_id";
    dc.executeSqlInsert("personorg_relationship", fields, columns,
autoinc, autoincStore);


    // process address information
    fields = "line1,line2,city,state,zip,country";
    columns = "parent_id=custom-sql:" + person_id +
       ",cr_stamp=custom-sql:sysdate";
    autoinc = "system_id,PerAddr_system_id_SEQ";
    autoincStore = "request-attr:new_address_id";
    dc.executeSqlInsert("person_address", fields, columns, autoinc,
autoincStore);


    // process email information
    if (dc.getValue("email") != null && dc.getValue("email").length() >
0) {
      fields = "email=method_value,organization=cr_org_id";
      columns = "method_name=custom-sql:'" + CONTACT_METHOD_NAME_EMAIL +
"',method_type=custom-sql:" +
        CONTACT_METHOD_TYPE_EMAIL + ",parent_id=custom-sql:" + person_id
+ ",cr_stamp=custom-sql:sysdate";
      autoinc = "system_id,PerCont_system_id_SEQ";
      autoincStore = "request-attr:new_contact_id";
      dc.executeSqlInsert("person_contact", fields, columns, autoinc,
autoincStore);
    }

    if (dc.getValue("url") != null && dc.getValue("url").length() > 0) {
      // process url information
      fields = "url=method_value,organization=cr_org_id";
      columns = "method_name=custom-sql:'" + CONTACT_METHOD_NAME_URL +
"',method_type=custom-sql:" +
        CONTACT_METHOD_TYPE_URL + ",parent_id=custom-sql:" + person_id +
",cr_stamp=custom-sql:sysdate";
      autoinc = "system_id,PerCont_system_id_SEQ";
      autoincStore = "request-attr:new_contact_id";
      dc.executeSqlInsert("person_contact", fields, columns, autoinc,
autoincStore);
    }
```

```
    if (dc.getValue("phone") != null && dc.getValue("phone").length() >
0) {
        // process url information
        fields = "phone=method_value,organization=cr_org_id";
        columns = "method_name=custom-sql:'" + CONTACT_METHOD_NAME_PHONE +
"',method_type=custom-sql:" +
            CONTACT_METHOD_TYPE_PHONE + ",parent_id=custom-sql:" + person_id
+ ",cr_stamp=custom-sql:sysdate";
        autoinc = "system_id,PerCont_system_id_SEQ";
        autoincStore = "request-attr:new_contact_id";
        dc.executeSqlInsert("person_contact", fields, columns, autoinc,
autoincStore);
    }
    // end transaction
    dc.commitSqlTransaction();
    dc.getRequest().setAttribute("person_id", person_id.toString());
    dc.getRequest().setAttribute("person_name", fullName);
  }
  catch (TaskExecuteException tee) {
    tee.printStackTrace();
    try {
        System.out.println("calling rollback!!");
        dc.rollbackSqlTransaction();
    } catch (TaskExecuteException e) {
        e.printStackTrace();
    }
  }
}
```

Conclusion

Over the last few chapters you have seen a very complex project management application dissected to reveal the simple set of parts that are combined to make it. You have seen how Sparx can be used to create these application components using very intuitive and highly understandable code. Whether it is XML definitions of components or custom Java supporting code, any programmer with just the faintest inkling of database programming can understand the Sparx code that any other programmer writes. The sheer power of the database manipulation code that Sparx provides a developer means that no-one has to be a Java or database guru to write great applications. The speed at which an entire Sparx application can be developed from the ground up is a testament to its natural way of doing things. The ease with which errors in your Sparx code can be located at compile-time or run-time shows the vast array of utilities at your disposal for debugging and unit testing.

Keeping the power of Sparx in mind, Cura boils down to nothing more than a few strategic dialogs and reports coupled with a database schema to store all the data being manipulated. Similarly, most (if not all) complex applications written with Sparx end up being key elements bound together by a well designed UI and database schema. When you look at a complex application and can see such a simplistic implementation possible, nothing becomes too difficult to write.

Developers can focus on creating a good application design while data modelers ensure that the database schema they are using is scalable and well suited to the application. UI specialists can make sure that interaction with the user is as productive as possible given the data that needs to be shown or input. Graphic artists can concentrate on creating the best look for the application. Each one of these important tasks is made easy by harnessing Sparx and using its advanced set of tools. Finally, software development is wrenched from the hands of the elite and put well within the reach of the average.