XMLPipeDB User and Developer's Manual

The XMLPipeDB Group
Loyola Marymount University

July 13, 2006

Contents

1 Overview

XMLPipeDB is a suite of tools for managing, querying, importing, and exporting information to/from XML data based on some specific XML schema (XSD). While its applicability is fairly general, the original motivation for XMLPipeDB is the management of biological data from different sources. Thus, XMLPipeDB's end-user applications are bioinformatics-oriented, although other applications may be developed using the project's underlying tools.

1.1 Who Should Read What Part of this Document

- If you are a GenMAPP or other biological database user, you will want to read Section 2.
- If you are a bioinformatics software developer, you may also be interested in Sections 3.
- If you are a database administrator who is preparing an installation of one of XMLPipeDB's end-user applications, you will want to read Section 4.
- If you are a software developer or database designer who is interested in learning how XMLPipeDB's end-user applications were developed, you will want to read Section 5.
- Finally, if you are a software developer who wishes to participate in or contribute to the XMLPipeDB project, you will want to read Section 6.

2 End-User Software

2.1 Pre-Built GenMAPP Files

If your primary interest is to use GenMAPP for a particular organism, please consult the following list for GenMAPP files that have already been created using GenMAPP Builder (Section 2.2). In many cases, you can simply download them and point GenMAPP to them, and you'll be ready to load up expression data sets. Full details on GenMAPP are available on the GenMAPP Web site (`http://www.genmapp.org`).

- *Escherichia coli K12*
- *Pseudomonas putida*
- *Bacillus subtilis*

2.2 GenMAPP Builder

GenMAPP Builder is an application for creating GenMAPP Gene Database files, or GDBs. These files are actually Microsoft Access database files (MDBs) with a `.gdb` filename extension.[1]

The application works by first importing supported data files into a relational database. The database can then be queried by organism in order to produce a GenMAPP database file. While one UniProt import per species would be the norm, there is no actual restriction on the content of the imported XML. With a sufficiently large and fast workstation, one can theoretically import the entire UniProt dataset into GenMAPP Builder's internal database.

2.2.1 Supported Data Sets

The current version of GenMAPP Builder requires data from the following types of files in order to build a GenMAPP Gene Database file:

- UniProt XML
- Gene Ontology (GO) OBO XML
- Tab-delimited GO gene associations (various sources, primarily GOA)

Typically, the GO file need only be loaded once, since GO makes the entire set of terms available as a single download. UniProt XML files can be downloaded by species from the Integr8 Web interface. While one UniProt import per species would be the norm, there is no actual restriction on the content of the imported XML. With a sufficiently large and fast workstation, one can theoretically import the entire UniProt dataset into GenMAPP Builder's internal database.

Once the source files have been imported into the relational database, they are no longer needed; exports can be performed any number of times after that. In practice, however, periodic reloads will be necessary in order to remain in sync with the datasets' providers.

2.2.2 Requirements

To use GenMAPP Builder, you need:

- A Java 5 runtime environment
- A relational database with an available JDBC 3 driver; however, at this writing, only PostgreSQL has been used and tested
- One or more data sets downloaded from the Internet, from data sources that are currently supported by GenMAPP Builder

[1] GenMAPP handles three types of MDB files, distinguished by an extension other than `.mdb`: Gene Databases (`.gdb`), MAPPs (`.mapp`), and Expression Datasets (`.gex`).

Once these are properly installed and operational, you can perform these primary functions:

1. Import one or more supported data files into the GenMAPP Builder database
2. Build a GenMAPP Gene Database file based on a particular species in the database

The built files can then be read directly by GenMAPP.

2.2.3 Database Setup

Using GenMAPP Builder requires one setup step that is external to the GenMAPP Builder application: setting up the intermediate relational database that GenMAPP Builder uses to store imported XML information. This database is technically any relational database for which a JDBC 3 driver is available, although thus far only PostgreSQL has been used and tested.

To set up the GenMAPP Builder intermediate database, perform the following steps:

1. Install the database server — this varies according to the specific database server software that you have chosen. For PostgreSQL, this involves installing the software, initializing a database cluster using the `initdb` program, then starting the server using the `pg_ctl` script or invoking `postmaster` directly.
2. Load the GenMAPP Builder schema file (`gmbuilder.sql`) into the relational database. Specifics for this step also vary depending on the database server that is used. For PostgreSQL, this step requires the creation of a named database using the `createdb` command, then importing the `gmbuilder.sql` file into that database through `psql`.

Once the database has been set up, all other configuration activities can be performed from the GenMAPP Builder application.

2.2.4 Configuration

When GenMAPP Builder is run for the first time, a configuration dialog appears (Figure 1). A valid configuration is required in order for GenMAPP Builder to operate properly; many problems are caused by incorrect database settings. The configuration can be changed at any time via the *Configure Database* command in the *File* menu.

The configuration dialog provides an interface for a wide variety of database-related settings; many of these can be left at their default values. The most important item to configure is the database server itself, a sample of which is shown in Figure 1:

1. Select the *Platforms* radio button.
2. Select the relational database that you are using from the drop-down menu.

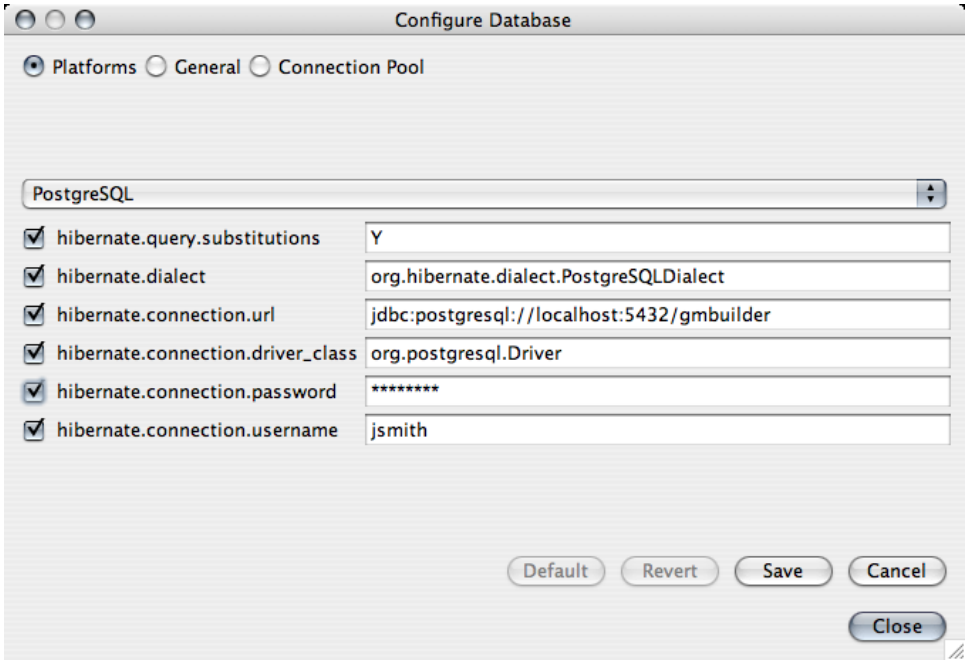

Figure 1: GenMAPP Builder configuration dialog.

3. Fill out the relevant settings from the list that appears.

The example shown in Figure 1 shows a GenMAPP Builder installation that uses a PostgreSQL relational database called *gmbuilder* on a server that is running on the same machine as GenMAPP Builder (*localhost*) using the default PostgreSQL network port, 5432. The standard PostgreSQL database driver is used (*org.postgresql.Driver*), and the database username is *jsmith* with an accompanying password.

Most of the time, these are the only settings to modify. Click the *Save* button when configuration is complete.

GenMAPP Builder relies on the Hibernate library for its database interactions; a full reference of all settable parameters can be found on the Hibernate Web site (`http://www.hibernate.org`).

2.2.5 File Import

A "fresh" install of GenMAPP Builder contains no data — supported files must be *imported* into the relational database first. There is one *Import* command for each supported dataset, and these commands are all available under the application's *File* menu (Figure 2).

Each import command works in the same way: once chosen, a file chooser dialog appears. Locate the file to be imported through this dialog, then click on *Import*. The

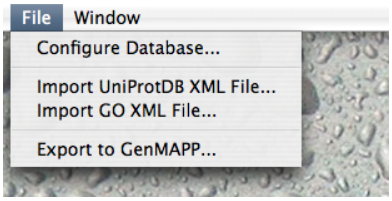

Figure 2: GenMAPP Builder *File* menu.

import process then begins. A progress display tracks the status of the import, from which two commands are available: *Pause* or *Cancel*.

The *Pause* command allows you to stop the import operation temporarily, to be resumed at a later time. This may be useful for large data sets on a computer that can't be "left alone" to complete the entire import operation. GenMAPP Builder tracks the list of paused imports for you; the primary requirement is that you do *not* delete the files being imported until the operation is complete.

The *Cancel* command cancels the *entire* import, including data that has already been read. This command is useful if you change your mind during an import operation, and decide not to load any of that information into the relational database after all.The database remains in precisely the state that it was prior to the import command, as if the import was never invoked at all.

Once an import operation finishes, the data in the imported file now becomes accessible as a set of relational database tables.

2.2.6 Browsing the Database

GenMAPP Builder includes a query engine that allows you to browse the contents of the underlying relational database. With this engine, you can look at the species that are available and all imported information about the species' genes and proteins. If a compatible GO gene associations file has also been imported into the database, you can browse the GO terms that have been associated with that species as well.

The query engine allows direct SQL or HQL queries, returning the results of these queries in a table or tree view (for SQL and HQL queries, respectively). The direct queries allow for maximum flexibility in examining the loaded data, but they require detailed knowledge of the intermediate relational database schema. For convenience, a number of particularly useful queries have been pre-loaded into GenMAPP Builder.

Preloaded queries are available in the *Query* menu. When a preloaded query is invoked, a dialog may appear requesting required parameters such as taxon ID, protein name, etc. Fill these parameters out and click on the *Perform Query* button to view the results.

A third query mechanism is available in the form of a Google-style "free text" query. For this type of query, enter the text that you would like to find, and GenMAPP Builder

will return all records that somehow match this text. The query is a simple substring search on the following fields: (this list is to be specified).

2.2.7 Building a GenMAPP Gene Database

Once the underlying relational database contains all of the information that you need (i.e., you have imported all of the XML data sets that you wish to export to a GenMAPP Gene Database), you can build a GenMAPP Gene Database file. To do this, choose the *Export to GenMAPP* command from the *File* menu.

Definitions The following definitions are helpful in assimilating the information in this section:

- The term "system" by itself is a generic term referring to any gene ID system in the abstract (e.g., UniProt).
- A "systems table" generically refers to a table containing data from a particular gene ID system (e.g., the *UniProt* table in the GenMAPP Gene Database file is a systems table).
- *Systems* (capitalized, italicized) is the specific table used by GenMAPP to determine which gene ID systems (and thus, systems tables) are present in the Gene Database.
- A "relations table" generically refers to a table that relates two gene ID systems (e.g., *UniProt-EMBL* is a relations table that links UniProt IDs with EMBL IDs).
- *Relations* (capitalized, italicized), is the specific table used by GenMAPP to determine which gene ID systems are related in the database (and thus, which relations tables should exist).

Export Dialog The export operation opens with a dialog that requests the following information. Where possible, the destinations of these parameters in the final Gene Database are indicated.

- The owner of the database (limited to 200 characters): This goes in the *Owner* field of the *Info* table.
- The creation date of the database: This defaults to the current date, and goes in the *Version* and *Modify* fields of the *Info* table.
- The species to export: GenMAPP Builder examines the intermediate relational database and provides a drop-down menu for the species that are available there. Select the species for which you would like to export a Gene Database. GenMAPP Builder will automatically use this information wherever this is needed in the Gene Database, such as the *Info* table and the *Species* fields of the systems tables.

- The systems tables to include: GenMAPP Builder presents a checklist of what systems tables are available.
- One of these systems tables then needs to be designated as the *Model Organism Database* (MOD) system for the Gene Database. For a UniProt-centric Gene Database, this would be UniProt. The table name goes in the *MODSystem* field of the *Info* table.
- Systems table display order: An explicit ordering can be specified for the systems tables, determining the order in which the systems will display in the GenMAPP Gene Finder and Backpage. This information is stored in the *DisplayOrder* field of the *Info* table.
- "Primary" systems tables: systems tables whose database references are used to build the relations tables (see next item) are considered to be *primary*. Relations tables that include such systems are said to have *direct* relations, while relations tables whose linkages are transitively built from primary database references are said to be *inferred* (e.g., if UniProt is marked as a primary systems table, and it contains references to PDB and EMBL, then UniProt-PDB and UniProt-EMBL relations tables are direct, while a PDB-EMBL relations table is inferred, because the UniProt references are used to determine related IDs between PDB and EMBL).
- The relations tables to include: GenMAPP Builder automatically generates a default list according to an internal algorithm (specified in Section 2.2.9); the final list can then be modified (added to or deleted from) if desired.
- The location and name of the destination Gene Database file: As required by GenMAPP, the file has a `.gdb` extension. In addition, while not strictly enforced, the filename itself should follow this naming convention: two-letter code for genus/species, dash, the letters "Std" meaning standard or official, underscore, date in `yyyymmdd` format, file extension `.gdb`. For example, an *E. coli* Gene Database generated on June 6, 2006 should, by convention, have the filename `Ec-Std_20060606.gdb`.

When these export parameters have been set, the export operation begins. As with the import operation, a progress display tracks the status of the export. *Pause* and *Cancel* commands, which operate in the same way as their import operation counterparts, are also available while the export operation is taking place.

Once the export operation finishes, the specified destination file can be opened and used by GenMAPP.

2.2.8 Implementation Notes

The configuration, import, and query functions in GenMAPP Builder make direct use of the XMLPipeDB Utilities (*xmlpipedbutils*) library.

2.2.9 Database Details

GenMAPP Builder performs the following specific details in generating a Gene Database file that can be used by GenMAPP. Based on the chosen species, GenMAPP Builder first extracts the needed information from its intermediate relational database. Once this information has been retrieved, it is written to the specified Gene Database file as detailed in the rest of this section.

A sample GenMAPP Gene Database schema that results from the export operation appears in Figure 3. The schema in the figure is the result of a GenMAPP Builder export for *E. coli* with UniProt as the model organism system table.

1. The export begins by copying a blank GenMAPP Gene Database file, `GeneDBTmpl.gtp`, to the specified destination file (i.e., the one specified in the export dialog). The file is provided by GenMAPP.org, and consists of the following four tables: *Info*, *Systems*, *Relations*, and *Other*. The template file is actually a Microsoft Access database file with the extension `.gtp`. GenMAPP Builder comes with a default template file, but a different template file may be used since it may be changed periodically by GenMAPP.org.

2. The *Owner* field of the *Info* table is filled with the *Owner* text given in the export dialog.

3. The *Version* and *Modify* fields of the *Info* table are filled with the date given in the export dialog.

4. The *Species* field of the *Info* table is set to the genus-species designation given in the export dialog.

5. The systems tables specified in the export dialog are created. For UniProt-centric Gene Databases, at a minimum the *UniProt* table is created according to the schema shown in Figure 3. Other systems tables take on the schema of the *EMBL*, *PDB*, *InterPro*, etc. tables in the figure. Each of these tables has a corresponding row in the *Systems* table.

6. For each systems table that gets made, the *Date* field of the corresponding record in the *Systems* table is set to the current date.

7. The *OrderedLocusNames* table is created from the *ordered locus* name entries in the UniProt XML. Note, for *E. coli*, this table is renamed "Blattner" — an *E. coli*-specific post-processing step.

8. In the export dialog, the user was asked to designate one system as the MOD for the database. For UniProt-centric databases, it should be UniProt. The *MODSystem* field of the *Info* table is set to the name of this MOD system.

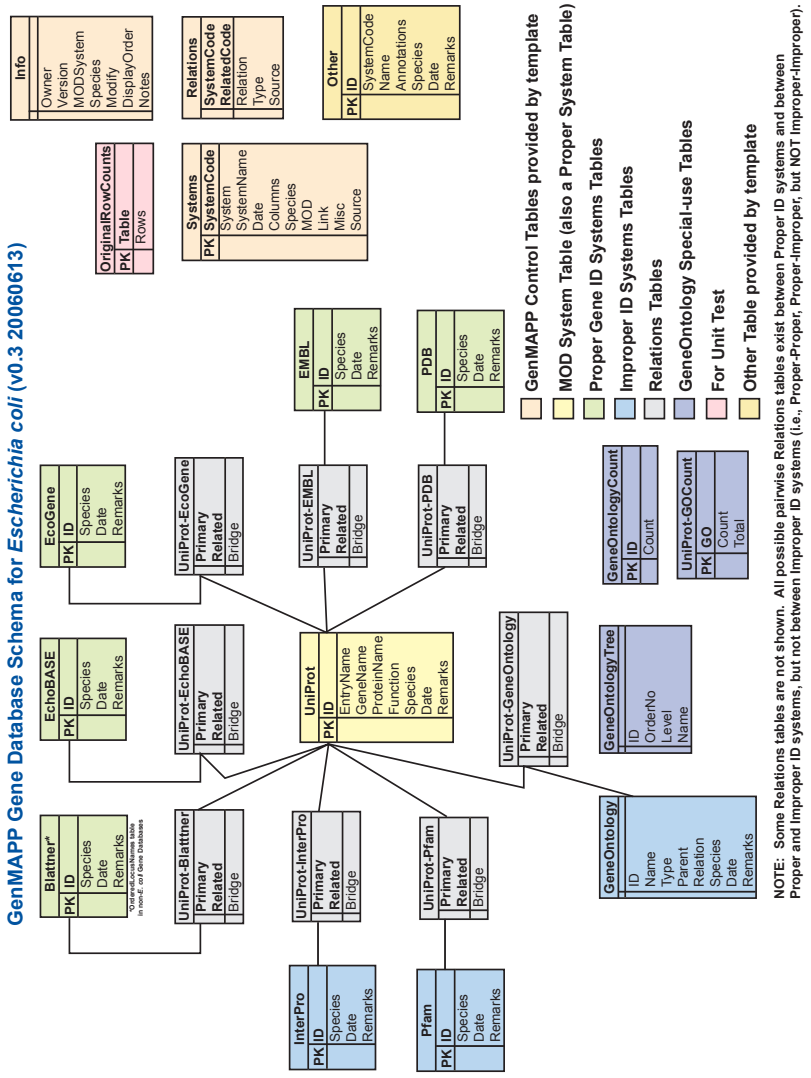

Figure 3: Sample GenMAPP Gene Database for *E. coli*.

9. The *DisplayOrder* field of the *Info* table is set to the systems table order specified in the export dialog. The table sequence is specified by the abbreviated code (i.e., *SystemCode*) of each systems table delimited by the pipe character (e.g., "|S|Em|").

10. The relations tables called for in the export dialog are created. The default list of relations tables in that dialog is generated using the following algorithm: first, each system table is designated as being *Proper* or *Improper*, according to the *Misc* field in the *Systems* table of GenMAPP.org's existing Gene Database template file (the *Misc* field contains the string "|I|" if a systems table is improper). "Proper" systems tables contain IDs that refer to individual genes or proteins as objects in the database and therefore can be used as IDs for genes/proteins on MAPPs and in Expression Datasets. "Improper" systems store annotations that can refer to one or many different genes or proteins. Pairwise relations tables between Proper systems, i.e., all possible Proper–Proper, are created, with repetitions removed (e.g., if *UniProt-EMBL* has been added to the relations tables list, then *EMBL-UniProt* will not be added). Pairwise relations between Proper and Improper, i.e., all possible Proper–Improper, are also added to the relations tables list. No Improper–Improper relations tables are added.

    For example, the *E. coli* systems tables are classified as follows: UniProt, EMBL, EchoBASE, EcoGene, Blattner, and PDB are proper, while InterPro, Pfam, and GeneOntology are improper.

11. System tables may be marked as "primary" in the export dialog. The set of primary system tables determines whether a relationship is *Direct* or *Inferred*. If a relations table includes a system table that has been marked as primary, then it is a Direct relations table. If neither system in the relation is marked primary, then the relations table is Inferred. The *Type* field of the *Relations* table holds this information for each relationship table.

    For example, with *E. coli*, UniProt is the sole primary systems table. Thus, direct relations can be made between UniProt and InterPro, Pfam, PDB, EMBL, EcoGene, EchoBASE, and Blattner. All combinations between EMBL, EcoGene, EchoBASE, and Blattner and between these systems and InterPro, Pfam, and PDB are thus inferred relations. We do not need to build indirect relations amongst InterPro and Pfam themselves, since these tables are classified by GenMAPP.org as Improper.

12. The template-provided *Relations* table is then filled with one record for each relations table in the exported Gene Database.

13. Gene Ontology-related tables are then created, based on the system table that has been designated as the MOD for the exported database. These tables are *GeneOntologyCount*, *GeneOntologyTree*, and *MODSystem-GOCount* (e.g., *UniProt-GOCount*, in the case where *UniProt* has been designated as the MOD system table).

14. The *OriginalRowCounts* table is created and populated; this table can be used to verify that the tables are complete. The information in this table should correspond to the number of records for the chosen organism that were originally imported from XML into the intermediate relational database.

15. *Species-specific post-processing.* The variety of data sources available for a particular organism occasionally necessitates export operations that are specific to that organism. These activities comprise the final step of the export process.

    For example, in the case of *E. coli*, the following steps need to be taken:

    - In a generic UniProt-centric Gene Database, an *OrderedLocusNames* table would be created. However, in *E. coli* this table is called "Blattner." Thus, for *E. coli* only, the *OrderedLocusNames* system name must be changed to *Blattner* in the *Systems* table.
    - Further, a historical artifact with Blattner identifiers results in the inclusion of multiple IDs in a single field in some cases (e.g., "b1964/b1965/b1966"). These IDs have to be separated into distinct records in the final Blattner table.
    - An *E. coli* Gene Database has UniProt as the MOD system table. There are more annotation fields than are in the Ensembl-centric Gene Databases provided by GenMAPP.org; as a result, the *Columns* field of the *Systems* table's *UniProt* record should be set to:

      `ID|EntryName\sBF|GeneName\sBF|ProteinName\BF|Function\BF|`

3 Database Libraries

The database libraries in XMLPipeDB are based, in turn, on the JAXB and Hibernate libraries. To use these libraries in other software, you will need the following in your Java classpath:

- the database library itself (UniProtDB, GoDB, etc.)
- Hibernate
- JAXB
- JDBC driver for your relational database

The libraries work like standard JAXB applications when importing XML files: the data in the loaded file gets converted into a set of corresponding objects. The libraries then work like standard Hibernate applications when saving these objects to the database: configure Hibernate, then use its classes to save/update the objects or to perform HQL queries.

To further assist in software development, you may want to look at XMLPipeDB Utilities (Section 5.2), which provide frequently-used routines and GUI components that are usually needed by applications using these libraries.

3.1 UniProtDB

UniProtDB is the library of Java classes that allows UniProt XML files to be transferred into a relational database. The GenMAPP Exporter uses the UniProtDB library; in turn, the UniProtDB library is semi-automatically generated from the XMLPipeDB project's XSD-to-DB tool.

3.2 GODB

Information from the GO (Gene Ontology) database is required to run GenMAPP Builder. GODB provides a library of Java classes and Hibernate mapping (HMB)files that allows data from GO XML files to be transfered into a relational database. The library is delivered and used as a Java Archive (jar) file called godb.jar. The classes and HBM files generated by xsd2db are processed by a utility called GodbPostProcessor prior to being built into the jar file. This is required because of some irregularities in the output provided by Hyperjaxb.

Using godb.jar The godb release includes a pre-built, ready-to-use copy of godb.jar. If the GO DTD hasn't changed, this may be used in GenMAPP Builder or in any other application you wish to build.

Creating godb.jar godb.jar only needs to be re-generated if the GO DTD has changed. However, in this case, the utility provide to do the post processing, GodbPostProcessor, may also need to be updated, since new issues may have been introduced with the GO schema change. In this case, please put in a support request on the XmlPipeDb project's support page (7).

Creating the godb.jar is a multi-step process. The steps are designed to be easy to perform, though.

1. Run xsd2db, providing the GO DTD URL as input 2. Run GO DB Post Processor on the output from step 1. 3. Run ANT using the build.xml in the root of GODB on the output from step 2.

To be written...

4 Database Administration and Setup

The applications in XMLPipeDB require any relational database for which a JDBC driver exists. Thus, they all require some degree of configuration, for specifying the database

server, database name, username, password, etc. To facilitate this, XMLPipeDB includes a common database configuration GUI in all of its applications.

5 Developer Tools

Many components in XMLPipeDB's end-user applications are actually built semi-automatically. The XMLPipeDB project includes these tools as well. End-users don't need to know about these tools; instead, they use the *output* generated by them. This section is for database designers and software developers who are interested in creating their own XML-compatible database applications.

5.1 XSD-to-DB

The XSD-to-DB application takes a well-formed XML Schema (XSD) file and converts it into a collection of Java source code and Hibernate mapping files that allows XML files based on that XSD to be read into a relational database. It was used to create the UniProtDB library which is used by GenMAPP Exporter.

XSD-to-DB's conversion functions are based on the open-source Hyperjaxb2 project (`https://hyperjaxb2.dev.java.net`). It requires the following information to do its work:

- URL for the XML Schema (XSD) file to convert
- Hyperjaxb2 binding file
- Directory that will contain its output

XSD-to-DB comes with a default binding file, which it copies into its output directory when it is invoked for the first time. Additional customization can then be performed on the copied binding file, and subsequent invocations of XSD-to-DB will use that binding instead of the default.

5.1.1 Usage

XSD-to-DB currently works as a command-line application, `xsd2db`. If you invoke it without any arguments, XSD-to-DB will ask for the URL of the XSD file to convert. Once the URL has been provided, XSD-to-DB then processes it produces Java source code, Hibernate mapping files, and an SQL DDL file that corresponds to the schema defined by the XSD file. These files are placed in an output directory, which defaults to `db-gen` if it isn't otherwise specified.

XSD-to-DB behaves a little differently depending on the presence of certain files in the output directory, or even the presence of the output directory itself. If the output directory is not present, then XSD-to-DB does the following:

1. Create the output directory.
2. Copy the XSD file from its given URL to the output directory.
3. Copy the default XSD-to-DB bindings file to the output directory.

If the output directory already exists, then XSD-to-DB looks for the XSD and bindings files at the expected locations within that directory. If those files are not present, then it performs the same steps as before.

- XSD-to-DB can be told whether or not to update the XSD file from its URL, in case the XSD file might have changed. If it is told to perform an update, it will ask for the URL of the updated XSD file.

Once the output directory has been set up, XSD-to-DB then processes the XSD and bindings files to generate:

- Java source code for classes represented in the XSD
- SQL DDL file defining the relational database tables that correspond to the Java classes
- Hibernate mapping files that determine how the Java classes are convert to and from the relational tables
- An Apache Ant `build.xml` file which can compile everything into a Java archive, ready for further development or deployment

From this point, a typical workflow would be:

1. Build a relational database using the generated SQL DDL file
2. Build the database library using the supplied Ant file, then use that library to test XML import, queries, and other database functions
3. Edit the bindings file to customize, correct, or improve the Java classes, Hibernate mappings, and relational tables generated from the XSD file
4. Re-run XSD-to-DB to actually create the new files

Certain conversions might not be adequate even with extensive editing of the bindings file; in this case, the last resort is to manually edit the Java, Hibernate, and SQL files generated by XSD-to-DB. If this is done, be careful about re-running XSD-to-DB on this particular output directory — XSD-to-DB *always* generates the Java, Hibernate, and SQL files "from scratch," using the XSD and bindings files in the output directory.

Typically, once the generated files work as desired, the output directory becomes a software project in and of itself, to be edited, debugged, tested, and deployed like any

other database library. The fact that it was initially generated by XSD-to-DB merely indicates that some time was saved in creating this database library as compared to a manual Java-to-relational implementation of the XSD file.

5.1.2 Output Details

XSD-to-DB's output directory structure is shown in Figure 4. Directories are shown in boldface, and files are shown in italics. Names in parentheses indicate placeholders for specific names that depend on the XSD being converted.

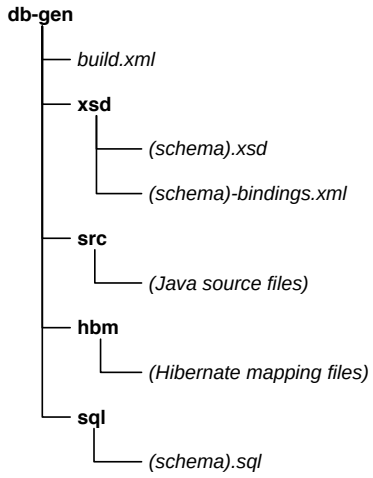

Figure 4: Structure of an XSD-to-DB output directory.

The actual filenames and contents of terms in angle brackets (<>) depend on the XSD file that was used to generate the output. For instance, the UniProtDB library has `uniprot.xsd`, `uniprot-bindings.xml`, and `uniprot.sql`.

5.1.3 Reference

The `xsd2db` command line application accepts the following arguments:

`--outputDirectory=`*`dir`* The directory to use when generating (or re-generating) the database source code and files; defaults to `db-gen`. Specifying a non-existent or empty directory essentially counts as a "first-time run" of XSD-to-DB.

`--xsdURL=`*`url`* The URL for the XSD to convert; required when XSD-to-DB is run for the first time, or when *updateXSD* (see below) is requested.

`--bindings=`*`filename`* The bindings file to use when generating the database source code and files for the first time; this file is then copied into the `xsd` subdirectory. Defaults to a standard bindings file supplied by XSD-to-DB.

`-updateXSD` Replaces the XSD being used with a new version; applicable only after XSD-to-DB has been run for the first time.

`-help` Displays a help message for how to use `xsd2db`.

5.2 XMLPipeDB Utilities

The XMLPipeDB Utilities library is a suite of Java classes that provide functions that are common to most XMLPipeDB database applications. Specifically, the library includes reusable classes for:

- Loading of XML files into Java objects
- Saving XML-derived Java objects to a relational database
- Rudimentary query and retrieval of Java objects from the relational database
- Configuring a client application to communicate with a relational database

XMLPipeDB Utilities includes a sample GUI application that demonstrates these functions, using database code that was generated by XSD-to-DB.

By "rudimentary query and retrieval," XMLPipeDB Utilities provides a facility for typing in an HQL (Hibernate Query Language) or an SQL (Structured Query Language) query coupled with an object browser that allows the user to examine the results of that query. This is meant primarily for debugging or advanced purposes.

The library is separated into two layers: one layer provides the functionality, meant to be called programmatically, and another layer is a set of user interface components that allow end-user to invoke those functions. Figure 5 provides an overview of the library.

5.2.1 Utility Functions

The following entities capture the functions provided by XMLPipeDB Utilities:

- `ImportEngine` provides a family of *loadToDB()* functions, which takes any compliant file or input stream, parses them into their corresponding objects, then commits the objects to the database.
- `Configurator` is the centralized location for configuration information used by XMLPipeDB. It provides functions for retrieving, setting, and validating configuration properties. The Configurator also has a method for obtaining a Hibernate Configuration object, which is needed by the ImportEngine and the QueryEngine. The configuration object, however, only contains the Hibernate properties and NOT the mapping files. The HBM files must be added by the caller before passing the Configuration object to the Query or Import Engines.

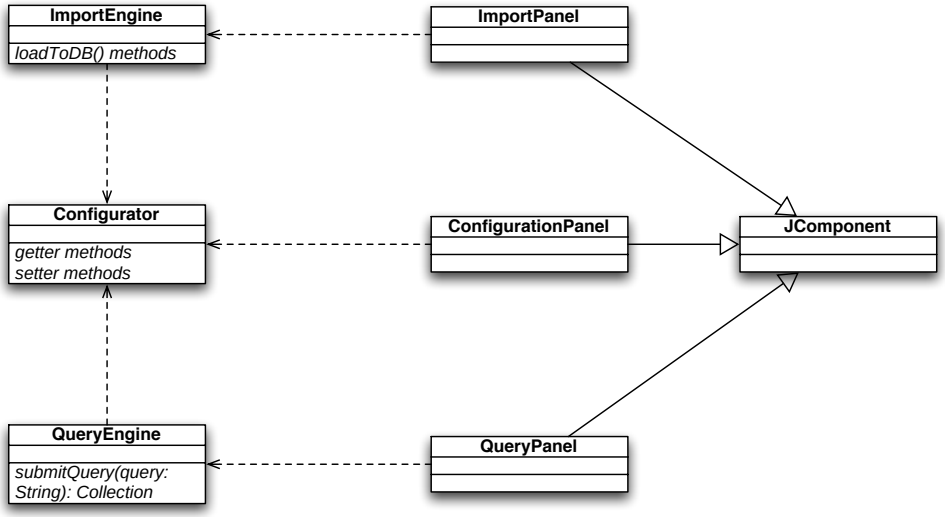

Figure 5: Overview of XMLPipeDB Utilities.

- `QueryEngine` is a generalized query wrapper whose *submitQuery()* function takes a Hibernate query (in HQL, Hibernate Query Language) or SQL and returns the results as a Java collection.

Programs using the XMLPipeDB Utilities library can invoke these functions as necessary, using whatever mechanism is most appropriate for a particular application.

5.2.2 User Interface Components

To ease the development and delivery of these functions to end-user applications, XMLPipeDB Utilities also includes a component library that can be added directly into Java Swing windows and panels. Each component provides a "hook function" that invokes the underlying operation, assuming that sufficient information has been gathered by the component:

- The `ImportPanel` family provides front-ends for database imports, displaying components like file choosers, text editors, and preview panels to make database loading as easy as possible for end-users.

- The `ConfigurationPanel` family of components provides front-ends for the properties in `Configurator`. The components are designed to be easily added to dialog boxes or preferences windows, and provide direct hooks to the `Configurator` functions.

- The `QueryPanel` components allow users to enter HQL queries for submission to `QueryEngine`. Correspondingly, a family of `ResultPanel` components provide reusable

displays for the collections returned by `QueryEngine`.

The demo application that comes with XMLPipeDB Utilities shows how these components are used, and how they interact with the underlying functionality.

6 Developer Documentation

6.1 Use Case Model

Figure 6 shows the use case model for the various components of the XMLPipeDB project. These use cases are the basis for the activities described in this manual.

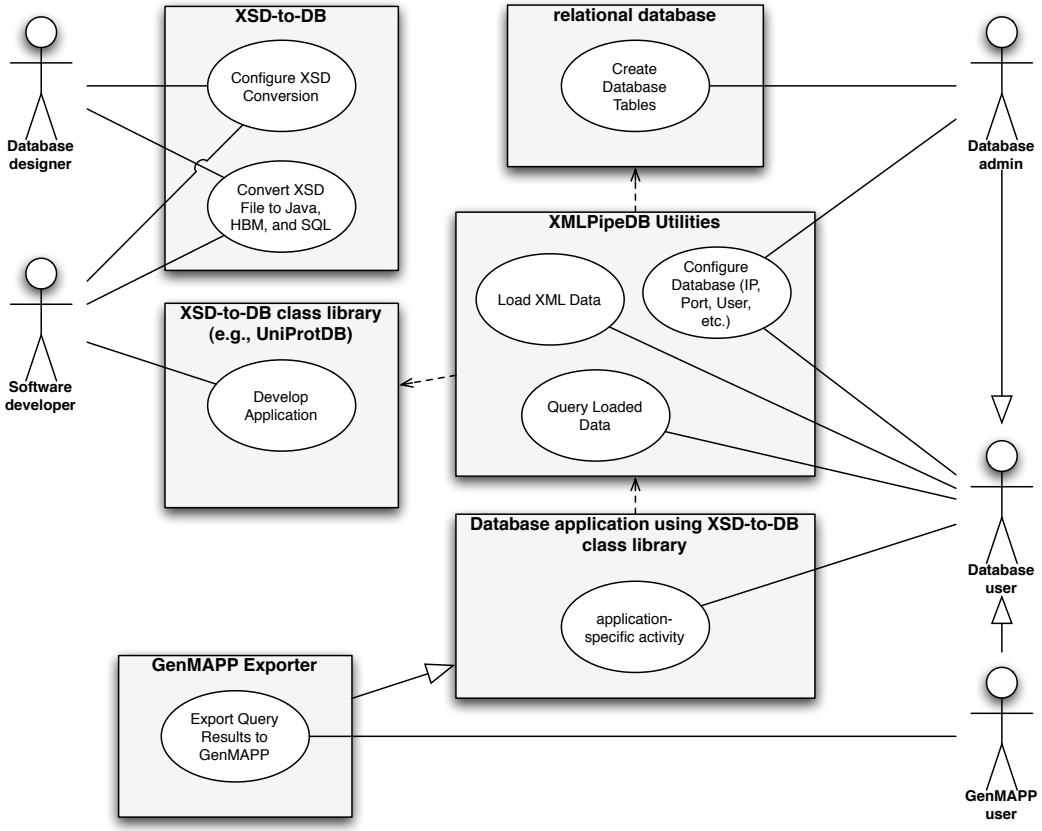

Figure 6: Use case model for the XMLPipeDB project.

6.2 Guide to the Repository

The XMLPipeDB project consists of a number of modules, listed hierarchically in Figure 7. They are related but intended to be standalone.

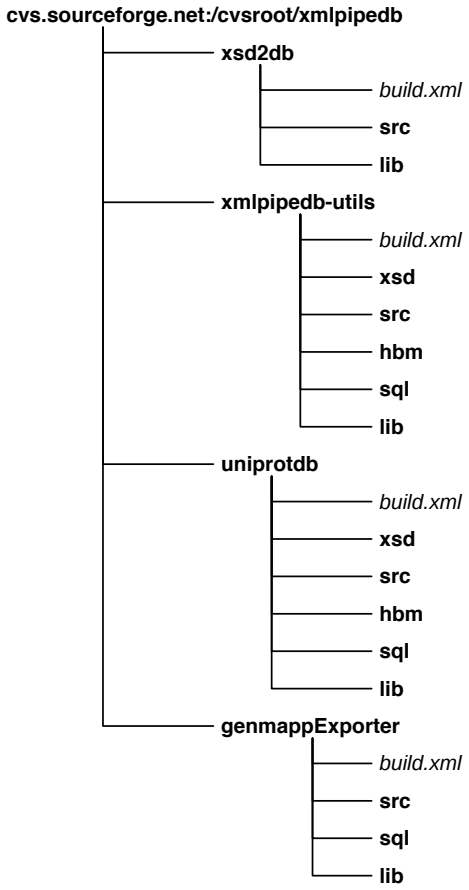

Figure 7: Structure of the XMLPipeDB CVS repository.

The `xsd2db` module contains the source code and libraries to XSD-to-DB. XMLPipeDB Utilities resides in the `xmlpipedb-utils` module, including the source for its demo application. When it is built, XMLPipeDB Utilities separates its reusable component (*xmlpipedb-utils.jar*) from the demo application that uses it (*xmlpipedb-utils-demo.jar*).

The sources for the UniProtDB library, which was generated by XSD-to-DB then manually customized, resides in the `uniprotdb` module. Finally, the source code to the GenMAPP Exporter application resides in the `genmappExporter` module. Each module contains an Apache Ant `build.xml` file that can compile the source into distributable binaries; these build products are placed in the `dist` subdirectory, and should not be committed to

the repository.

Similarly, output produced by XSD-to-DB (which defaults to `db-gen` in the current working directory at the time XSD-to-DB is invoked) should be committed as new modules in the repository, such as with `uniprotdb`. Of course, the default `db-gen` name should be changed to something more descriptive.

6.3 Building from Source

As mentioned, building fresh binaries of each XMLPipeDB module is a matter of downloading or checking out the source then invoking Apache Ant in each module's top-level directory. Build products are placed in the `dist` subdirectory, and should not be committed to the repository.

6.4 Unit Tests

We need them... 'nuff said!

6.5 Third-Party Libraries

XMLPipeDB "stands on the shoulders" of a wide variety of third-party libraries. These libraries are always stored in the `lib/` subdirectories of each XMLPipeDB module. Most of these third-party libraries are themselves active open source projects that continue to be developed; thus, to avoid confusion and accidental incompatibilities, all third-party libraries committed to XMLPipeDB are appended with a version number. For example, if an XMLPipeDB module uses Jakarta Commons Logging 1.0.4, then `-1.0.4` is appended to the standard library name, so that it is stored in `lib/` as `commons-logging-1.0.4.jar`.

If a third-party library is to be updated with a new version — and of course this new version has been verified to work with the existing code — then its prior version should be deleted and a new file added with the appropriate version suffix.

7 Support

Support can be obtained by going to xmlpipedb.sourceforge.net ??? ...and submitting a support request. As always, we appreciate your diligence in reading all the documentation, readme's and FAQ's before submitting a request for help. If the Uniprot XSD or GO DTD has changed and a change to the respective post processor is needed (or you suspect this to be the case), please include the version number of godb or uniprotdb that you have (this can be found in the readme file, which you would have known if you'd read it, so please go back and do this now). Since the project is staffed on an all volunteer basis, no guarantees are made about turn around on updates. However, since it is open source and the code is really, fairly straight forward, you could probably make the changes yourself

and simply run the ANT build.xml provided to re-generate the jar file. Of course, if you need an update urgently and can't make the changes yourself, a small gratuity might get our butts in gear (nudge, nudge). – your friendly neighborhood xpd team.