The Design and Implementation of Database-Access Middleware for Live Object-Oriented Programming

Adam H. Mitz and Kenneth J. Goldman
Washington University in St. Louis
{mitz, kjg}@cse.wustl.edu

Abstract

We describe middleware and programming environment tools (JPie/qt) that allow programmers to access relational databases in an object-oriented way. Building on top of the JDBC API and leveraging live dynamic class creation and modification in JPie, the JPie/qt middleware presents the user with a simple interactive mechanism for creating object-oriented applications that access databases. Classes are generated mirroring the database schema and programmers deal directly with these classes. Objects of these classes can be database-bound, so reads and writes to their fields are reflected in the relational database immediately. Database transactions are supported by connecting commit and rollback to Java exception semantics.

1. Introduction

Object-oriented programming is intended to increase reusability and maintainability [11]. Programmers concentrate on data abstraction, encapsulation, inheritance, and polymorphism. However, many real-world programs require communicating with a relational database to ensure that data persists between program invocations, to communicate with an established corporate database, or to use as a communications channel for distributed systems. Using these databases means mixing two different programming models, and essentially two programming languages, within a single application. This added complexity makes programs harder to construct and reason about. The existing relational database APIs for most object-oriented languages (such as JDBC in Java) provide a bridge from the object-oriented model to the relational database model. However, they essentially provide a means to forward queries written in a relational database language (e.g. SQL) and therefore require extensive knowledge and use of the relational database model. Consequently, programmers must think about their applications as artificially split into heap objects (which are accessed using an object-oriented language) and database tuples (which are accessed using a relational database language). This paper presents a middleware-based approach to insulating the object-oriented programmer from the relational query language and model.

1.1. Design goals

We present a middleware framework called JPie/qt (JPie queries and transactions), that bridges the object-oriented and relational models in order to provide programmers with object-oriented access to data stored in a relational database management system. JPie/qt exposes an object-oriented API to programmers and internally uses Java's JDBC [9] system to communicate with the relational database. The specific design goals for JPie/qt are described below.

1.1.1 Programming model unification. JPie/qt's major design goal is unifying the programming model. The programmer needs no knowledge of SQL and its associated programming model (data types, semantics, etc). Using JPie/qt requires only that the programmer be familiar with basic object-oriented concepts such as user-defined types, data encapsulation, iteration over collections, exceptions, and static methods. Another aspect of this unification is the naming of classes and fields. JPie/qt classes and fields are named corresponding to the application-specific table and column names in the database. Thus the names are directly relevant to the programmer. Additionally, the concepts of failed execution in the two models, exceptions and transaction aborts, are unified in JPie/qt.

1.1.2 Flexibility. JPie/qt is designed to work with any relational database for which a JDBC driver is available. This offers the programmer flexibility in designing applications. The programmer is not bound to a certain database vendor. Additionally, the database schema is assumed to be outside the programmer's control. The system is designed to work with databases that already exist or are set up by a database administrator who is unaware of JPie/qt.

1.1.3. Efficiency. Run-time performance of applications

```
String query = "SELECT P.Name,
     R.Name FROM Player P, Room R
     WHERE P.Location = R.ID";
ResultSet rs =
     conn.createStatement()
     .executeQuery(query);
while(rs.next()){
     printOut( rs.getString(1),
                    rs.getString(2)
     );
}
rs.close();
```

Figure 1. Pseudocode for JDBC example

developed with JPie/qt should be acceptable for development purposes. To this end, JPie/qt makes use of lazy instantiation and caching in order to minimize communication between the JPie/qt and JDBC layers.

JPie/qt represents database rows as Java objects, but these are initialized lazily. When an object corresponding to a database row is instantiated, only the primary key field is set to contain data matching that in the database. None of the other fields' data is fetched eagerly. Thus, programs only incur the costs of fetching data that they need.

1.1.4. Host language integration. JPie/qt's host environment is JPie (see Section 3). In order to provide the programmer with an easy-to-use system, one of the goals of the JPie/qt design is tight integration with the JPie environment. To achieve this goal, the JPie/qt system is designed to offer a GUI look-and-feel and semantics consistent with JPie.

1.2. A motivating example

Consider a programmer working on a multiplayer online game. The system uses a centralized relational database to store the game state so that all players can see the shared state and so that the state persists between player sessions. To support a "scoreboard" feature that summarizes the game, we require a function called *listPlayerLocations* that reads each player's name from the database and lists it, along with the name of the room the player is currently in.

Let's begin by considering how one would construct this method in JPie using JDBC directly, instead of JPie/qt. Figure 1 reveals a surprising amount of incidental complexity, sometimes called "accidental complexity" [12], which the programmer must deal with in constructing a correct JDBC program. First, and of utmost concern, is the requirement that the programmer express the data query in the SQL textual language. The programmer must also know the relational schema and its table relationships. As with any embedded textual

```
for( Player p :
     Player.getAllRecords() ){
     printOut(p.getName(),
       p.getLocationAsRoom().getName(
     ));
}
```

Figure 2. Pseudocode for JPie/qt example

language, no compile-time syntax or type checking can be done to validate the SQL strings. Secondly, the program is composed almost entirely of operations on objects of JDBC types such as *java.sql.Connection*, *java.sql.Statement*, and *java.sql.ResultSet*. All three of these types are needed to construct even the simplest JDBC operation. Their use is unintuitive and they are obfuscated by an overabundance of methods in their public interfaces (especially ResultSet which deals with iterating, inserting, updating, and retrieving data).

Figure 2 shows a method that is equivalent to the one in Figure 1, but that uses the JPie/qt middleware for database access. Use of JPie/qt has eliminated the incidental complexities of both SQL syntax and the often rigid associated API (such as JDBC). The program is written in terms of the user-domain classes *database.Player* and *database.Room*, and logically named (and automatically created) methods on them: *getAllRecords*, *getName*, and *getLocationAsRoom*.

1.4. Contributions

JPie/qt brings the power of a relational database management system to a novice programmer or application-domain expert familiar with object-oriented programming abstractions. JPie/qt eliminates many of the accidental complexities encountered when writing object-oriented programs that interface with relational databases. The programmer's productivity is increased because he or she interacts with rows of database tables directly as objects on the Java heap.

2. Related work

Due to the prevalence of both the object-oriented programming model and the relational database model, seamless integration of the two is an important research goal that has resulted in a number of technologies. Most notably, the problem of better integrating data-access features into the Java programming environment has been addressed by Sun in JDO [13] and by various third-party vendors in OO-Relational mapping tools [1]. In addition, related work in the patterns literature [2], [6], [12], [13] has some bearing on this problem. We discuss each of these in turn.

2.1. JDO

Java Data Objects (JDO) [13] is an API that allows Java-domain objects to be stored in a database. JDO users can make instances of a class storable in a database by changing the class to implement the PersistanceCapable interface. (Unlike java.io.Serializable which serves a similar purpose, PersistanceCapable is not an empty or "marker" interface. There are a many methods that need to be implemented; hence tools are used to automate this process.) Since this may require far-reaching changes to source code, a shortcut is provided by bytecode enhancer utilities that modify Java class files directly. To retrieve persistent objects, the JDO Query API is used with filters specified as strings in the JDOQL language. JDOQL is a textual query language (using Java-like syntax) that serves the same purpose as SQL WHERE clauses.

JDO is flexible with regard to what backing store system is used to store the objects. One commonly used type of backing store is a relational database. When used this way, JDO achieves goals similar to those of JPie/qt. There are two major differences, however. First, JPie/qt leverages JPie's features to become very transparent to the programmer. Therefore a separate textual query language or object-oriented query API doesn't need to be learned. Additionally, JPie/qt is designed to work with legacy databases in that the object-oriented schema is created from the database instead of vice versa.

2.2. OO-relational mapping tools

Software packages such as IBM's Rational Rose XDE [1] provide professional programmers working in textual languages with many of the same features that JPie/qt provides to JPie programmers. In general, these tools work by allowing the user to construct a model of the data in a vendor-neutral format (UML, for example) and then generating the relational tables and code stubs from this model. The programmer then customizes the code and the tool keeps the model in synch. In terms of both expense and complexity these tools are typically out of reach for the beginning or casual programmer. The main focus of these systems is to assist developers with experience and knowledge of both the relational and object models. This is in contrast to JPie/qt, which seeks to abstract away much of the relational model.

2.3. Patterns

Architectural and design patterns can be found in any well designed system and JPie/qt is no exception. Throughout this paper some instances of patterns are noted in references. A few of the major patterns affecting the system as a whole are described below.

2.3.1. Wrapper Facade. Wrapper Facade [12] is a pattern that builds object-oriented APIs from procedural operating-system level APIs. JPie/qt itself can be seen as a Wrapper Facade around the relational database model and JDBC. Wrapper Facades for OS APIs reduce the incidental complexities of using those APIs and JPie/qt does the same for relational databases and JDBC. Traditional Wrapper Facades wrap OS concepts such as sockets, threads, and locks and elevate them from untyped handles or pointers to classes. JPie/qt on the other hand is not a set of classes but a system that generates classes (and a supporting class library). Thus, JPie/qt is able to present the programmer with domain-specific abstractions that encapsulate the relational database-access logic.

2.3.2. Half-Object Plus Protocol. Half-Object Plus Protocol [2] is a pattern for distributed objects. Each address-space contains "half" of an object, plus the logic to communicate with its counterparts in other address-spaces. In JPie/qt, database-bound objects (see Section 5.5) act like half-objects. They are objects of classes that may have a number of fields, but the contents of some fields are ignored. Instead accesses to the fields are redirected to the JPie/qt system and on to the underlying database. This simplified the design of JPie/qt. By using this design the user's interface to JPie/qt is primarily through the classes with which they are familiar.

Half-Object Plus Protocol can be seen in another JPie system known as CDE [10], a Client Development Environment for distributed applications.

2.3.3. Crossing Chasms. The topic of object-relational integration is discussed in Crossing Chasms: A Pattern Language for Object-RDMBS Integration [14]. This work lists patterns such as "Representing Objects as Tables," "Object Identifier," and "Foreign Key Reference." These patterns document a straightforward approach to representing relational tables in an object-oriented system.

Representing Objects as Tables describes the analogy between classes in the object-oriented model and tables in the relational model. Object Identifier notes that classes don't necessarily have fields whose values are unique across all objects. To mesh with the relational model, each class needs a field that serves as its unique identifier, also known as a primary key. Foreign Key Reference describes how the object reference graph is represented in the database. It is used in concert with Object Identifier, since object references are represented as foreign key fields in the database. Foreign key fields are those that contain copies of the primary key of another table.

JPie/qt uses all three of the patterns mentioned above to map the object-oriented model to the relational database model. JPie/qt adds additional features such as Primary Key Reference (following a Foreign Key Reference in the opposite direction to obtain a collection of referents) and

Table Browsing (scrolling through all existing instances).

3. The JPie environment

JPie/qt is an extension of JPie, a tightly-integrated development environment for live construction of Java applications [7]. JPie enables inexperienced programmers to create Java programs through direct manipulation of program constructs, using the standard Java platform APIs. JPie focuses on the process of software design and creation as an experience that can be made simpler and more intuitive for programmers. JPie/qt extends JPie's progress towards these goals in the area of database-connected programs. JPie also provides on-the-fly class creation and modification mechanisms that are used to implement JPie/qt.

3.1. Programming in Java with JPie

In JPie, fully functional Java programs are created without entering code as text. Instead of encoding programs textually, users manipulate graphical components directly representing the programming abstractions. JPie users have access to the classes in the Java 2 Platform SDK (J2SE) as well as third party classes. Classes created within JPie are known as "dynamic classes" because they can be modified during program execution, providing a live development experience [8].

The JPie user can define the fields, methods, and constructors of the class. Additionally, support is given

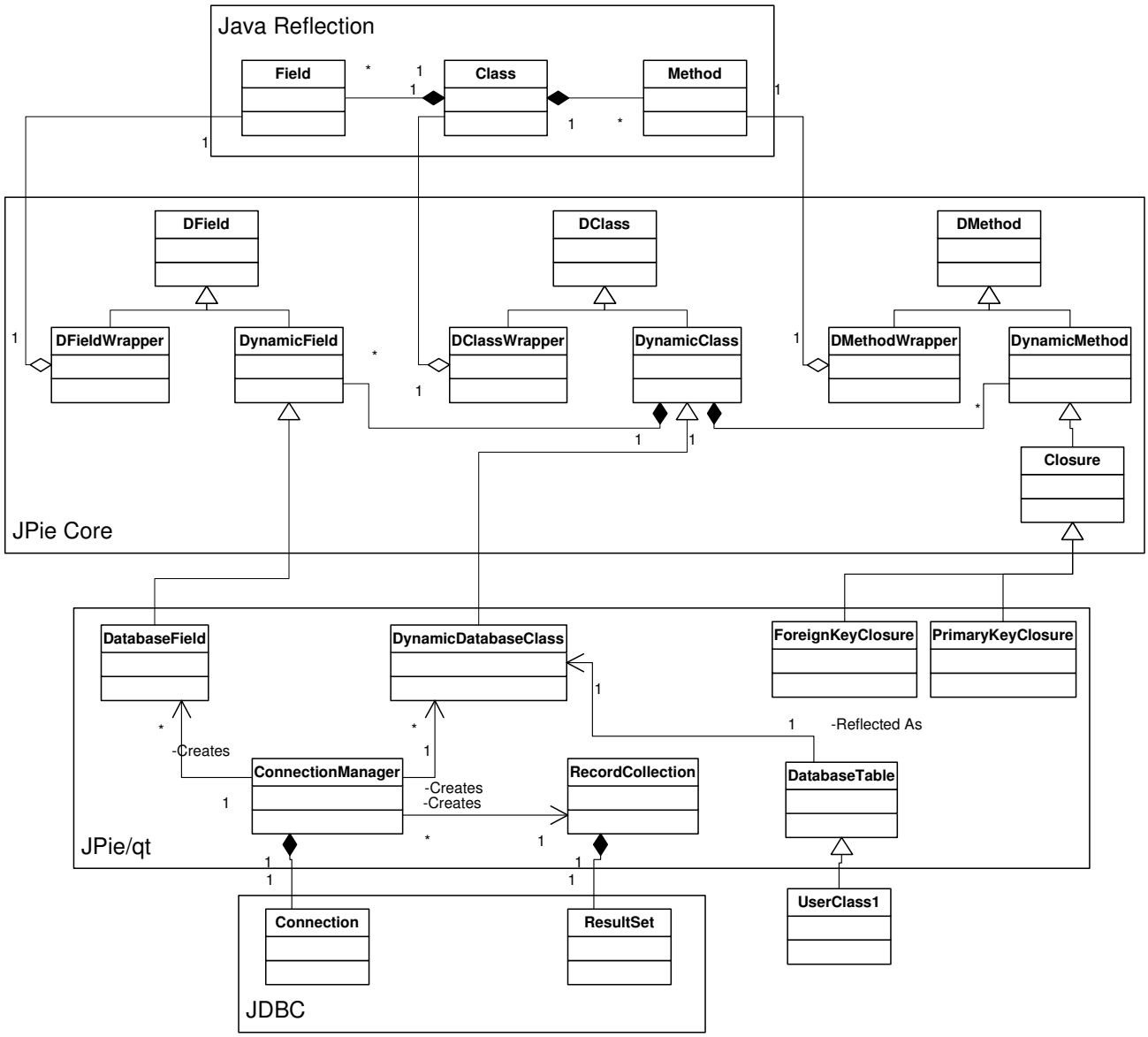

Figure 3. JPie/qt UML static structure

for designing a graphical view for the class, creating event handlers that react to events on the view, and defining behaviors (methods that execute periodically in their own threads). Java statements are created within method bodies inside method, constructor, event handler, or behavior definitions.

3.2. Executing and debugging dynamic classes

JPie users can see a list of all instances of a class that have been created. Instances can be created by invoking a menu command (for classes with a default constructor). When an instance is selected in the list, its graphical view is shown. Should an exception occur in a dynamic class method, a debugger appears. The debugger shows the same graphical view of methods as is used in method definition. Methods can be changed in the debugger or the dynamic class window and all changes are reflected in already existing objects. By taking advantage of live code modification, developers can avoid the edit-recompile-execute cycle when debugging their programs.

3.3. JPie internals

To support dynamic classes, JPie extends the Java reflection API's capabilities to allow modifying classes, fields, methods, and constructors. An intuitive way to structure the API would be to inherit from the Java reflection classes like Class and Field. Unfortunately these are final classes, so instead the JPie reflection hierarchy wraps them in the wrapper classes (Figure 3) ClassWrapper, FieldWrapper, MethodWrapper and ConstructorWrapper. The abstract classes DClass, DMember, DField, and DConstructor represent the interfaces common to both compiled and dynamic classes.

Extending the reflection system, however, isn't enough to make dynamic classes fully interoperable with compiled classes. To achieve this, each dynamic class has a compiled peer. The compiled peer is a traditional Java class with the same name and inheritance ancestors as the dynamic class. In addition, all compiled peers implement a special interface, DInstance, used by the JPie system. Instances of the compiled peer class (known as peer instances), act as proxies [6] for the dynamic class instances in the JVM. Using this mechanism, compiled code (such as the standard Java APIs) can be called from dynamic classes and issue callbacks (Swing events, for instance) to the dynamic class instances.

4. Using JPie/qt

This section illustrates the programming experience supported by JPie/qt, concentrating on how programmers interact with a database while constructing an object-oriented application.

4.1. Example program: SmartMail

SmartMail is a system for automating multicast email requests and managing responses. For example, a survey may be sent to many people. SmartMail would be used to collect and tabulate the replies instead of the sender doing so manually. SmartMail is being developed by our research group in order to test and evaluate JPie (including JPie/qt) as a development environment for medium-sized software applications.

SmartMail messages originate in the author's standard email client. An interceptor component acts as an SMTP proxy and begins the process of turning a regular message into SmartMail. The SmartMail contains HTML form fields for recipients to fill in. Before being sent, it is handed off to a *register* component which records information about the SmartMail in a relational database. The SmartMail is then sent to each recipient as an HTML message with an embedded form.

Recipients read SmartMail with any standard HTML-enabled email client. Once they have filled out the form fields and press the "Submit" button, the form contents are sent to the *accumulator* component via HTTP. The accumulator tabulates the results in the relational database. The originating author can then query the database for the results.

The accumulator component's design and implementation are described in this section as an example of a JPie/qt client application. A Java Servlet written entirely in JPie, the accumulator is responsible for receiving replies and storing them in the central database.

4.2. Initial setup: database and JPie/qt

For the SmartMail prototype, we chose the free database MySQL [4] as the relational database engine. We designed a schema (Figure 4) for the SmartMail system and using the MySQL tools. Arrows in the diagram indicate N-to-one relationships.

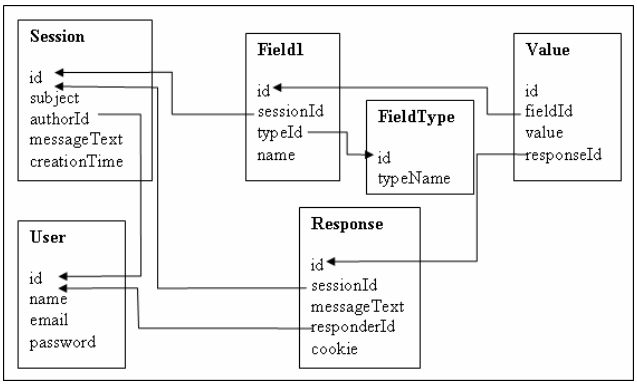

Figure 4. SmartMail database schema

4.2.1. Opening the database in JPie. With the schema in place, JPie's "Open Database" command is executed. This brings up a wizard that prompts the user for the JDBC connection string, user name, and password, followed by a selection box listing the available database tables. Selecting all tables causes each one to be brought in to JPie. Next, a folder in the user's classpath is chosen to store the database-connected classes. When the operation completes, JPie makes available the classes corresponding to each table.

4.2.2. Establishing table relationships. When JPie/qt generates the database-connected classes, it includes both declarations for the fields mirroring the fields of the database table and accessor/mutator methods (get and set). What's missing is a way to deal with inter-table relationships. Since JDBC doesn't provide a widely-supported method to retrieve this information from the database itself (some databases do not even keep track of table relationships), the user needs to indicate these relationships to JPie/qt. To do so, the programmer drags the related type onto the "links to" area of the field declaration for the foreign key field. In the example of Figure 4, to associate the Session table's field "authorId" with the User table, the programmer drags the User class to the "links to" area of the declaration of authorId inside the Session class.

4.3. Developing the example program

Only one class is needed to implement the accumulator component. The Accumulator class extends the J2EE provided class javax.servlet.http.HttpServlet. The Servlet methods init() and destroy() are called at the beginning and end of the object's lifetime, respectively. The doPost() method is called to process each HTTP request (we're using the POST request verb).

The algorithm for doPost() is simple. First we must look up the response object from the database that corresponds to this response. A cookie (Also known as an "Asynchronous Completion Token") [12] is used to uniquely identify responses (it is embedded in a hidden form field in the HTML). This ensures that the response is valid. Next we need to iterate over the Fields (from the Field1 table -- so named because "Field" caused naming conflicts with JPie and/or java.lang.reflect.Field) for this session and create corresponding Values for this response.

This operation comprises multiple updates to the database. In order for it to appear to be one atomic change, we use the transaction feature of JPie/qt. JPie/qt provides a method modifier named "transaction" right alongside "synchronized," "static," and "final." Just as "synchronized" prevents multiple threads from entering a critical section, "transaction" protects multiple threads or processes (possibly outside of JPie) from race conditions or seeing incomplete data in the database.

5. Implementation

The Java classes that make up the implementation of JPie/qt connect the core JPie extended reflection system (see Section 3.3) to JDBC. This is shown in Figure 3.

5.1. Metadata inspection and dynamic class generation

The user begins the first session with the JPie/qt API by invoking the "Open Database" menu command, as described in Section 4.2.1. Internally, the system then creates a dynamic class for each selected table. These dynamic classes are each represented in the JPie system with objects of the class DynamicDatabaseClass. JPie/qt adds fields to the class corresponding to the columns of the table. These fields are DatabaseField objects in JPie. JDBC metadata and type mapping features are used to retrieve the table names and types. Metadata inspection and dynamic class generation is not repeated in subsequent sessions; the user need only open or refer to the generated dynamic classes.

5.1.1. The primary key. When the system needs to know which column holds the primary key for a table, it uses the following algorithm. First the metadata is queried through JDBC. However, not all JDBC drivers support this method. If the metadata key information is unavailable, the first integer column is considered the primary key. This is a reasonable heuristic because conventional database design positions the primary key first. Clustered primary keys (those that span multiple fields) are not currently supported.

5.1.2. Related tables. After the dynamic classes are created and populated with fields, they appear in JPie with all of the other dynamic classes. Database-generated classes, however, have augmented views in JPie. The field declaration view has an additional area called "links to." A class name (another dynamic database class) can be dragged into this area. This establishes a relationship between the two tables with the field as the foreign key. Note that the foreign key field still has a primitive type, but it also has a related type (another dynamic database class). When a relationship between tables is established, two methods are generated, with natural application-specific method names built from the table and field names in the database. These two methods allow the programmer to navigate the N-to-1 table relationship using the Java objects.

One is a method on the foreign key's table class that returns the corresponding object of the related table class. Object-oriented programmers will find this to be a natural

operation, since it mirrors following an object reference. This is known internally as the Foreign Key Closure (since it is implemented as a closure in the JPie system, as explained in Section 5.1.3). An example of such a closure is "getAuthorIdAsUser" in the Session class, generated from the relationship between the Session and User tables in Figure 4.

The other is a method on the related table that returns a set of objects (using java.util.Collection) representing the set of rows in the foreign key table related to the current object (this) in the related table. This is the Primary Key Closure. An example of such a closure is "getSessionSetForAuthorID" in the User class, generated from the relationship between the Session and User tables in Figure 4. This closure can be thought of as the dual of "getAuthorIdAsUser."

5.1.3. Classes in the JPie/qt implementation. Instances of the ConnectionManager class wrap JDBC Connection objects. This class provides the main point of interface between JPie/qt and JDBC. Along with the wrapped JDBC Connection object, the ConnectionManager instance stores connection-specific data such as the URL, username, and password, as well as the per-thread transaction information. Because the other JPie/qt classes use the ConnectionManager to perform their functions, a convenient way to access the ConnectionManager objects is needed. To this end, a static hash map from Strings to ConnectionManagers is kept, where the strings are the fully-qualified class names of the dynamic database classes.

So that the classes corresponding to database tables can be dynamically modified in JPie, they are represented as instances of DynamicDatabaseClass, which extends JPie's DynamicClass. Instances of DynamicDatabaseClass represent classes whose objects have the ability to be database-bound. This class's sole responsibility is to distinguish DynamicDatabaseClasses from other DynamicClasses so that its fields are DatabaseFields (Section 5.2.1), and the correct GUI views are created by JPie. (As of this writing, the JPie and JPie/qt implementations are being changed so that any DynamicClass can be a database class. This will permit dynamic changes to the database status of any DynamicClass.)

Each class generated by JPie/qt extends DatabaseTable. Therefore instances of DatabaseTable represent the rows of a given table. This is where per-object information is stored, such as whether or not the object is currently database-bound. DatabaseTable also provides methods that can be called on any JPie/qt object, such as insert. The insert method, when called on an unbound object, inserts a row in the database with the field values of the object, and binds the object to the row.

Closure is a JPie class that extends DynamicMethod. The subclasses of Closure are regular compiled Java classes that override an "invoke" method. Closures are added to DynamicClasses and appear in the list of declared methods for that class. When the closure methods are executed in dynamic classes, the compiled "invoke" method is called. Closures are used here because the system needs a way to create methods at run-time that execute boilerplate code that is parameterized by (and bound up with) precomputed run-time data, such as which tables and fields to operate on.

ForeignKeyClosure and PrimaryKeyClosure each extend Closure to allow traversal of database relationships. ForeignKeyClosures, such as getAuthorIdAsUser (see Section 5.1.2), traverse the relationship from foreign key to primary key. For example, calling the closure method on a given Session object results in the related User object being returned. This is implemented by calling the getInstance method on the related table (User), passing in the value of the foreign key. PrimaryKeyClosures, such as getSessionSetForAuthorId (see Section 5.1.2), traverse the relationship in the opposite direction. Given a User object, a set of Session objects are related by the AuthorId field in the Session table. When the PrimaryKeyClosure returns a set as an object implementing java.util.Collection is returned, the RecordCollection class (Section 5.3.3) is used internally to implement this functionality.

5.2. Field access and update

Fields belonging to dynamic classes that are generated to mirror the database schema have overridden behavior for both reading and writing, so that these operations affect the appropriate data in the database. Therefore all expressions using or assigning these fields (not just accessor and mutator methods) will result in underlying calls to the database. For either a read or a write of a specific value, a JDBC ResultSet object is generated. The ResultSet will have exactly one row because only the row corresponding to "this" object is queried. This ResultSet is cached for future use and also used to complete the requested operation.

5.2.1. Class in the JPie/qt implementation: DatabaseField. DatabaseField extends JPie's DynamicField class. DatabaseFields add a field for the related class. Besides tracking table relationships, the purpose of DatabaseField is to override get and set so that access to fields of database-bound objects results in reading or writing data in the database.

5.3. Table iteration

User programs need to iterate through the objects representing the rows in a table. JPie/qt provides the programmer with ways to iterate through all of the rows of

the table (complete iteration) or through a subset of rows (filtered iteration).

5.3.1. Complete iteration. Each dynamic database class is generated with a static method "getAllRecords()." This method returns a java.util.Collection to the caller that represents all objects in the table. Calling this method causes one query to be run against the database (internally creating a JDBC ResultSet as the backing store for the collection) to get the primary key values for each row. These keys are not loaded into the system at once. Instead the iterator class for this collection steps through the ResultSet, getting one key at a time and returning a reference to the corresponding Java object.

5.3.2. Filtered iteration. Iterating over a complete table may be undesirable when only a limited number of rows are really needed and the table extent is large. Therefore a "getSomeRecords(String where)" static method is provided in each generated class. This method allows the user to input a textual WHERE clause to be passed directly to the JDBC layer in order to select only certain rows to be iterated over. This allows programmers with some knowledge of SQL to apply that knowledge to JPie/qt. This is the part of JPie/qt that, for efficiency reasons, exposes SQL to the programmer. However, programmers without knowledge of SQL syntax do not need to use this mechanism. They can use complete iteration combined with "if" statements to achieve the same results, albeit with a decrease in performance.

5.3.3. RecordCollection. The RecordCollection class implements java.util.Collection and represents a set of rows from a database table. RecordCollection objects can be constructed in three ways to support sets representing all rows in a table, a subset of rows using the "where" clause, and a subset of rows using the primary key traversal. In each of these cases, an SQL statement is constructed by the middleware and run against the database. The ResultSet from JDBC is used internally by the iterator. The iterator's hasNext method maps to the ResultSet's isLast method (negated). The iterator's next method maps to ResultSet's next method. The ConnectionManager iterator translates the current database row into objects to be returned.

5.4. Transactions, Abort, and Exceptions

Methods in JPie have an optional extra modifier (like "final" or "static") called "transaction." The normal transaction mode in JDBC is known as auto-commit. Each operation is committed to the database upon completion. Any method defined as a transaction will cause database operations to not be auto-committed (when the JDBC driver supports this). Instead, the commit will be done when the method exits by returning. The JPie/qt design further unifies the transaction and method execution semantics by causing a database abort (rollback) to occur if the method terminates by throwing an exception. Commit-mode information is stored per-thread since different threads may be executing different database operations, some in transactions and some not. (This is an example of thread-specific storage [12].)

Transactions have nesting semantics, much like recursive locks. If a thread is in the course of executing a transaction method and it calls another transaction method, the transaction doesn't commit when the nested method exits. Instead, a save-point is defined when the nested method begins. If the nested method exits by throwing, a partial rollback (to this save-point) is done. If the nested method exits without throwing, execution continues as normal in the caller.

5.5. Browsing tables

Within the context of an executing program, instances of DynamicDatabaseClass classes are in one of two states: bound or unbound. Bound instances have a corresponding row in the database table. Reads or writes to fields of these instances have direct and immediate effect on the database. Unbound instances reside on the Java heap with no connection to the database. The binding state is kept in a Boolean instance variable by the system.

Similarly, each row of the database table is also in one

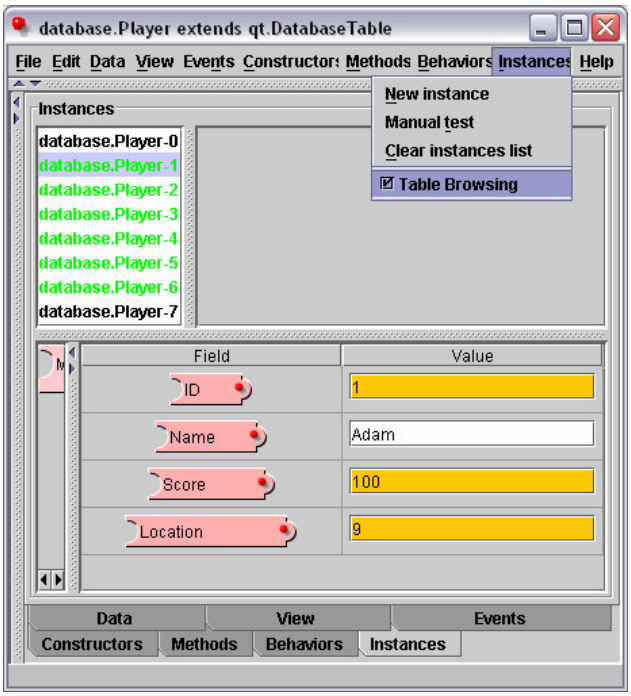

Figure 5. Highlighted instances and table browsing

of two states with respect to the JPie/qt system: resident or nonresident. Resident rows have a corresponding Java heap object (which is a bound instance) whereas nonresident rows do not.

In JPie, users view and interact with instances of dynamic classes in the instances panel. The instances panel consists of a list of instances on the left, and a panel for the graphical view of the currently selected instance on the right. JPie/qt enhances the instances list in two ways. First, bound instances are highlighted in green. Second, table browsing can be enabled. Table browsing adds an entry to the instance list corresponding to each row in the database table. See Figure 5.

6. Performance

JPie/qt adds a layer between the programmer's code and JDBC. Therefore it is expected that programs using JPie/qt will have somewhat greater memory and execution time requirements. To test the worst-case execution time overhead, a database table was filled with 25,000 rows, each containing a random integer. Procedures were written to compute the sum of the 25,000 integers using four different methods: JPie/qt iteration, JDBC iteration in JPie, JDBC iteration in Java and an SQL aggregate function ("SELECT SUM(X)…"). The first three procedures brought the data into the program's address space in order to compute the sum, whereas the last procedure instructed the database to compute the sum on behalf of the program. The platform used for these tests was an IBM Thinkpad T40 with the Intel Pentium M processor at 1400 MHz and 512 MB of RAM. The database used was a local instance of MySQL [4]. The JVM running JPie was started with an initial heap size of 1 billion bytes. Results from averaging five runs are shown in Table 1.

Table 1. Execution time for 25,000 records (ms)

JPie/qt	JPie/JDBC	JDBC	SQL Aggregate
10,100	2,490	376	68.2

The experiment shows that within the current implementation of JPie/qt, the approximate time to fetch a record is 0.4 milliseconds, which is acceptable for development work but not for sizable batch processing. We observe an order of magnitude cost increase using JPie/qt over JDBC for batch-processing operations. This can be attributed to the fact that JPie and JPie/qt create an instance of a dynamic class for each row. To provide interoperability between compiled and dynamic classes, the JPie run-time system allocates two "peer" objects to represent each instance of a dynamic class and uses synchronized method calls (with the related locking overhead) to make the two peers mutually referential. In addition, JPie/qt incurs overhead for cache management. Exporting completed JPie/qt programs as standard Java applications, as discussed in Section 8, could alleviate much of this overhead and still provide the benefits of live development and model unification.

7. User Experience

The students of Washington University CS123 in the Spring 2003 semester completed a project using an early version of JPie/qt. The project involved constructing software that ran on multiple hosts, each connected to a central database. The database described the state of a multiplayer adventure game. Tables included Player, Room, ItemDescription, and ItemPlacement. The students implemented methods that displayed the current state and allowed players to move through the virtual world.

This experience didn't uncover any performance problems, in contrast to the experiments in Section 6. The discrepancy between the two scenarios can be explained by the differences between the two applications. While the experiments in Section 6 were data-intensive batch-processing tasks, the CS123 project was interactive. At each button click (a request to move rooms, pick up an item, or "tag" another player) the application has a very limited amount of database-related work to do.

JPie/qt is suitable for development and interactive applications, but in its current form it has a high performance overhead for batch-processing applications.

8. Future work

Several additional features are being considered for inclusion in the JPie/qt system. These features would extend the system's performance and its scope of applicability.

Programs using JPie/qt currently must run within the JPie environment. Code generation features are available in JPie to automatically generate and compile Java source code from JPie applications. However, these features currently do not support JPie/qt programs. To allow exported programs to use JPie/qt, parts of the JPie/qt system could be factored out as a runtime library. Then, during code generation, accesses of database fields would result in calls into this library. This runtime library and the JDBC driver code would be combined with the programmer's exported code to form a complete program.

Even with code generation, performance could suffer when computations that the database engine could do are instead carried out in the user process. Unfortunately, JPie/qt programs can easily suffer from this problem. As a simple example, a user might call getAllRecords() to iterate over a table and then within the loop compare a field of each row to a constant value. By doing this the user has inefficiently implemented the "select" operation

of the relational algebra [5], instead of relying on the relational database. Programs executing in the JPie environment are not optimized, allowing for live debugging and modification. As a future extension, code generated from JPie/qt programs could be, when possible, converted to SQL queries.

JPie/qt maps schemas from databases to class and object relationships in and object-oriented program. The converse operation is also useful in many cases. Sometimes an object-oriented design describes objects that the programmer wishes to make persistent in the database. Work is underway to enable JPie/qt to transform any DynamicClass into a database-connected class, and in so doing create a corresponding table in the relational database. Additionally, design changes made to the dynamic class (adding, removing, renaming fields) will be reflected in the database schema.

JPie/qt allows programmers to pass a String argument to getSomeRecords(String where) in order to select a subset of rows from the table to iterate over. Writing a well-formed WHERE clause string is up to the programmer. If the string is rejected by JDBC, an exception is thrown at runtime. This mechanism doesn't fall within the JPie philosophy of making programming easier by elevating the level of discourse. In fact it can be seen as an "escape hatch," as described in [12]. Ideally JPie/qt, could provide a visual expression builder tool for where clauses, relieving the programmer of the need to know SQL syntax and constructs.

9. Conclusion

Object-oriented middleware is typically thought of as a fixed library of classes. JPie/qt is an example of how database-access middleware can be more than just an object-oriented API. JPie/qt unifies the relational and object-oriented models using dynamically generated classes. Method calls and exceptions are tied to database transactions, and objects whose instance variables can be database-bound. In this way the database features are tightly-integrated into the programming and execution environment.

As a result, the programmer no longer need be concerned with the details of the relational database model. The notion of traversing an object graph is certainly more intuitive to object-oriented programmers than constructing table joins. Having to deal directly with relational algebra is added complexity from which the middleware can shield the programmer.

Designers of programming systems should not ignore the fact that the programmer will want to store data persistently, most likely in a relational database. Libraries such as JDO [13] can help integrate the database aspects into the language. Tools such as Aspect-Oriented Programming [3] "weavers" may lead to systems that transparently add persistence to traditional programming the way JPie/qt adds it to live object-oriented software development.

Acknowledgements

We thank the entire JPie research group at Washington University in St. Louis, as well as the students and teaching assistants from CS123. We also thank Chris Gill and David Butler for useful discussions and comments on an earlier version of this paper.

This research was supported in part by the National Science Foundation under CISE Educational Innovation Grant 0305954.

References

[1] Boggs, Wendy and Boggs, Michael. 2002. Mastering UML with Rational Rose 2002. Sybex.
[2] Coplien, James and Schmidt, Douglas, eds. 1995. Pattern Languages of Program Design. Addison-Wesley.
[3] Crawford, Diane, ed. 2001. Communications of the ACM: Special Issue on Aspect-Oriented Programming. Volume 44, Issue 10. ACM Press.
[4] DuBois, Paul. 2003. MySQL. 2nd ed. SAMS.
[5] Elmasri, Ramez and Navathe, Shamkant B. 2000. Fundamentals of Database Systems. 3rd ed. Addison-Wesley.
[6] Gamma, Erich, et al. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
[7] Goldman, Kenneth J., et al. 2003. JPie: Programming is Easy. http://jpie.cse.wustl.edu/
[8] Goldman, Kenneth J. 2004. Live Software Development with Dynamic Classes. *(submitted for publication)*
[9] Hamilton, Graham, et al. 1997. JDBC Database Access with Java. Addison-Wesley.
[10] Pallemulle, Clark, and Goldman. 2004. Supporting Live Development of SOAP and CORBA Clients. http://jpie.cse.wustl.edu/sub_sections/publications/cde.htm
[11] Rumbaugh, James, et al. 1991. Object-Oriented Modeling and Design. Prentice Hall.
[12] Schmidt, Douglas, et al. 2000. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (Volume 2). John Wiley & Sons.
[13] Sun Microsystems. 2003. Java Data Objects (JDO). http://java.sun.com/products/jdo/
[14] Vlissides, John, et. al, eds. 1996. Pattern Languages of Program Design 2. Addison-Wesley.