WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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

By

Adam H. Mitz

Prepared under the direction of Professor K. J. Goldman

Thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements of the degree of

MASTER OF SCIENCE

May 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

ADVISOR: Kenneth J. Goldman

May 2004

Saint Louis, Missouri

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.

Contents

Figures.. v
1 Introduction.. 1
1.1 The Problem.. 1
1.2 Design Goals.. 2
1.2.1 API Simplicity.. 2
1.2.2 Flexibility.. 3
1.2.3 Efficiency.. 3
1.2.4 Host Language Integration.. 4
1.3 A Motivating Example.. 4
1.4 Contributions.. 7
2 Related Work.. 9
2.1 JDO.. 9
2.2 OO-Relational Mapping Tools.. 10
2.3 Patterns.. 11
2.3.1 Wrapper Facade.. 11
2.3.2 Half-Object Plus Protocol.. 12
2.3.3 Crossing Chasms.. 12
3 The JPie Environment.. 14

3.1 Programming in Java with JPie.. 14
3.2 Executing and Debugging Dynamic Classes 15
3.3 Java Source Code Generation 16
3.4 JPie Internals 16
4 Using JPie/qt 18
4.1 Example Program Context 18
4.1.1 The SmartMail System 18
4.1.2 The SmartMail Accumulator 19
4.1.3 Java Servlets with JPie 20
4.2 Initial Setup: Database and JPie/qt 21
4.2.1 Database Engine and Schema 21
4.2.2 Opening the Database in JPie... 21
4.2.3 Establishing Table Relationships 23
4.3 Developing the Example Program 24
4.4 Test and Debugging 27
4.5 Server-side Execution 27
5 Implementation 28
5.1 Metadata Inspection and Dynamic Class Generation 28
5.1.1 The Primary Key 29
5.1.2 Related Tables 29
5.1.3 Classes in the JPie/QT Implementation 31
5.2 Field Access and Update 33
5.2.1 Class in the JPie/QT Implementation: DatabaseField 34
5.3 Table Iteration 34
5.3.1 Complete Iteration 34
5.3.2 Filtered Iteration... 34
5.3.3 Class in the JPie/QT Implementation: RecordCollection 35
5.4 Transactions 36
5.5 Browsing Tables 37

6 Performance 40
6.1 Overhead 40
6.2 Experience with CS123.. 41
7 Future Work 43
7.1 Additional Features 43
7.1.1 SQL Code Generation and Optimization 43
7.1.2 Runtime Library 44
7.1.3 Database Creation and Schema Modification...................................... 44
7.1.4 Visual SQL WHERE Clause Builder.. 45
7.2 Conclusion 45
Appendix - UML Structure Diagram for JPie/qt 47
References.. 49
Vita... 51

Figures

1-1. Listing Game Players and Locations Using JDBC Directly.. 6

1-2. Listing Game Players and Locations Using JPie/qt.. 7

3-1. The JPie Extended Reflection Hierarchy [7].. 17

4-1. SmartMail Database Schema.. 21

4-2. Launching the Database Connection Wizard 22

4-3. Specifying the JDBC Connection Parameters... 22

4-4. Selecting a Subset of Tables to Import... 23

4-5. The Database Table Represented By a JPie Class... 23

4-6. The "links to" Element of a Field Declaration 24

4-7. Relevant Methods from HttpServlet.. 25

4-8. Methods May Be Declared As Transactions 26

5-1. Methods Generated Based on Table Relationships.. 30

5-2. Highlighted Instances and Table Browsing... 38

6-1. Execution Time Comparison (ms).. 40

1 Introduction

1.1 The Problem

Modern programming languages and environments are making great progress in reducing the complexity of software construction. The stated goal of object-oriented programming is to increase reusability and maintainability [10]. Besides making it easier for established programmers to write more robust and larger software systems, advances have also made it possible for a much wider audience to learn the craft of software construction. Curricula instruct new programmers in the ways of object-oriented programming and concentrate on data abstraction, encapsulation, inheritance, and polymorphism. They learn through example laboratory programs that are coded from scratch using only the standard library of their chosen language.

Having learned the fundamental concepts behind object-oriented programming, the syntax and library of a given platform, and coded some sample programs, one might expect the new programmer to be able to apply these skills to building real-world software systems. However, this is not often the case. 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. The existing relational database APIs for most languages (with JDBC for Java as our example) are too complex for object-oriented programmers in that they require extensive knowledge of the relational database model. The complexity of relational database APIs is a problem not just for beginning programmers, but also for veteran object-oriented programmers with limited knowledge of the theory of relational databases.

1.2 Design Goals

JPie/qt is a part of JPie, a tightly-integrated development environment for live construction of Java applications [6]. JPie enables inexperienced programmers to create Java programs with the standard Java platform APIs. The JPie/qt middleware bridges the object-oriented and relational worlds in order to provide JPie programmers with access to a relational database management system. JPie/qt exposes an object-oriented API and internally uses Java's JDBC [8] system to communicate with any relational database.

1.2.1 API Simplicity

JPie/qt's major design goal is API simplicity. 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 API simplicity is the naming of classes and fields. JPie/qt classes and fields are named corresponding to tables and columns in the database. Thus the names are directly relevant to the programmer.

1.2.2 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.

1.2.3 Efficiency

Run-time performance of applications developed with JPie/qt should be comparable to those programmed directly with JDBC in the JPie environment. To this end, JPie/qt makes use of lazy instantiation and caching in order to minimize the amount of communication necessary between the JPie/qt and JDBC layers.

The primary example of lazy instantiation involves the representation of database rows as Java objects. 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 copied in eagerly. Thus programs that don't care about some of the data fields will not pay the costs of copying that data.

1.2.4 Host Language Integration

JPie/qt's host environment is JPie. 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 similar GUI look-and-feel and semantics to those found in JPie.

1.3 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, a function called "listPlayerLocations" needs to be written. This function 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 1-1 reveals a surprising amount of incidental complexity [11][1] that this 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. Implicit in this statement is that the programmer must know the relational schema and its primary key-foreign key

[1] The referred-to text uses the term "accidental complexity" for this idea.

relationships. As with any embedded textual 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).

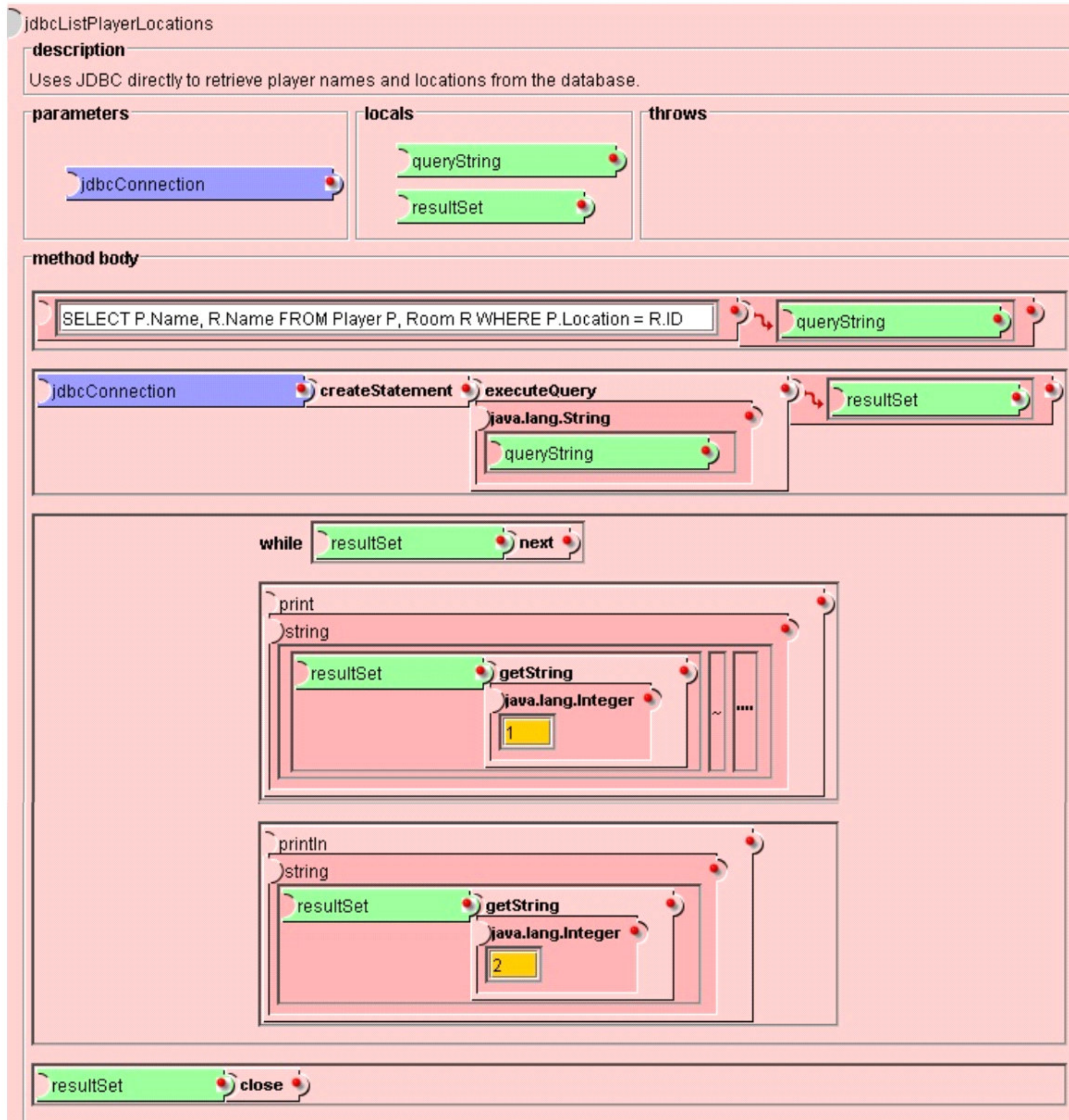


Figure 1-1. Listing Game Players and Locations Using JDBC Directly

Figure 1-2 shows a method that is equivalent to the one in Figure 1-1, 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 operations on them: getAllRecords, getName, and getLocationAsRoom.

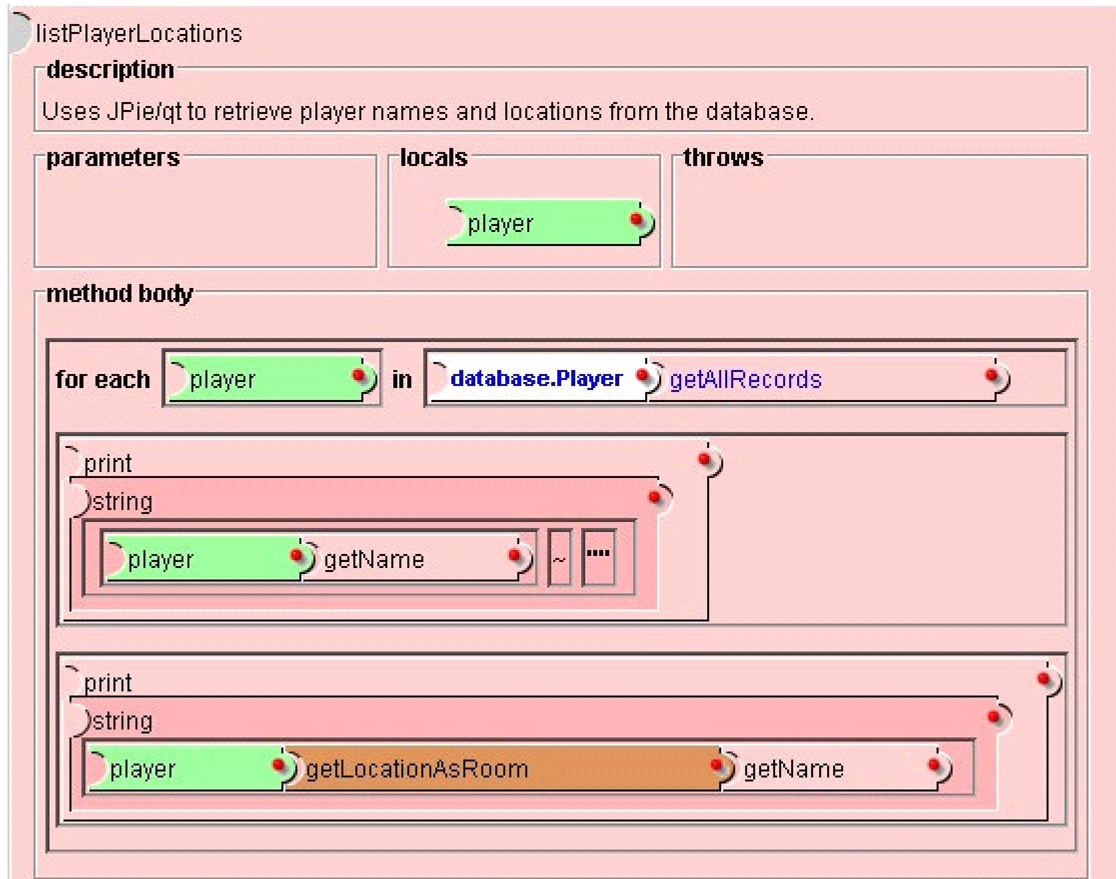


Figure 1-2. Listing Game Players and Locations Using JPie/qt

1.4 Contributions

JPie/qt brings the power of a relational database management system to a novice programmer or application-domain expert who knows only the basic 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], [5], [11], [14] 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[2]. 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 is very similar to 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

---

[2] 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.

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.

2.3 Patterns

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

2.3.1 Wrapper Facade

Wrapper Facade [11] 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.

In addition to its effect on JPie/qt, Half-Object Plus Protocol can be seen in another JPie system known as CDE [9], the Client Development Environment for distributed systems.

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.

The authors of the pattern language have used the patterns to implement a system similar to JPie/qt using Smalltalk as the object-oriented language. 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 [6] is an interactive programming environment that makes software development accessible to a wider audience. It 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. The "Packages and Classes" window provides a starting point by listing in a tree view all of the types available to the system. This includes the Java 2 Platform SDK (J2SE) as well as the user's own classes. Classes

created in the JPie system are known as "dynamic classes" and any dynamic class can be opened, showing a class designer window.

The class designer allows the user to define the fields, methods, and constructors of the class. Additionally, support is given for designing a 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. The JPie statement builder is a calculator-like interface with buttons for building literals, algebraic and Boolean expressions, java.lang.Math functions, and control structures.

3.2 Executing and Debugging Dynamic Classes

In addition to the class design elements, the dynamic class window also lists 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 view is shown in the window. Should an exception occur in a dynamic class method, a debugger window will appear. 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 Java Source Code Generation

JPie can generate textual Java source code corresponding to dynamic classes. In doing so, the JPie environment allows the programmer to read the dynamic class in standard Java syntax. This is helpful for those learning the Java syntax. Moreover, this feature allows JPie to compile the user's code to Java bytecode for execution as a standalone application or applet. The source code generator currently doesn't support database-connected classes. This is an area for future development.

3.4 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 3-1) ClassWrapper, FieldWrapper, MethodWrapper and ConstructorWrapper. The abstract classes DClass, DMember, DField, and DConstructor represent the interfaces common to both compiled and dynamic classes.

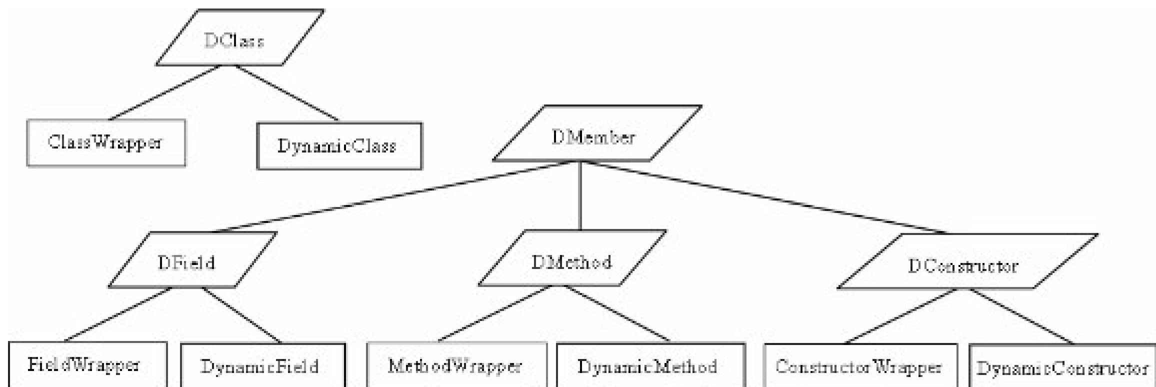


Figure 3-1. The JPie Extended Reflection Hierarchy [7]

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 [5] 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

4.1 Example Program Context

We illustrate the JPie/qt system in the context of a mail service application. We begin with a description of the application itself. Then we discuss how JPie and JPie/qt are used to build a software component of the mail service.

4.1.1 The SmartMail System

SmartMail is a system for automating "multicast" email requests and 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 the JPie 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[3] for recipients to fill in. Before being sent out 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.

4.1.2 The SmartMail Accumulator

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

---

[3] Support for plain-text clients is planned by providing a URL that can be copied and pasted into the user's web browser, bringing up a form similar to what would have been seen in an HTML-enabled email client.

4.1.3 Java Servlets with JPie

Java Servlets [12] are classes that implement a given interface so they can be used by a web server to process HTTP requests. Whereas a web server would by default read a static page of content from disk in response to an HTTP request, a Servlet-enabled web server can execute a method of user code (the Servlet) in response to an HTTP request. This allows dynamic and flexible web services to be engineered with Java technology.

JPie is a client-side GUI application for software construction. One way to write a Servlet using JPie would be to export the code as a standard Java class. The JPie/qt system doesn't currently support exporting code (see section 7.1.1). Instead of exporting Java code, another strategy is to run the JPie application itself as the Servlet. In order to support JPie on the server-side, a "silent" mode and a "Servlet" mode were created. In silent mode, JPie shows no GUI elements. It interacts only with the standard input/output streams and file system. The Servlet mode is actually a wrapper class, "ServletPie," which conforms to the Servlet interface. This allows JPie itself to fit into the Servlet architecture. The Servlet's XML configuration file indicates to ServletPie which DynamicClass should be used to handle the actual HTTP requests.

4.2 Initial Setup: Database and JPie/qt

4.2.1 Database Engine and Schema

For the SmartMail prototype, we chose the free database mysql as the relational database engine. We designed a schema (Figure 4-1) for the SmartMail system and used the mysql tools to set it up. Arrows in the diagram indicate N-to-one relationships.

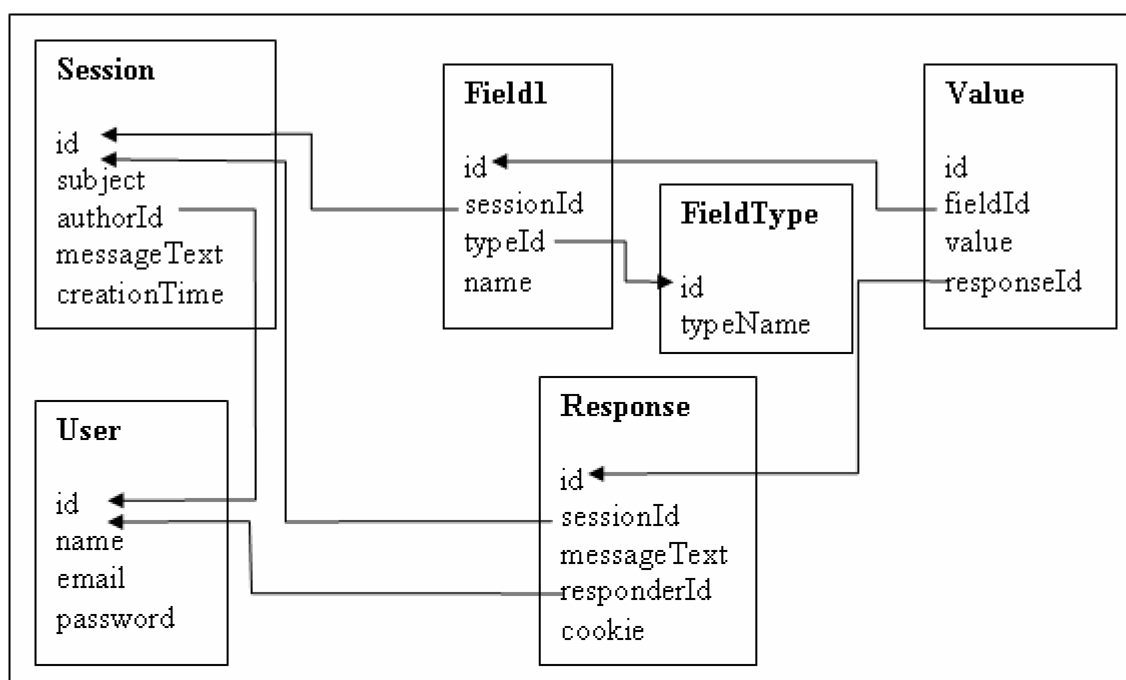


Figure 4-1. SmartMail Database Schema

4.2.2 Opening the Database in JPie

With the schema in place, JPie's "Open Database" command is executed (Figure 4-2). This brings up a wizard that prompts the user for the JDBC connection string, user name, and password (Figure 4-3), followed by a selection box listing the available

database tables (Figure 4-4). 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, the JPie Packages and Classes window lists classes corresponding to each table. (Figure 4-7)

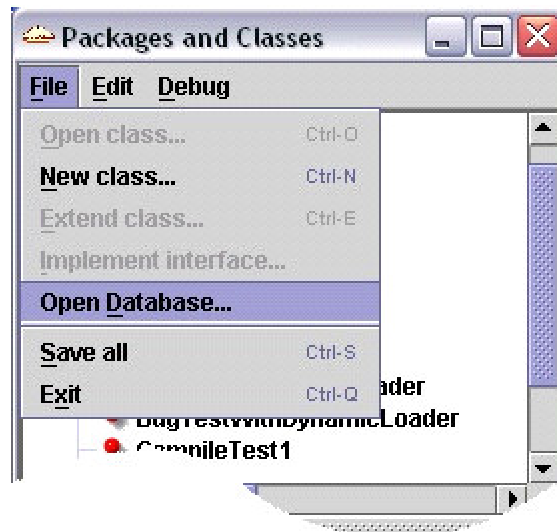


Figure 4-2. Launching the Database Connection Wizard

Figure 4-3. Specifying the JDBC Connection Parameters

Figure 4-4. Selecting a Subset of Tables to Import

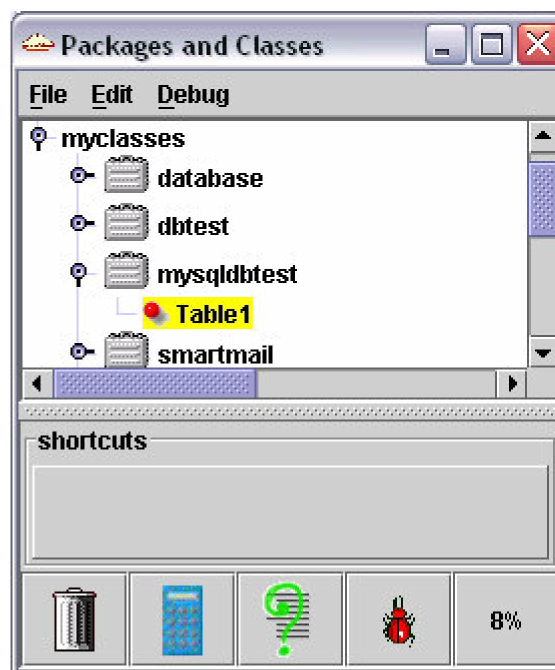


Figure 4-5. The Database Table Represented By a JPie Class

4.2.3 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[4], 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 (Figure 4-6) for the foreign key field. In the example of Figure 4-1, to associate the Session table's field "authorId" with the User table, the programmer drags the User class from the Packages and Classes Window on to the "links to" area of the declaration of authorId inside the Session class. This step is repeated for each relationship.

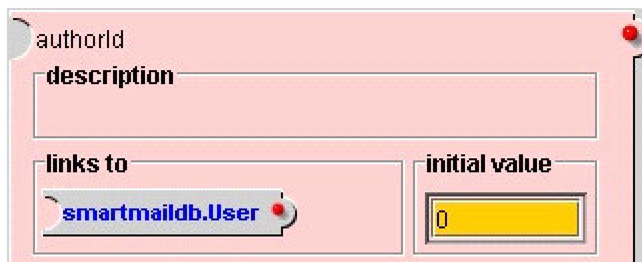


Figure 4-6. The "links to" Element of a Field Declaration

4.3 Developing the Example Program

Only one class is needed to implement the accumulator component. This class is named Accumulator and it extends the J2EE provided class javax.servlet.http.HttpServlet. The important methods that need to be overridden to

---

[4] Some databases (mysql included) do not even keep track of table relationships.

provide our application-specific functionality are summarized in figure 4-7. The methods init() and destroy() are called at the beginning and end of the Servlet object's lifetime, respectively. doPost() is called to process each HTTP request (we're using the POST request verb).

Name	Parameters	Returns	Throws
init	( )	void	ServletException
doPost	(HttpServletRequest, HttpServletResponse)	void	ServletException, IOException
destroy	( )	void	

Figure 4-7. Relevant Methods from HttpServlet

The init() and destroy() methods are useful for test and debugging but do not contribute to our Servlet. This is because our Servlet is simple enough that it is stateless (the state is kept in the database). The doPost() method's parameters provide it all of the information it needs. It uses a JDBC database connection, but that setup is handled automatically by JPie/qt.

The algorithm for doPost() is simple. First we must look up the response object from the database that corresponds to this response. A cookie[5] [11] is used to uniquely identify responses (it's embedded in a hidden form field in the HTML). This ensures that

[5] Also known as an "Asynchronous Completion Token"

the response is valid. Next we need to iterate over the Fields (from the Field1 table[6]) for this session and create corresponding Values for this response.

This operation comprises multiple changes 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" (Figure 4-8) 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.

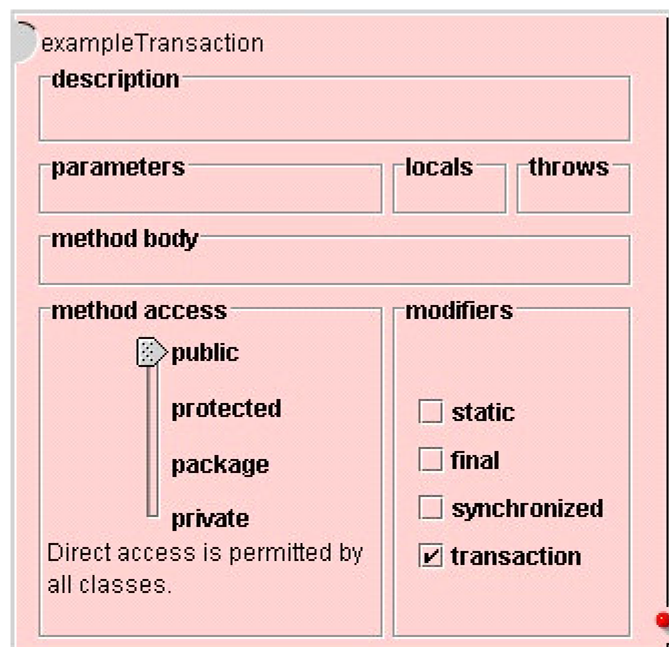


Figure 4-8. Methods May Be Declared As Transactions

---

[6] So named because "Field" caused naming conflicts with JPie and/or java.lang.reflect.Field

4.4 Test and Debugging

Testing and debugging a server-side program can prove challenging. The Accumulator Servlet is simple enough that it can be used outside of the Servlet framework for debugging purposes. The test strategy is to simply call the doPost method manually. This will allow us to take advantage of JPie's full-featured debugger. JPie provides a "Manual Test" method in the instances pane. By calling doPost from within the manual test method and examining the results in the database, we can ensure that doPost is behaviorally correct independent of the Servlet framework.

4.5 Server-side Execution

After the Servlet is tested it can be executed on the server. Dynamic classes are saved by JPie as two files: one .class and one .dclass, and each file needs to be copied to the server. The Servlet can then be configured as described in section 4.1.3. At this point the Servlet engine is actually running an instance of JPie, and forwarding method calls for doPost through ServletPie to the Accumulator. The Accumulator proceeds by inserting the appropriate rows into the database tables.

5 Implementation

The Java classes that make up the implementation of JPie/qt connect the core JPie extended reflection system (see section 3.4) to JDBC. This is shown in the UML Static Structure diagram in the appendix.

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 in the "Packages and Classes" window. The purpose of this command is to establish a link between the relational database and the JPie environment. The user is prompted for the JDBC connection string and username/password pair. The system then makes an initial connection to the database and retrieves the list of available tables. The user selects a subset of tables to bring into JPie and a folder (package) in which to put the classes.

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. This procedure is not repeated in subsequent sessions, the user needs only to open or refer to the generated JPie 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. Not all JDBC drivers support this method, however. If this first approach fails, a simple heuristic is applied and the first integer column is considered the primary key. This is a reasonable heuristic because it is traditional in database design to position 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” (next to “initial value,” see Figure 4-6). 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).

Figure 5-1. Methods Generated Based on Table Relationships

When a relationship between tables is established, two methods are generated. These methods allow the programmer to navigate the N-to-1 table relationship using the Java objects. Figure 5-1 shows the methods generated when the AuthorId field of the Session table is related to the User table (see the schema in Figure 4-1).

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 as the Foreign Key Closure (since it is implemented as a closure in the JPie system, as explained in section 5.1.3) and in the example of Figure 5-1 it is "getAuthorIdAsUser," in the Session class.

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 and in the example of Figure 5-1 it is "getSessionSetForAuthorId," in the User class.

5.1.3 Classes in the JPie/QT Implementation

ConnectionManager

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.

DynamicDatabaseClass

DynamicDatabaseClass 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[7].

DatabaseTable

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 it 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.

ForeignKeyClosure, PrimaryKeyClosure

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.

ForeignKeyClosure and PrimaryKeyClosure each extend Closure to allow traversal of database relationships. ForeignKeyClosures, such as getAuthorIdAsUser

---

[7] 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.

(Figure 5-1), traverse the relationship from foreign key to primary key. In this 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, 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. Since the PrimaryKeyClosure logically returns a set, an object implementing java.util.Collection is returned. The RecordCollection class (section 5.3.3) is used 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 (Figure 4-6). 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

5.3.1 Complete Iteration

Each dynamic database class is generated with a static method "getAllRecords()". This method returns a java.util.Collection that represents all objects in the table. Calling this method causes one query to be run against the database (creating a JDBC ResultSet) 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

"Where" Clause

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[8]. 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. Notice that programmers without knowledge of SQL syntax are not hampered by this. They can use complete iteration combined with "if" statements to achieve the same results, perhaps with a decrease in performance.

Primary Key Traversal

Traversing the table relationship from the primary key side to the foreign key side can be thought of as a special case of the "where" clause. In the example of Figure 5-1, the primary key closure getSessionSetForAuthorId is translated into the SQL clause "WHERE AuthorId = X" and applied to the Session table. This results in the correct subset of rows being returned from the RecordCollection's iterator.

5.3.3 Class in the JPie/QT Implementation: 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

[8] getSomeRecords() is so named to contrast with getAllRecords()

statement is constructed 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

Methods in JPie have an extra modifier (like "final" or "static") called "transaction" (Figure 4-8). 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. A rollback is done if the method exits by throwing. Commit-mode information is stored per-thread[9] since different threads may be executing different database operations, some in transactions and some not.

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

[9] This is an example of thread-specific storage [11].

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

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 currently selected instance on the right. The panel on the right shows the view of the current instance. JPie/qt enhances the instances list in two ways. First, bound instances (defined below) are highlighted in green. Second, table browsing can be enabled in order to list instances corresponding to each row in the table. Each of these features can be seen in Figure 5-2.

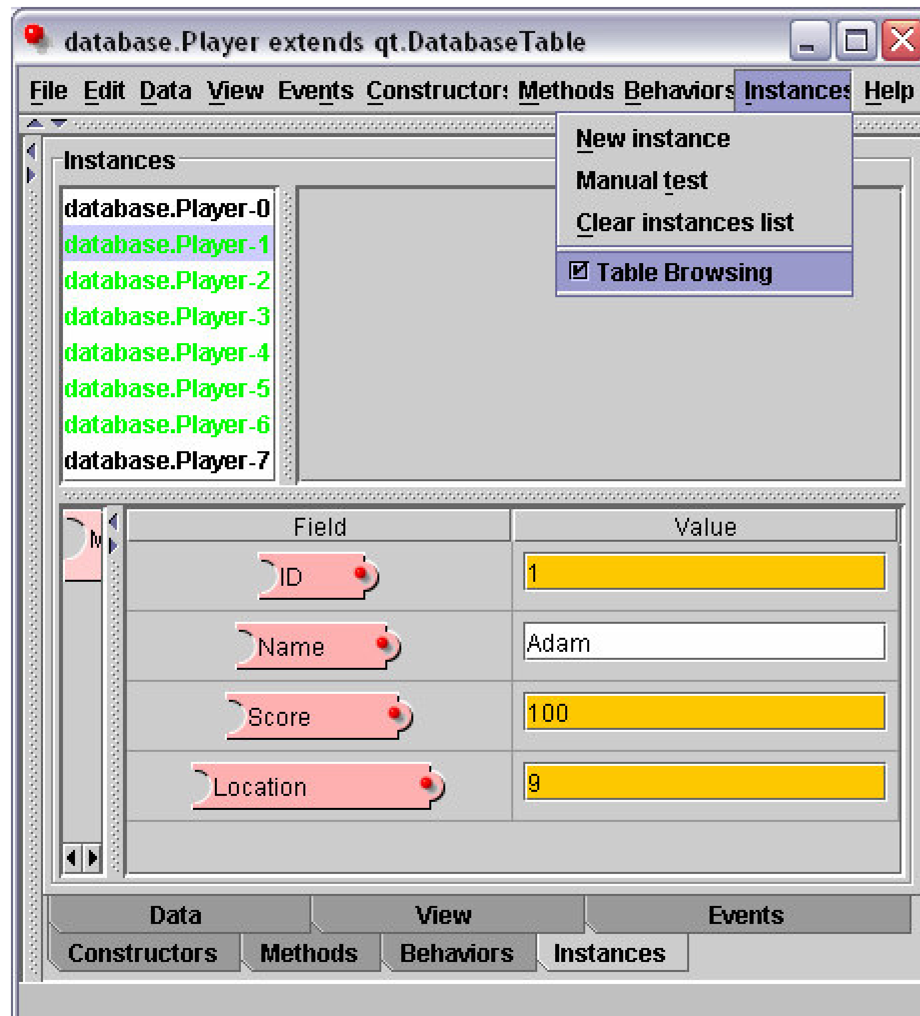


Figure 5-2. Highlighted Instances and Table Browsing

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. Bound instances are highlighted in color in the JPie instances list.

Similarly, each row of the database table is also in one 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. Dynamic database classes also add a menu option to the JPie "Instances" menu that allows the user to enable and disable the "table browsing" mode. In this mode all rows of the table are shown in the instances list, whether or not they are resident.

6 Performance

6.1 Overhead

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. Testing for execution-time overhead was completed using the following procedure. 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)…"). This is a worst-case scenario for JPie/qt since it pays the costs of caching the objects but does not get any benefit from it. Results from averaging five runs are shown in Figure 6-1.

JPie/qt	JPie/JDBC	JDBC	SQL Aggregate
79,100	2,490	376	68.2

Figure 6-1. Execution Time Comparison (ms)

From the experimental results, we can see the order of magnitude cost of using JPie/qt over JDBC for batch-processing operations. This can be attributed to the fact that JPie/qt and JPie create objects for each row. Included in the costs of object creation are the costs of cache management in both JPie/qt and JPie itself. Since exported applications would be compiled Java programs, many of these costs would be avoided. However, JPie/qt is not currently suitable for production software where batch-processing performance is a concern. The database used for these tests was a local instance of MySQL [3].

6.2 Experience with CS123

The students of 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 followed written directions to complete the software that displayed the current state and allowed players to move through the virtual world. Anecdotal evidence from this experience didn't uncover any performance problems. This is in contrast to the experiments in Section 6.1.

The discrepancy between the two scenarios can be explained by the differences between the two applications. While the experiments in Section 6.1 were CPU-bound 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. Future work will seek to remedy this by "compiling" JPie/qt applications directly to the equivalent SQL statements (see Section 7.1.1). This is analogous to JPie itself, which uses a mixed compiled/interpreted mode of execution, but allows the programmer to compile their work down to Java classes for deployment.

7 Future Work

7.1 Additional Features

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.

7.1.1 SQL Code Generation and Optimization

Performance may 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 implemented the "select" operation of the relational algebra [3], instead of relying on the relational database.

Programs executing in the JPie environment are not optimized, allowing for live debugging and modification. Should the programmer need to run their software in a

high-performance environment, the code generation features are available to create Java .class files. As a future extension, code generated from JPie/qt programs could be, when possible, optimized SQL queries.

7.1.2 Runtime Library

Programs using JPie/qt currently must run within the JPie environment. JPie has Java code generation features that allow programs to be exported as .java or .class files. At this time, these features 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.

7.1.3 Database Creation and Schema Modification

Currently JPie/qt maps schemas from databases to the object-oriented program. The converse operation is also useful in many cases. This is the situation where 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, schema changes made to the dynamic class (adding, removing, renaming fields) will be reflected in the database table.

7.1.4 Visual SQL WHERE Clause Builder

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 [11]. 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.

7.2 Conclusion

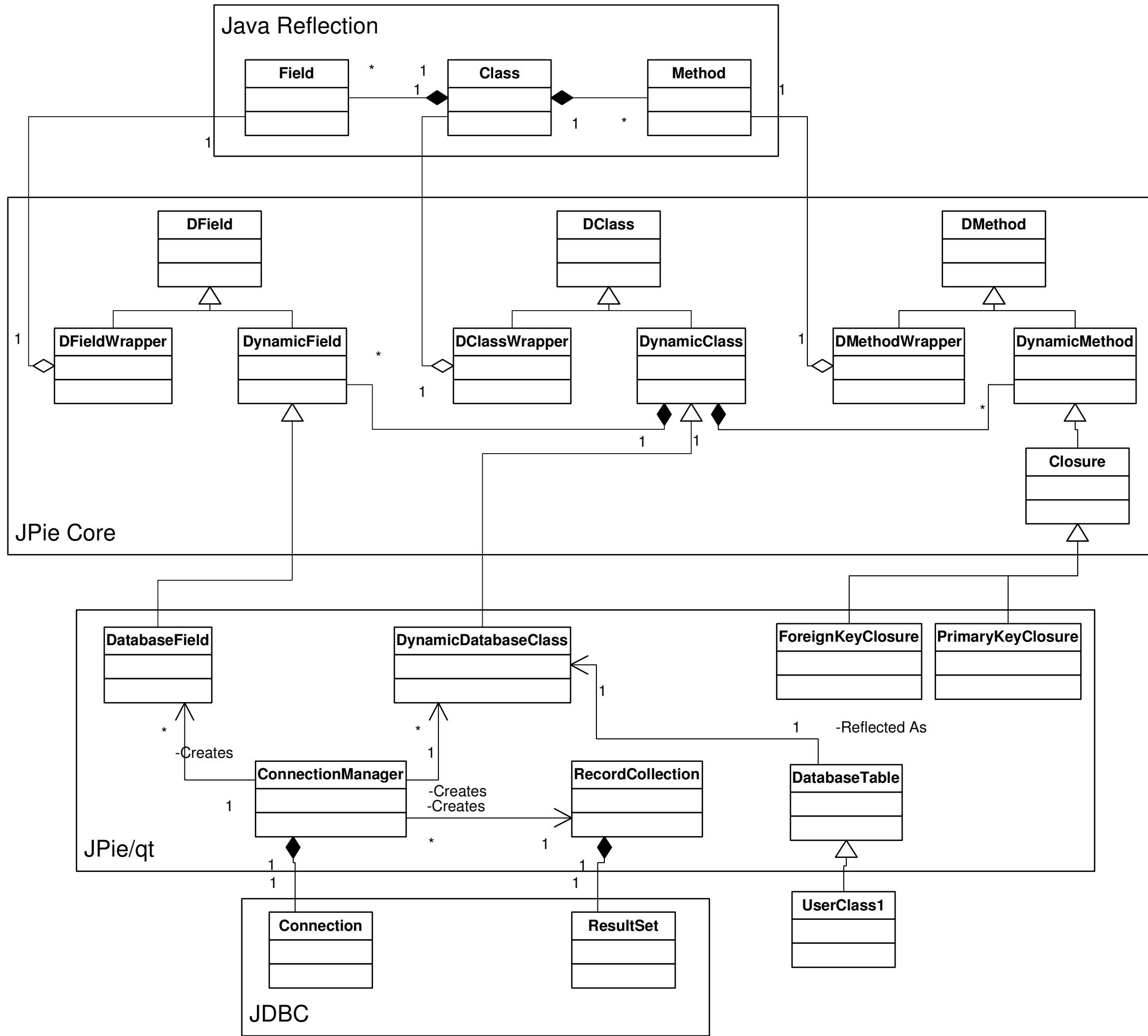
JPie/qt is an example of how database-access middleware can be more than just an object-oriented API. By generating classes for the programmer and overriding runtime behavior like method calls (for transactions) and field access, the database features are tightly-integrated into the programming and execution environment.

This results in productivity gains for the programmer, who no longer has to concern himself or herself with the details of the relational database model. The notion of traversing an object graph is certainly more intuitive than constructing table joins. Programmers think in terms of classes and the resulting rows of a join are cross-products of classes, a concept not typically dealt with.

Designers of programming systems shouldn't ignore the fact that the programmer will want to store data persistently, most likely in a relational database. Libraries such as JDO are starting to do a better job integrating the database aspects into the language. Tools such as Aspect-Oriented Programming "weavers" could form the basis of systems that truly transparently add persistence to traditional programming the way JPie/qt adds it to JPie.

Appendix - UML Structure Diagram for JPie/qt

(See following page)


Java Reflection
Field
Class
Method
JPie Core
DField
DClass
DMethod
DFieldWrapper
DynamicField
DClassWrapper
DynamicClass
DMethodWrapper
DynamicMethod
Closure
JPie/qt
DatabaseField
DynamicDatabaseClass
ForeignKeyClosure
PrimaryKeyClosure
-Creates
ConnectionManager
-Creates
-Creates
RecordCollection
-Reflected As
DatabaseTable
UserClass1
JDBC
Connection
ResultSet

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] DuBois, Paul. 2003. MySQL. 2nd ed. SAMS.

[4] Elmasri, Ramez and Navathe, Shamkant B. 2000. Fundamentals of Database Systems. 3rd ed. Addison-Wesley.

[5] Gamma, Erich, et al. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[6] Goldman, Kenneth J., et al. 2003. JPie: Programming is Easy. http://jpie.cse.wustl.edu/

[7] Goldman, Kenneth J. 2004 (in progress). Fine-Grain Dynamic Classes.

[8] Hamilton, Graham, et al. 1997. JDBC Database Access with Java. Addison-Wesley.

[9] Pallemulle, Sajeeva, Clark, Vanessa H., and Goldman Kenneth J. 2004. Supporting Live Development of SOAP and CORBA Clients. http://jpie.cse.wustl.edu/sub_sections/publications/cde.htm

[10] Rumbaugh, James, et al. 1991. Object-Oriented Modeling and Design. Prentice Hall.

[11] Schmidt, Douglas, et al. 2000. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (Volume 2). John Wiley & Sons.

[12] Singh, Inderjeet, et al. 2002. Designing Enterprise Applications with the J2EE Platform. 2nd ed. Addison-Wesley.

[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.

Vita

Adam Mitz received Bachelor of Science degrees in Computer Science and Computer Engineering from Washington University in St. Louis in May, 2003. Having begun graduate research and coursework in January, 2002, he continued his studies in the Computer Science and Engineering department through May 2004 when he received a Master of Science in Computer Science degree. Professor Kenneth J. Goldman was his advisor on the research and this thesis.

M.S. May 2004

Short Title: Database Middleware / Live Software Mitz, M.S. 2004