SQL Compiler (for Java)

Free your Java code from SQL statements – compile them to Java classes.

Abstract

This article describes a SQL Compiler tool (SQLC), which generates Java classes from SQL statement and table metadata. By doing so SQLC decouples Java and SQL code and enforces clear separation of concerns between database and Java code and division of labor between data modeler and Java developer. The article also ruminates about SQLC, O/R mapping frameworks (taking Hibernate as an example) and plain JDBC applicability in different contexts.

Introduction

Recently I worked on a Java application and needed to do a lot of database interaction. I didn't want to use plain JDBC for obvious reasons (nonetheless described below). I already created a data model and I didn't want to do double work by mirroring data model in Java by hand.

So I looked around for a carpal channel friendly tool, something like erstwhile FoxPro 2.6. To my surprise I found none. Nowadays everybody seemed to care about flexibility but not simplicity.

After pondering for awhile I preferred creative typing over mundane one and wrote a tool which generates interfaces and methods from database metamodel. Lo and behold: SQLC was born!

SQLC generates Java classes and interfaces from information provided by `java.sql.Connection.getMetaData()`, `java.sql.PreparedStatement.getMetaData()` and `java.sql.PreparedStatement.getParameterMetaData()` methods. It uses BCEL and helper classes and techniques described in XREF:ARTICLE_6207. SQLC-generated classes use Squirrel[3] as a foundation which allows to minimize amount of generated code.

You may think: "Oh, yet another O/R mapping tool, why would I waste my time reading about it?" Well, SQLC is not an O/R mapping tool at all. Like spiritualists and materialists answer differently to the question "What is primary – spirit or matter?", SQLC and O/R mapping tools answer differently to the question "What is primary – data or objects?". SQLC treats data being primary and being only data – i.e. no behavior. E.g. credit card has no behavior per se – it is just a tuple. Behavior belongs to objects operating with credit card, not to the credit card itself.

SQLC also does something what other tools don't do – it compiles parameterized statements to Java methods.

This is a short summary of SQLC advantages:

- Simplicity – SQLC requires very little development time configuration and no deployment time configuration (no deployment descriptors at runtime).
- Robustness – Build time verification of SQL statements. SQLC uses metadata information provided by the target database. Problems with SQL – syntax errors, invalid DB object names – are revealed at build time.
- Reusability – compiled classes can be used by many applications working with the same database. Generated classes, for example, can be distributed as a jar file. In application server environment engine classes can be mounted to JNDI tree. Storing statements in the database allows to reuse tuned statements in non-Java applications as well.
- Separation of concerns and division of labor – Data modeler models the database (DDL) and also develops SQL statements (DML) optimal for the database. Java developer uses generated classes/methods to access/modify data. (S)he is not required to know SQL at all and needs to possess only superficial understanding of the underlying database structure.

Corollaries of the previous point are:

- Modifiability and maintainability – compiled classes define an interface between the database and Java code. Data modeler is free to modify DDL and DML – replace joins with nested selects, or include execution plan to queries – without the need of touching the Java code.
- Reduced resource demand – lower grade, and thus cheaper, Java resources (developers) can be used. With SQLC Java developers don't need to use JDBC API or other tool-specific API, they use compiled methods, which, if named properly, are self-descriptive.
- Increased productivity according to Adam Smith.
- Testability – statements can be tested independently of Java application in SQL console. Compiled classes can also be tested independently by, say, JUnit without the need of complicated fixtures.

The following sections show how to use SQLC and then compare SQLC, plain JDBC and O/R mapping (using Hibernate[9] as an example) applicability in different contexts.

Compilation

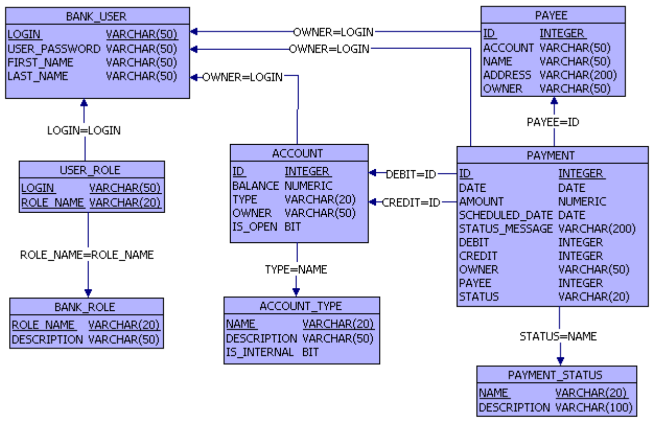

Illustration 1

Illustration 1 shows a sample model, which will be used to generate classes and interfaces. We will create a simple banking application capable to make fund transfers and collecting service charge from accounts with low balance. Full source code of the sample application can be downloaded from [1].

The first step is to generate classes using `sqlc` Ant task:

```
<sqlc
script="src/com/pavelvlasov/sqlc/samples/Bank.sql"
    dir="sqlc_generated"
    docDir="sqlcDoc"
    package="com.pavelvlasov.sqlc.samples"
    masterEngine="BankEngine"
>
    <table/>
</sqlc>
```

In the snippet above classes are generated using `Bank.sql` script. In this case Hypersonic[6] in-memory database is created, the script is executed to create database objects, and then Java classes are generated. This approach works fine if DDL in the script is compatible with Hypersonic. If you'd like to generate classes straight from the target database (preferred method) then instead of `script` attribute nested `<connection>` element shall be specified. This is a sample connection element for Oracle:

```
<sqlc
    dir="sqlc_generated"
    docDir="sqlcDoc"
    package="com.pavelvlasov.sqlc.samples"
    masterEngine="BankEngine"
>
    <connection
        driverClass="com.inet.ora.OraDriver"
        url="jdbc:inetora:server:port:DB"
         user="DBUSER"
         password="DBPASSWORD"/>
     <table/>
</sqlc>
```

Please note that it uses ORANXO[11] driver because drivers provided by Oracle itself are not fully JDBC compatible (see compatibility section below).

SQLC task shown above produces .class files in `sqlc_generated` directory and HTML documentation in `sqlcDoc` directory using table metadata. `<table/>` element matches all tables in the database.

Now we will use the generated `com.pavelvlasov.sql.samples.BankEngine` class to perform database operations. BankEngine constructor takes one argument of type `com.pavelvlasov.sql.SQLProcessor` [3]. `SQLProcessor` can be created either from `java.sql.Connection` or from `javax.sql.DataSource`. Our banking application uses getProcessor() and getEngine() helper method to obtain the engine:

```
private SQLProcessor processor;

private SQLProcessor getProcessor() throws BankException {
   if (processor==null) {
       ...
    }
    return processor;
}

private BankEngine engine;

private BankEngine getEngine() throws BankException {
    if (engine==null) {
        engine=new BankEngine(getProcessor());
    }
    return engine;
}
```

In application server environment the engine could be bound to JNDI tree and obtained by the application through JNDI lookup.

We can't make (financial) transactions without creating customers, accounts and other reference objects first. Below is a fragment of database initialization code, which shows how to insert records in each of database tables except `PAYMENT`:

```
BankEngine engine=new BankEngine(processor);
engine.insertBankUser("joe", "pwd", "Joe", "Brown");
...
engine.insertBankRole("customer", "Bank customer");
...
engine.insertUserRole("joe", "customer");
...
engine.insertAccountType("checking", "Checking account", false);
...
engine.insertAccount(1, new BigDecimal(1000), "checking", "joe", true);
...
engine.insertPaymentStatus("scheduled", "Scheduled to be processed in the future");
...
engine.insertPayee(1, "123-456", "KCP&L", "Kansas", "joe");
```

Simple, huh? No SQL involved. Database objects (Connection, Statement, ResultSet) lifecycle management (open/close) is encapsulated in SQLProcessor[3].

SQLC takes column and parameter nullability into account. If there is a column which maps to Java primitive type and it is nullable then corresponding wrapper class is used in generated interfaces and methods. E.g. `Payment.getPayee()` returns `java.lang.Integer`. On the other hand, if column/parameter is not nullable then primitive type is used. E.g. `Payment.getId()` returns `int`.

Now let's get to the business and create a method which transfers funds from one account to another. We want database to do calculations of the new account balance and thus we add two lines to `sqlc` task:

```
<update name="AccountCredit">UPDATE ACCOUNT SET BALANCE=BALANCE+? WHERE ID=?</update>
<update name="AccountDebit">UPDATE ACCOUNT SET BALANCE=BALANCE-? WHERE ID=?</update>
```

Then we run the task again. This results in two additional methods in `BankEngine`: `accountCredit(BigDecimal, int)` and `accountDebit(BigDecimal, int)`. We could also store the statements in a database table and use `<dbstatements>` nested element in `<sqlc>` task or store them to an xml file and use `<statements>` nested element.

Here is the `transfer()` method per se:

```
void transfer(BigDecimal amount, int debit, int
credit, String owner, Integer payee) throws BankException, SQLException {
    BankEngine engine=getEngine();
    Account da=engine.getAccount(debit);
    if (da.getBalance().compareTo(amount)<0) {
        throw new InsufficientFundsException(
            "There are not enough funds on account "
            +da.getId()
            +". Required: "
            +amount
            +", available: "
            +da.getBalance());
    }
    engine.accountDebit(amount, debit);
    engine.accountCredit(amount, credit);
    Date now=new Date(System.currentTimeMillis());
    engine.insertPayment(
            getProcessor().nextPK("PRIMARY_KEY", "PAYMENT"),
            now,
            amount,
            now,
            "Processed",
            debit,
            credit,
            owner,
            payee,
            "processed");
}
```

Now we'll take a look at how SQLC generates methods using index information. We need to collect service charge for accounts with balance below some limit. So first of all we need to iterate over such accounts. It's not a bad idea to use an index on `BALANCE` column to make iteration more effective. We'll name index `IX_ACCOUNT_SQLC$M_BALANCE`. `SQLC$` tells SQLC to generate methods using this index. `M` is the index mode. See [2] for full list of modes. `BALANCE` is the postfix for generated method. Using this information SQLC generates `getAccountBalanceLE()` methods, which we'll use in our first implementation of `serviceCharge()` method:

```
void serviceCharge(BigDecimal limit, BigDecimal charge, int chargeAccount) throws BankException, SQLException {
    BankEngine engine=getEngine();
    Iterator it=engine.getAccountBalanceLE(limit).iterator();
    while (it.hasNext()) {
        Account account=(Account) it.next();
        if (!account.getOwner().equals("_bank") && account.getId()!=chargeAccount && account.getBalance().compareTo(ZERO)>0) {
            BigDecimal amount=charge.min(account.getBalance());
            transfer(amount, account.getId(), chargeAccount, "_bank", null);
        }
    }
}
```

While the above method demonstrates usage of index-generated method it has several flaws:

- All account types are charged in the same manner
- We had to use if-condition to avoid charging of accounts belonging to the bank itself and accounts with no balance.

To improve the method we'll add a query to `sqlc` task:

```
<query name="AccountSubjectToServiceCharge">
SELECT * FROM ACCOUNT WHERE BALANCE &gt; 0
AND BALANCE &lt; ? AND TYPE=?</query>
```

This will result in generation of `getAccountSubjectToServiceCharge()` method, which will be used for enhanced service charge collection:

```
void serviceChargeEx(String accountType, BigDecimal limit, BigDecimal charge, int chargeAccount) throws BankException, SQLException {
    BankEngine engine=getEngine();
    Iterator it=engine.getAccountSubjectToServiceCharge(limit, accountType).iterator();
    while (it.hasNext()) {
        Account account=(Account) it.next();
        BigDecimal amount=charge.min(account.getBalance());
        transfer(amount, account.getId(), chargeAccount, "_bank", null);
    }
}
```

And finally the main method of the application:

```
public static void main(String[] args) throws Exception {
    Bank bank=new Bank();
    bank.transfer(new BigDecimal(250), 1, 1000000, "joe", new Integer(1));
    bank.serviceCharge(new BigDecimal(500), new BigDecimal(5), 1000001);
    bank.serviceChargeEx("checking", new BigDecimal(500), new BigDecimal(5), 1000001);
    bank.serviceChargeEx("savings", new BigDecimal(5000), new BigDecimal(15), 1000001);
}
```

The following sections summarize SQLC capabilities.

Tables

From a table SQLC generates:

- Interface with accessors and (optionally) mutators for all table columns.
- Implementation of the interface (Java bean).
- Select methods, which return all table rows.
- Select method returning single row by primary key (if table has primary key).
- Delete method, which deletes all rows.
- Delete method deleting a row by primary key (if table has primary key).
- Insert method with as many parameters as there are columns in the database.
- Insert method with one parameter of type of generated table's interface.
- Update method with one parameter of type of generated table's interface. The method is generated if table has both primary key and non-primary key columns. The method updates non-primary key columns using primary key columns in `WHERE` clause.

Queries (select statements)

For each query SQLC generates:

- Interface with accessors and (optionally) mutators for all query columns. SQLC tries to reuse/extend already defined interfaces.
- Implementation of the interface.
- Select method, which returns database-backed collection. The primary purpose of the database-backed collection is to iterate over results without placing them all into memory.
- Select method, which places query results into provided collection.

Updates (insert, update, delete statements)

For DML statements SQLC generates one method per statement.

Indices

For an index, which name matches generation policy template, SQLC generates, depending of mode(s) specified:

- Select for all rows ordered by index.
- Select and delete of rows with equal index columns
- Select and delete of rows with non-equal index columns
- Select and delete of rows with positions in the index less than specified in parameters
- Select and delete of rows with positions in the index less or equal than specified in parameters
- Select and delete of rows with positions in the index greater than specified in parameters
- Select and delete of rows with positions in the index greater or equal than specified in parameters.

The reason behind generating methods from indices is that the generated method uses the index to do the operation. This is a mechanism of securing the application from long-running queries running on unindexed columns.

Automatic interface inheritance

SQLC automatically builds hierarchy of generated interfaces. E.g. `AccountType` interface extends `PaymentStatus` interface because both of them have `getDescription()` and `getName()` methods. It may seem insane at first time but don't forget – the interfaces just represent data and do not bear any other semantics. Inheritance is convenient for writing generic data-processing algorithms.

It is also possible to hint SQLC to reuse existing interfaces or to generate interfaces, which extend existing interfaces. It is done by adding `<interface>` element to `<sqlc>` task.

One more feature in interface generation is generation of “common denominator” interfaces. Common denominator interface is generated if there are two or more interfaces, which start with the same word(s) and have common methods. E.g. if there are two queries which generate two interfaces - `PersonUsa` and PersonRussia:

```
public interface PersonUsa {
    int getId();
    String getLastName();
    String getFirstName();
    String getSsn();
}

public interface PersonRussia {
    int getId();
    String getLastName();
    String getFirstName();
    String getPassportNo();
}
```

SQLC will find that

1. Both queries names start with word '`Person`'. Word is defined as a sequence of characters starting with capital letter.
2. Queries have common columns.

And it will generate inerface `Person`. `PersonUsa` and `PersonRussia` will extend interface `Person`:

```
public interface Person {
    int getId();
    String getLastName();
    String getFirstName();
}

public interface PersonUsa extends Person {
    String getSsn();
}

public interface PersonRussia extends Person{
    String getPassportNo();
}
```

Dealing with big databases

If you have a database with hundreds of tables it is not a good idea to generate classes for all of them in one package and one engine. You should compartmentalize you schema into manageable subject areas and generate classes for each area into a different package.

Instead of `<table/>` element you will need to explicitly provide catalog, schema and table names. For example, to generate classes for all tables in the `BANK` schema you'll need to add `<table schema="BANK"/>` to `sqlc` task.

Database standards and generation policy

SQLC uses names of database objects to generate Java classes and methods. With this database standards gain clear purpose – table and field names shall comply with organization's naming conventions not just because pesky DBA's want to show their power, but because it is needed to generate intuitive and comprehensible Java API from the database.

Different organizations have different DB standards. And standard isn't something that can be easily changed. E.g. according to DB standards of the organization where I'm working field `BALANCE` in `ACCOUNT` table shall have name `ACCOUNT_BALANCE_NBR`.

That's OK, but I don't want to have Java property `AccountBalanceNbr`, as it would be generated by SQLC by default. I want it to be `Balance`. The solution is to provide custome implementation of `com.pavelvlasov.sql.metadata.GenerationPolicy`. The easiest way to do so is to subclass `com.pavelvlasov.sql.metadata.DefaultGenerationPolicy`. To solve the aforementioned problem `generateColumnName()` method shall be overridden.

Transactions

SQLC doesn't mess with transaction management. Transactions are attributes of invocation context. E.g. in EJB when you obtain a datasource and then a connection inside a method, which has transactional attribute set, that connection shall already be properly enrolled into a transaction.

SQL execution metrics

One more buy-in for SQLC, dear reader. Compiled classes use `com.pavelvlasov.sql.SQLProcessor` class for all database requests. SQLProcessor class can gather JDBC statistics for you – SQL statement that was executed, how many times, min, max, average and total execution time. To turn on this functionality JVM-wide you need to set `com.pavelvlasov.sql.SQLProcessor:sharedMetricConsumer` system property to the class name of an implementation of `com.pavelvlasov.metrics.MetricConsumer`. There is one readily available – `com.pavelvlasov.metrics.DumpingXmlOnShutdownMetricConsumer`. The class name is self-descriptive. This metric consumer collects metrics and then dumps them in XML format to file on JVM shutdown. Default file name is `sql-metrics.xml`. It can be changed by setting `com.pavelvlasov.metrics.DumpingXmlOnShutdownMetricConsumer:output` system property.

Command-line switch sample:

```
-Dcom.
pavelvlasov.sql.SQLProcessor:sharedMetricConsum
er=com.pavelvlasov.metrics.DumpingXmlOnShutdown
MetricConsumer
```

You can provide your own implementation of MetricConsumer, which, for example, would save statistics to the same database or send to some agent over network. This would obviate bringing the JVM down to obtain statistics. Another option (in server environment) is to have a web page which would display SQL execution statistics.

Metric can also be set only for particular `SQLProcessor`. In JBoss you can expose `SQLProcessor` as Mbean and turn metrics gathering on and off without bouncing the server.

Database and JDK compatibility

SQLC requires JDK 1.4 and compatible JDBC drivers. Drivers shall implement `Connection.getMetaData()`, `PreparedStatement.getMetaData()` and `PreparedStatement.getParameterMetaData()`.

This is a list of databases, which I tested for SQLC compatibility.

Compatible:

- Oracle 9.2 with ORANXO[11] driver
- Hypersonic 1.7.2
- Firebird 5.1
- Cloudscape 10

Incompatible:

- MySql 4.2 – According to MySQL JDBC driver documentation, there is no such paradigm as statement parameter in MySQL engine. So parameterized statements are 'emulated' and implementation of `getParameterMetadata()` would require having parser embedded in MySQL JDBC driver.
- Oracle 9.2 with Oracle's drivers – Oracle's drivers throw exception with the message “Unsupported feature” on invocation of `getParameterMetadata()` method.
- Sun ODBC-JDBC bridge from JDK 1.4 (tested with Oracle 9.2)

Please note that incompatibility doesn't mean that you can't take advantage of SQLC when you use one of incompatible databases. If your statements and database DDL is compatible with Hypersonic you can use `script` element or attribute of `sqlc` task to have Hypersonic provide needed metadata information and then use compiled classes with your target database. I did so with Oracle – it worked just fine.

Compiled classes do not use any JDK-1.4 features and shall be 1.3.x compatible, but I didn't test them on Java 1.3.

Hibernate or Incarnate

This is a philosophical section which ruminates what is primary – data or Java objects? If data is primary then hibernating Java objects is wrong – what we should do is incarnation of data in a convenient way in Java. SQLC does just that.

So what's the answer? The answer is that there is no singular answer, it all depends on a situation. If you have already existing Java classes which you need to store to relational database and amount of data you need to store isn't too big then O/R mapping does its work very well.

If amount of data is huge then it would, most probably, require special attention to database modeling and thus data becomes primary.

Data also is primary in RUP-like development process. Illustration 6 elucidates the point. The process is goes in the direction of lowering of level of abstraction. In other words, each step adds more details.

The process starts with identifying requirements. After that Use Case and Business Object Domain (BDOM) models are produced. Use Case model represents system behavior, BDOM represents system data - subject area entities and their relationships. BDOM objects have attributes but don't have operations. So, in fact, BDOM is a logical model of a database from which a physical model is produced by adding a few details - attributes' types and constraints.

Java class model, on the other hand is a convergence of Use Case model and BDOM. It requires addition of much more details and thus is of lower level of abstraction than database model. The fact that RDMS semantics can be expressed in Java language (pure Java databases like Hypersonic is an example), but not vice versa is another justification of Java having lower level of abstraction than SQL.

Therefore, producing database schema from Java classes is conceptually wrong because it violates the principle of lowering of level of abstraction.

Handcrafting Java classes from database model is also a bad idea. Just duplicating database semantics is a waste of time, as it can be done automatically. Mixing business semantics with persistence semantics is even worse.

This is where SQLC idea comes from – database shall be developed first. Then bridge classes shall be generated by SQLC and only after that database-dealing Java classes shall be developed.

You may say – wait a minute, when SQLC-generated classes incarnate data they produce objects, so SQLC is essentially an O/R mapping tool! No, it is not a mapping tool, it is a bridge generating tool. Mapping assumes that for each table and column table name shall be mapped to a name of already existing class and column name shall be mapped to Java property or field of that class. Typically such a mapping is done in xml files. Another point is that SQLC is not an O/R mapping tool is that object is data+behavior. Objects produced by SQLC-generated classes and representing rows do not have behavior – just data accessors/mutators. Thus they are essentially data structures.

After reading this section one of reviewers remarked: “Who believes in RUP?” I think some people do, but this is not the point. The point is that if your application has just a dozen of business-domain objects and your developers are capable to envisage and code them right off the bat in Java then O/R mapping path is the right one. But if, by whatever reason, you tend to create database model first then SQLC is probably is a better option.

Applicability comparison

By now you got an idea what SQLC is and what it does. As was already mentioned above, SQLC is not an O/R mapping tool. It does not hibernate Java objects to relational database, it incarnates database objects in Java. As such it cannot be compared with, say Hibernate, as M-16 cannot be compared with Hatori Hanzo sword. Both of them use the same basic principle, serve the same purpose, but do it in different ways. What can be compared is their applicability depending on context and person wielding the tool.

Illustrations 2-5 show usage scenarios for JDBC, SQLC and Hibernate. Color codes:

- Red - Java concern.
- Blue - Database concern.
- Yellow – build concern.
- Navy – Database and HQL concerns.

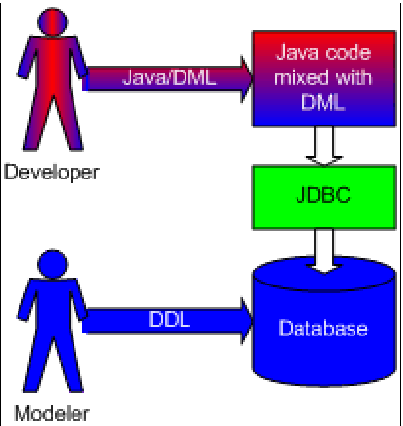

Illustration 2 Separation of concerns in plain JDBC programming

JDBC

Services offered by JDBC are too low-level. Working with database through plain JDBC is like writing XML file using OutputStream – it is possible but such a solution is poorly maintainable and error-prone.

With plain JDBC developer can mistype table or column name, forget to close JDBC resources, issue queries which will run for hours because proper index wasn't created and do a lot of other scary stuff.

Usage of plain JDBC also implies developer's knowledge of SQL and leads to a mix of Java and SQL.

JDBC is the most flexible, all other tools are built on it, but at the same time working with plain JDBC is the most code-intensive approach.

Hibernate

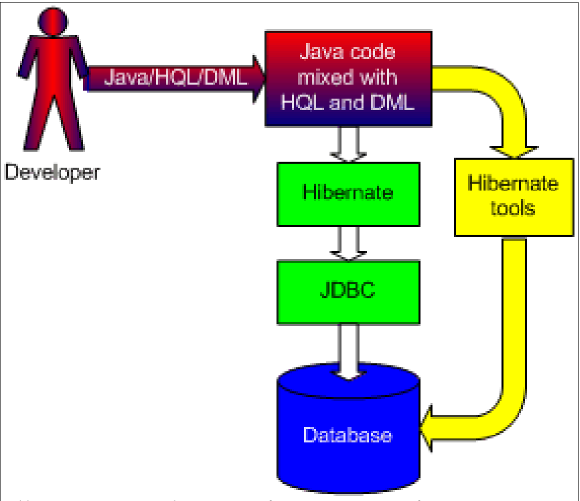

Illustration 3 Absence of separation of concerns in Hibernate.

Hibernate, no doubt, is a great O/R mapping tool. It also has many features which SQLC doesn't address, for example, caching and relationship navigations. Whether to prefer it over SQLC depends on whether your developers are already familiar with Hibernate.

If they are not or if you are going to outsource development then Hibernate can be costly. Hibernate itself is open source and doesn't cost a groat, but Hibernate reference manual is 140 pages and Hibernate book is 400 pages. Effectiveness with Hibernate requires knowledge of other tools like Xdoclet and Middlegen.

In case of in-house developers cost will be in form of low productivity and high level of rework during learning period. In case of outsourcing cost will be in form of higher rate of Hibernate-proficient contractors.

Also note that part of the cost maybe for nothing. For example, specifics of your application may make Hibernate cache not only useless, but dangerous.

Caching works fine if your application is the only one modifying the database. In such a situation roundtrip to the database is required only on data modification (insert/update/delete). This scenario corresponds to commit option A for EJB's.

If there is more than one process updating the database at the same time then the only advantage of caching is reuse of instances (commit option B in EJB).

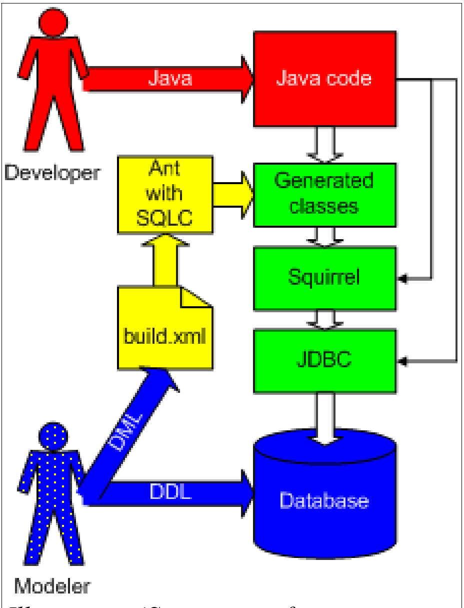

Illustration 4Separation of concerns in SQLC with statements kept in build file.

But in critical applications reuse of instances maybe not acceptable to avoid any chance of manipulating with stale data (commit option C in EJB). This is where caching must not be used at all.

Transparent relationships management is also is not as good as it may appear. Misunderstanding of the concept can leas to a mess. Trust me, I've seen it.

According to [10] "roughly 80% of a typical software system cost occurs *after* initial deployment". Development team is typically gone shortly after initial deployment. Thus skillset requirements apply also to maintainers.

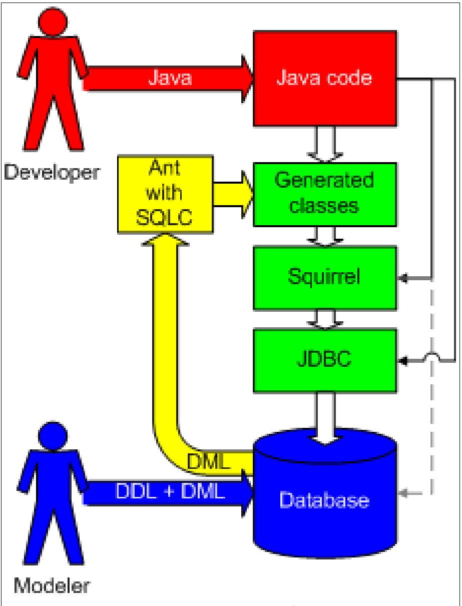

Illustration 5Separation of concerns in SQLC with statements kept in database itself

iBatis SQL Maps

This is another Java-DB tool. iBatis doesn't generate Java classes and methods. It maps database to existing Java classes. So this tool is more about flexibility through indirection (mapping). Developer needs to handcraft Java classes and XML maps. iBatis Java API takes map name as a parameter. Problems like typo in the map name or in the map's SQL or SQL being out of sync with the database cannot be revealed at build time, only at runtime.

SQLC

SQLC is best applicable in environments where

- Databases are created in accordance with database naming standards, which allows to generate self-descriptive names for Java methods and interfaces.
- Database objects are created before Java code dealing with them is written.
- DBA's and data modelers are more permanent than developers.
- Development teams are volatile – one team works on Phase one, another on Phase two and yet other provides support.
- Gartner says only 32% of the 2.5 million Java developers in the world have genuine knowledge, which means there is a serious lack of high-level development skills. SQLC is beneficial for organizations employing not only "smart 1/3" but also those who do not possess "genuine Java knowledge".
- Java application being developed is not required to run on a dozen of different RDBMS with different table and column names. In other words, configurability is not the key factor. It is possible to generate compatible Java classes from different databases, though.

SQLC doesn't cover all data-access scenarios and there will be times when you'll need to fall back to Squirrel[3] or plain JDBC.

Conclusion

Probably the best way to find out whether SQLC is right for you is to give it a try. After that you'll make your choice – to use O/R mapping tools, plain JDBC, SQLC or a mix of them.

Resources

1 Sample code for this article http://www.pavelvlasov.com/articles/sqlc/sqlc-samples.zip

2 SQLC: Manual – http://www.pavelvlasov.com/pv/content/Articles/sqlc/sqlc.html; Download - http://www.pavelvlasov.com/products/Common/pvcommons-2.8.0.zip

3 Squirrel (http://www.pavelvlasov.com/pv/content/Articles/articles.sql.html, http://www.pavelvlasov.com/products/Common/pvcommons-2.8.0.zip) - classes simplifying common database operations and encapsulating JDBC resource management patterns.

4 Antlr (http://www.antlr.org) – Parser generator. Used by code generation classes.

5 BCEL (http://jakarta.apache.org/bcel/) - Bytecode generation library.

6 Hypersonic (http://sourceforge.net/projects/hsqldb/) - 100% Java SQL database.

7 Ant (http://ant.apache.org/) - Java based build tool.

8 Azzurri Clay (http://www.azzurri.jp/en/software/clay/index.jsp) – Database modeling plugin for Eclipse.

9 Hibernate (http://www.hibernate.org) – most popular Java O/R mapping framework.

10 Software Architecture in practice. Len Bass, Paul Clemens, Rick Kazman. ISBN 0-321-15495-9

11 ORANXO (http://www.inetsoftware.de/English/Produkte/ORANXO/default_main.htm) – Fully JDBC compatible fast driver for Oracle.

12 iBatis SQL Maps (http://www.ibatis.com/common/sqlmaps.html) – Java – DB mapper.

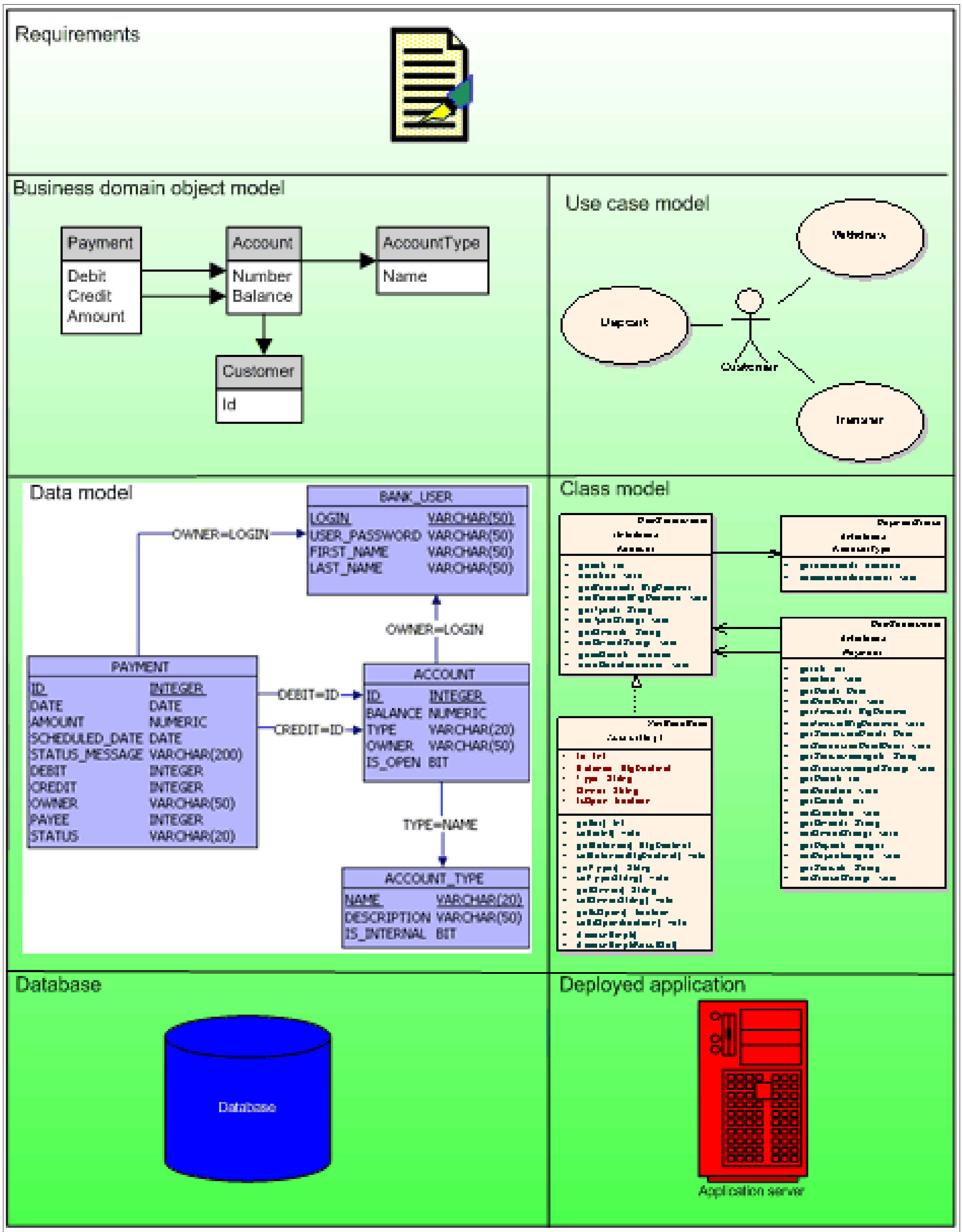

Illustration 6Development from requirements to deployed application