From Start To Finish: Designing and Implementing a Database in SmallTalk
By: Sweta Vajjhala & Brian Stebar

INTRODUCTION

When designing a project, there are many different steps involved. These steps include planning, implementing, testing, and then finally releasing the application. For this particular project, a stock market simulation called StockTrader, which was coded in SmallTalk, various milestones were set to complete different parts of the process. In StockTrader, a database was used, in conjunction with SmallTalk, to keep track of information in the system. This case study thoroughly describes how a database is used in each step of the process and include step-by-step instructions for setting up the database and its complements (i.e. the database server), as well as recommendations for using a database with SmallTalk.

PROJECT DOMAIN ANALYSIS, CRC CARDS, SCENARIOS (M4)

Process
This phase of the project is the beginning of the design phase. While designing this project, it was important to consider how to manage the database. There were two basic options:

1. Have a singleton design pattern for the database, which keeps the database connection open constantly, or
2. Each class would have its own database connection code.

This implementation of StockTrader uses the first option. The second option did not seem viable, because there would be a lot of code duplication, if the connection code had to be replicated in every class. However, the connections were not the only part of the StockTrader system that were affected.

Having a database also removes the need for any OrderedCollections of objects. For example, storing histories of stock price changes is stored in a table in the database, rather than as an OrderedCollection. Instead, each history is represented as a separate object that represented an instance of the history. This object is then inserted into the database. This also changes the direction of dependence relationships between histories and investments. Without a database, the investment would have to hold all of its histories. With a database, however, the history knows its investment through the use of foreign keys, and the histories do not need to be maintained in memory.

Each investment is also associated with an InvestmentPurchase object. With a database, each portfolio is an OrderedCollection of investments and the quantity of each investment. Without a database, the Investments themselves would do this job. By using a database, it is important to consider how it maintains its relations. For example, the database cannot store an OrderedCollection of references into a row in the database, without some complicated string parsing and object processing methods. Therefore, the InvestmentPurchase object is created as a separate object to link the investment to the portfolio in the database.
Using a database with StockTrader also handles any loading and storing that may have come up. Instead, there is a table in the database called "settings", which stores all of the necessary information pertaining to the current system.

While designing the database, the goal was to ensure that a StockTrader user would not know that a database was being implemented: it should be in the background. Therefore, role-playing scenarios were not affected by the addition of a database.

**What Worked**
By the end of this phase, the classes that were needed for StockTrader were all set. Moreover, through the course of the project, none of the classes got deleted, but a few were added.

**What Did Not Work**
At this point in time, there was not a very good understanding of how SmallTalk works with the database.

**Quick Tips**
Consider implementing a database. If the development team decides that this is a good idea, then make sure to consider it from the very first step. If it is not considered right away, transitioning into using a database is almost impossible.

**FINAL PROJECT, DETAILED DESIGN, UML, ARCHITECTURE (M5)**

**Process**
After designing everything thoroughly for the first time, a final run-through was needed before implementation began. In this stage, the StockTrader system changed dramatically from the original design.

The original design for the database did not take into account storing and generation of queries. For this reason, two different categories of queries were developing: INPUT and SELECT. Queries of type INPUT include any SQL statements, such as DELETE, INSERT, or UPDATE, but not SELECT statements. SELECT queries are handled differently. The reason for this is because the INPUT queries never return data, but return the number of rows affected (numRowsAffected). The SELECT queries return the data itself. Moreover, INPUT queries must be generated in instance methods, while SELECT queries require no data from instances and are used as class methods.

**What Worked**
It was impossible to do SELECT queries without class methods, so this integrated very nicely. Moreover, this design allowed all of the code to be segmented properly, without very much repetition, and refactoring was not needed.

**What Did Not Work**
The design had to be drastically changed from the last milestone, and the UML diagram had to be redesigned when accounting for the database. This is because using a database adds an extra trust boundary, and this was not taken into consideration during the preliminary design process. Moreover, this also completely changed the sequence diagrams.

**Quick Tips**
Although there was a general idea of the sequence of method calls, the preliminary design did not have specific details on which step corresponded to which classes. Therefore, it is
important to go through lots of different scenarios, in order to make sure that every part of
the database is included in the final stages of the design.

**DOMAIN OBJECT CODING AND UNIT TESTS (M6)**

**Process**
There are multiple steps involved in implementing the planned design.

*Installing an ODBC-Compliant Database Server*
In this example, two different distributions of MySQL were used. Most of the work was
done using WAMPserver, but XAMPP was also successfully implemented. In this example,
WAMPserver was installed using all of the default settings. To make console access
easier, it is helpful to set a path to the database's binaries in the system's environment
variables. Instructions on how to do so will be given for both Windows XP and Vista users.

1. Click the 'Start' button, right-click 'Computer' (Vista) or 'My Computer' (XP), and
   select 'Properties'.

   ![Screenshot of 'Computer' context menu](image.png)
2. Click the 'Advanced system settings' link on the left navigation pane (Vista) or the 'Advanced' tab (XP).
3. Click 'Environment Variables'.

4. In the list labeled 'System variables', highlight the row labeled 'Path' and click 'Edit...'.

5. At the end of the field labeled 'Variable value:', append the path of the database server's binaries, separated by semicolons. (Ex: ...;c:\wamp\mysql\bin;...)

6. Click 'Ok' on all windows to save and close.

**Installing an ODBC Connector/Driver**
Windows users will find that some ODBC drivers are already bundled with the operating system. However, MySQL users will have to download the connector for their system from MySQL's website: [http://dev.mysql.com/downloads/connector/](http://dev.mysql.com/downloads/connector/).

**Configuring a Data Source**
When a data source is configured, it allows any ODBC client (in this case, SmallTalk) to connect to the database. To access Windows' data sources, the Administrative Tools must first be added to the programs menu. To do so:

1. Right-click the 'Start' button, and select 'Properties'.

![Properties menu.png](attachment://Properties%20menu.png)
2. Click 'Customize...'.

3. In Windows XP, click the 'Advanced' tab.
4. Near the bottom of the list under 'System Administrative Tools', select one of the options for displaying the tools.

5. Click 'Ok' on all windows to close.

Before configuring a data source, make sure the database server is running. To configure a data source:

1. Click Start -> Programs -> Administrative Tools -> Data Sources (ODBC).
2. Click 'Add...'.

3. Choose the ODBC driver for the appropriate database system and click 'Finish'.
4. Fill in the fields for the database server. The only required fields in this example are 'Data Source Name' (can be anything), 'Server', 'User', and 'Password'.

5. Click 'Test' to ensure connectivity.
6. Click 'Ok' on all windows to close.

The data source name will be the key for SmallTalk to access the database.
Installing ODBC Support in SmallTalk

Luckily, Cincom SmallTalk comes equipped with a package for interfacing with ODBC-compliant database systems. To set this package up:

1. From the main VisualWorks window, open the Parcel Manager.
2. In the Database category, load the package titled 'ODBCEXDI'.
3. Close the Parcel Manager.
Implementing a Database Connection Manager

Opening and reopening a connection to a database is very costly. To avoid this, a class was created to manage the connections to the database. This class is then instantiated as a global variable. This ensures that for the entire time that the application is use, only one connection to the database is ever created. The only overhead comes from executing individual queries. From the CLASS new method:

Notice that the 'environment' parameter MUST MATCH the data source name (DSN) for the ODBC connector.
The database manager has only two methods. The first is `setQuery`: which returns a two-dimensional array of strings containing the rows of data resulting from the given query.
The second method is `inputQuery`: which handles UPDATE, DELETE, and INSERT statements. Since all of these query types return the same type of value (an integer representing the number of rows affected), this method can accommodate all three query types.
Embedding Database Code in Objects
To illustrate, focus will be put on the User cluster of classes from the StockTrader system.
Each object instantiation has methods titled `insertQuery`, `updateQuery`, `deleteQuery`, and `getIdQuery`. Each of these queries is dynamically generated using data from the object's instance variables. For example, when a user is created and its instance variables populated, `insertQuery` is used to generate the query to send to the database manager's `inputQuery`.
It is important that after an object is inserted into the database, its primary key must be retrieved and stored in the object. Since the primary key is not returned when an INSERT statement is performed, the **getIdQuery** fetches it.
updateQuery and deleteQuery statements also use inputQuery for execution. Note the reliance on a database ID.
Retrieving Objects from a Database
SELECT statements are handled differently from the others. Since no instance of the desire object has been created yet, select methods must be CLASS methods.
Notice the call to a method name `buildUserFromRow`. Since `selectQuery:` returns data as a 2D array, it is convenient to implement a builder function to create SmallTalk objects from the data.

Using such a method will return easy-to-use Collections of Smalltalk options instead of abstract 2D arrays of data.

When implementing database code for individual objects, make sure that SUnit tests are created that test more than just object creation. Tests need to be written that ensure that a fresh instance of the object can be successfully retrieved from the database.

**What Worked**
Using the singleton design pattern for the database manager made database access relatively easy. SUnit tests ensured that code changes had far fewer unexpected side effects. Having universal input and select methods minimized code duplication. Since each class was responsible for generating all necessary queries, calls to the database could be
performed using only one line of code. Object builder methods eliminate the need to manipulate raw data from the database. Because database systems can easily handle multiply connections, this design can easily be extended to accommodate several user sessions simultaneously as long as each client has a working ODBC connection.

What Did Not Work
StockTrader's implementation of tracking price histories for investments did not record purchases or sales when something was bought or sold, causing some inaccuracies in reports. Expanding the schema becomes very difficult after much code is written. SmallTalk's Timestamp did not have a conversion method to an SQL timestamp, so writing one became necessary for tracking price changes.

Quick Tips
Assign instantiations of database manager to global variables to ensure that only one connection is opened. Make sure that globals are only used in a custom namespace to avoid conflicts with any existing SmallTalk globals. Try to minimize the number of database hits. If two SQL statements can be combined into a more sophisticated one (i.e. by using JOINs), do so. Executing queries is expensive. Go through scenarios again to ensure that schema design is accurate. It is much harder to change later in development. Run SUnit tests often during development to catch any strange bugs.

COMPLETE UI AND APPLICATION (M7)

Process
Implementing a Graphical User Interface (GUI) is the final step of finishing the functionality of StockTrader. During this phase, there was no need for any special considerations with relation to the database. If the domain model was designed properly, it should have included a Facade class. This class acts as a controller--the link between the domain model and the application model. The GUI will have to interact with this class only, by calling all of the domain model methods it needs through this class. Therefore, there does not need to be any database-specific code in this phase.

What Worked
Having a DomainModelFacade class really worked well. Since the GUI interacts solely with this class, it is impossible for an outsider-user to tell if a database has been implemented or not. Moreover, this database allowed load/store functionality to be instant. A settings table in the database was useful for storing program settings. The database also eliminated the need for passing lists between windows, making the system more memory-efficient and faster.

What Did Not Work
Because a database was implemented, the number of methods that have to be called through the facade became a lot higher. This is because many of the domain method that interacted with the database mapped directly to methods in the facade.

Quick Tips
Using global variables to store current values (such as the current user, portfolio, and investment) will make it much easier to pass this information between windows.
USER INTERFACE EVALUATION (M8)

Process
Since this StockTrader implementation did not receive any feedback, another test run would be required to find or fix any problems.

Quick Tips
Xerox provides a heuristics evaluation which is not only thorough, but very useful if it is completed. This checklist can be found at: http://www.stcsig.org/usability/resources/toolkit/hecklst.doc.

DESIGN SURPRISE TWIST - WEB FRONT END (M9)

Process
Because a database was implemented, less session variables were needed when making StockTrader a web application. The only session variables that were needed were to keep track of current information, such as the current user, portfolio, or investment. Moreover, rather than storing all of the different transactions that were taking place in request or session variables, the system can easily gather/store these values from/in the database.

What Worked
The implementation of a facade class, once again, made designing the web application much easier. Very little code rewriting was necessary. The database implementation almost eliminated the need for session attributes.

What Did Not Work
There did not appear to be any problems or design conflicts that occurred for this stage.

Quick Tips
Use session variables sparingly, but wisely. Objects can be stored as session attributes, but not as request parameters. (Request parameters can only be strings.)

CONCLUSION

Using a database may drastically change the design of a system. However, if it is designed properly and similar to the steps shown above, the database becomes a really handy tool for loading, storing, and retrieving information. There may also be a lot of unexpected changes in the design, scenarios, and schema, but this case study will hopefully help in making some correct decisions from the very beginning of the design of the system.