Supply Chain Project

Final Report
03/06/2003

E-Commerce Technologies
CSC513
Section 001
-
Spring 2003
Dr. Singh

Team Members:

Omer Ansari
Ryan Catherman
David Tran

Table of Content

#1 [(40 points) Describe in two pages the problem you are solving along with any assumptions. Describe the main processes and how they interplay in this scenario.][1]

This deliverable is broken up into three parts:

- Project Description, brief understanding of the goals addressed in this project
- Design Details, a top level overview of the design of the software, program flow, and various interfaces between the components of the project
- Assumption: some assumptions taken while designing the project.

Project Description

This project focuses on design and implementation of a supply chain management system. It introduces the concept of local as well as recursive ordering. Although these two concepts are not directly related, the system attempts to abstract the source of the order such that a local order appears the same as a remote order from another vendor.

Four vendors are involved. They are listed from upstream to downstream direction:

- Air Conditioner – In the source code this may be noted by 'C' or 'AC'
- Hose-cum-valve Assembly – Known as 'A' or 'Assm'
- Hose – Known as 'H' or 'Hose'
- Valve – Known as 'V' or 'Valve'

Each vendor will make decisions based on policy as to how it will react to a requested order. Although orders may come from both a remote vendor or a local request, perhaps via a web GUI, the underlying decision-making does not know the origin. The specific policy description for decision-making is found in the EntitySpecificContainer, also referred to as the DBEngine, description later in this document.

Each Vendor also maintains a list of customers whose information is entered locally, via the Web GUI.

A quality control process is maintained by the software, and controlled by the GUI where bad items are purged from the inventory, and the good items get a stamp of approval.

This software is written in Java. JDBC is used to interface with the database, while JSP is used for the web interface. Tomcat is used as the web server. GNUJAXP is used for XML parsing.

[1] Details from the project description page:
Architecture--You have to identify the following in the whole picture of your project:

- the entities in the supply chain
- what roles they are playing
- what functions they have
- How they interact with each other and how they are related to each other

Design Details

There are three basic entities comprising a vendor. Below shows where the respective code[2] lies:

1. GUI Interface [web/ **(JSP/HTML)** ; src/GUIClasses/ **(package)**]
2. XML Interface [src/XMLEngine/ **(package)**]
3. Database Engine [src/EntitySpecificContainer/ **(package)**]

Fig 1.1Simplified High Level Architecture

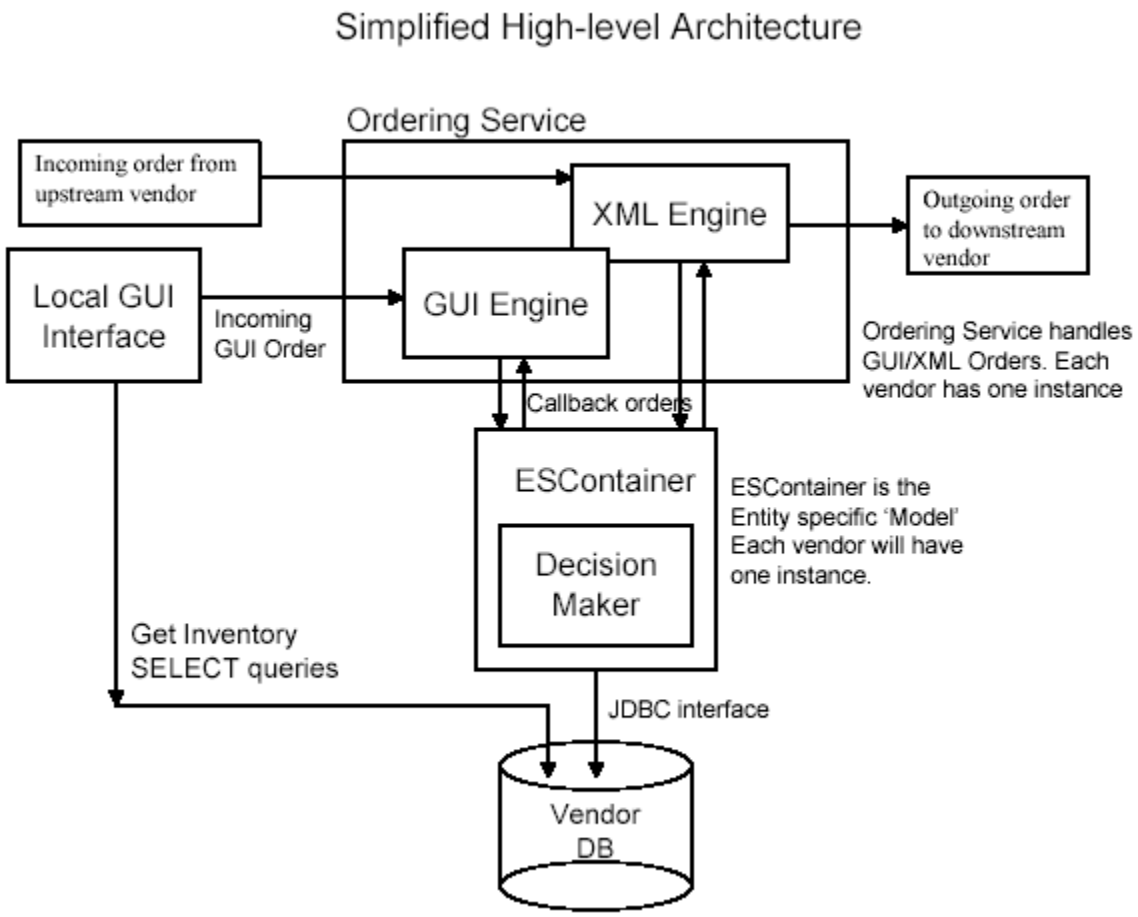

Fig 1.1 explains the design of a typical vendor. Order can be received from the GUI as well as XML. The GUI orders are placed locally, while the XML only come from the upstream vendor.

The software flow goes like this:

src/ScmMain is the main process that accepts, H,V A or C as an argument, and starts the respective engine. It then does the following

- Instantiates the XML Order Placer class (OrderPlacer op).
- Retrieves the singleton instance of ESContainer, the DBEngine.

[2] This is the reference point: http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/rodx-scm/ProjectOne_/

- Instantiates GUIOrderReceiver (GUIOrderReceiver guiServer), runs it as a thread, and passes the respective TCP port[3] (to listen for GUI orders on) and the dbEng instance previously created.

The GUI handles three separate functions

1. viewing current inventory
2. placing order
3. running quality control process.

The respective jsp/html code for each vendor is under web/. Thing to note is that GUIOrderPlacer class is instantiated within each vendors order placer jsp (e.g. HoseOrderPlacer)

Incoming GUI orders use this class to talk to the guiServer. The order placed is via XML over TCP/IP sockets, on which the guiServer is listening. The order is taken in parsed. Each item is then placed into a data structure (src/XMLEngine/OrderDataFormat.java), and is added to a vector which is then passed into the dbEng. The dbEng then make a decision if it has enough and how to acquire more parts to complete the order.

The currently implemented policy for the dbEng is as follows. The module looks locally to see if the number of items required exists. If there are enough items in the local DB, then they are subtracted and the order will pass. If there are not enough items, then the module will callback attempting to obtain enough parts to build the rest of the order. The module will only order what it needs to complete the order. When the parts are received and new items are built, they are immediately checked for quality. The module will iterate with the remaining 'good' parts until the order is completed or an error occurs.

When items are built and checked, if they fail quality control, they are placed into a repository until they are purged with the required quality control implementation.

An XML Order[4] is received and parsed from a downstream vendor the same exact way as a GUI order, and then handed to that vendor's dbEng , which makes a decision if it has the items locally or not and the above process is repeated, until the last downstream vendor returns.

The dbEng process uses JDBC to communicate with key.csc.ncsu.edu's oracle database. All writes are made via the dbEng, while some read-only queries (like acquiring inventory) are allowed and done directly from the JSP.

The above discussion is from a java package level. Details within each package would be too complex to summarize within 2 pages. For that, please read the comments within the source code or ask the developers for details.

[3] src/XMLEngine/XMLEngineLiterals.java contains the respective tcp ports for both the XML and the GUI order receiver for each vendor

[4] the XML file used for ordering etc is defined in Appendix A.

Assumptions

1. In order to run this, a tomcat server is required. build.xml can be used for installing the project
2. gnujaxp.jar[5] and classes12.zip[6] need to be put into src/lib
3. While placing an order, an order automatically confirms if the requested items are available, and automatically cancelled if even 1 of the items is not available. The user is not prompted to confirm/cancel.
4. In the tasks for group of three, this is requested*: Say one customer needs large amount of certain types of products, you should increase your inventory properly to meet the needs*. Our interpretation of this is to provide a GUI user to add items to any vendor's inventory, which we have thus done.
5. It was not requested to tabularize the relation of a product (e.g. 1 small assembly) and its components (e.g. 1 small valve + 1 small hose). Thus we have hard coded this within our EntitySpecific/*Specific.java files

[5] http://www.gnu.org/software/classpathx/jaxp [I did put this in the CVS repo b/c its small enough]
[6] http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html

#2 (80 points) Create base relations with the right attribute domains. Populate the base relations with 4-8 rows each. Show what is in each table by printing out a "SELECT * FROM table" for each table.

When deriving these relations, it is worth noting that although some of the tables could be combined, this did not fit the model. An example is that all inventories could be combined, but the model used is that each vendor would really have its down database and so they should not have access to the other data.

All tables are listed in src/tables. Here is data, some sample, from each table:

Price Tables:

SQL> select * from ac_price;

```
       PID      PRICE
---------- ----------
       400     105.51
       401     155.24
       402     174.68
       403     254.79
```

SQL> select * from hose_price;

```
       PID      PRICE
---------- ----------
       200      10.55
       201      15.25
       202      17.34
       203         25
```

SQL> select * from valve_price;

```
       PID      PRICE
---------- ----------
       100      10.11
       101       18.2
       102      19.87
       103         34
```

SQL> select * from assm_price;

```
       PID      PRICE
---------- ----------
       300      56.55
```

301	65.25
302	77.34
303	99.99

Inventory Tables:

SQL> select * from ac_inv;

PID	QUANTITY
400	0
401	0
402	0
403	0

SQL> select * from hose_inv;

PID	QUANTITY
200	7
201	200
202	2188
203	1196

SQL> select * from valve_inv;

PID	QUANTITY
100	7
101	529
102	1060
103	1099

SQL> select * from assm_inv;

PID	QUANTITY
300	13
301	44
302	664
303	33

Customer Tables:

SQL> select * from ac_cust;

```
      CID
CUSTOMERINFO
--------------------------------------------------------------------------------
        1
Sahara Networks Inc
        2
Sayanora Designs Inc
        3
SleepyTown Computers Inc
        4
Penchant Software Co.
```

SQL> select * from assm_cust;

```
      CID
----------
CUSTOMERINFO
--------------------------------------------------------------------------------
        1
AseemblerCoders Inc.
        2
Wasif Musty And Bros.
        3
Basf Networks Co.
        4
TownHall Corp.
```

SQL> select * from hose_cust;

```
      CID
----------
CUSTOMERINFO
--------------------------------------------------------------------------------
        1
Veritas Assemblies Co.
        2
Verizon Assemblers Co.
        3
Mitchells Fruit Farms Ltd
        4
Verification Equipment Ltd.
```

SQL> select * from valve_cust;

```
      CID
----------
CUSTOMERINFO
--------------------------------------------------------------------------------
         1
Volvo Inc.
         2
Richard Pybus Attorneys Inc.
         3
St. Marys Schools Inc.
         4
Foundry Networks Inc.
```

Quality Control Tables:

```
SQL> select * from hose_bad;

       PID   QUANTITY
---------- ----------
       200         30
       201          6
       202         16
       203         58

SQL> select * from valve_bad;

       PID   QUANTITY
---------- ----------
       100         63
       101          8
       102          8
       103        108

SQL> select * from assm_bad;

       PID   QUANTITY
---------- ----------
       300         18
       301          8
       302          8
       303         13
```

```
SQL> select * from ac_bad;

       PID   QUANTITY
---------- ----------
       400          3
       401          5
       402         19
       403         10
```

#3 (240 points) Write interactive SQL queries for each operation, and test via a terminal interface. The idea is that as the scenario progresses, the different steps in the process will execute based on the appropriate databases. These SQL queries will provide the basis for correct decisions. Turn in the results for each query. Create enough tuples and choose values for the constants so that a non-empty answer set is returned. Explain why each SQL query is correct.

Queries made within the GUI JSP: **The results of the below queries are not displayed because these are the same queries displayed in #2 above.**
[only Read type queries are made here]

GetAndAddInventory.jsp and GetInventory.jsp:

String query = "SELECT PID,QUANTITY FROM HOSE_INV ORDER BY PID";
String priceQuery = "SELECT PID,PRICE FROM HOSE_PRICE ORDER BY PID";
String badItemsQuery = "SELECT PID,QUANTITYFROMHOSE_BAD";

The above is an example from only the HOSE vendor, but applies to all others for the jsp. This JSP is responsible for displaying the current inventory of the entries both in the good lot (which have stamp of approval) and the bad lot (which have failed the quality control process) along with their prices I have entered the PIDs in the same order in both table types, so by using ORDER by PID I ensure the right price is lined up with the right product.

HoseOrderMenu.jsp:

String priceQuery = "SELECT PID,PRICE FROM HOSE_PRICE ORDER BY PID";
String getPriceQuery = "SELECT PID,PRICE FROM HOSE_PRICE WHERE PID IN (";
<selected PIDs>)";

The above is an example from only the HOSE vendor, but applies to all others for the jsp. The first select statement is just to show the price to the customer ordering that product. The second query is a dynamically created query and is generated when certain product Ids are selected for the order. This query gets constructed by the code accordingly, and is displayed at the end in the order results.

The EntitySpecificContainer, or DBEng, uses only a handful of commands to manipulate the database in each vendor. Since the vendors are all built on the same framework, the SQL statements are all very similar. The construction of these dynamic statements can be found in DBHelper.java.

Adding a Customer String:

The query is built with several variable parts that are resolved depending on the vendor type.
The statement for the HOSE vendor is as follows:

"INSERT INTO hose_cust(cid, customerinfo) values((SELECT COUNT(cid) FROM hose_cust) + 1, ***customer string ***)"
The ***customer string *** is the information to store.

The statement uses a nested query to setup the primary key, cid, and then adds the new information using the INSERT command.

Where the hose-specific names are used, the other vendors' names would be substituted.

Checking a quantity by PID:

"SELECT x.quantity FROM host_inv x WHERE x.pid = ***"
*** shows where the PID is a variable for the query.

This statement is changed for each vendor to reflect their respective tables. Also, the table name could be changed to reflect a query to the bad stock table or any other table as long as the attributes remain the same. This meant that the code only needed to be written for this scenario with variable names that could be dynamically inserted.

Modifying a quantity in a table by PID:

"UPDATE hose_bad SET quantity = x WHERE pid = p"
In this statement both x and p are the variables to the constructor of this statement. Also, hose_bad is a variable where any vendor table can be inserted to fit the need of the caller.

These three statement types, and their permutations, are all the EntitySpecificContainer, or DBEng, require for this portion of the project. It was decided that transactions would not be used at this time, so the assumption is that all 'write'activity will go through this singleton instance at this time.

#4 (320 points) Write application programs to carry out the various processes and to accommodate the situations discussed above. These programs should use embedded SQL. They should check the inputs for errors. Your applications should be as robust as you can make them.

The complete source code is available from sourceforge CVS from:

http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/rodx-scm/ProjectOne

#5 (120 points) Demo. Graded on functionality & robustness, and ease of use of your programs. You are required to implement a graphical user interface (GUI), e.g., using HTML and CGI, or Java, or any other suitable approach.

Demo has been scheduled for Saturday - March 22, 2003 at 10:00 am in TAs office (Venture 1 – room 103)

#6 [(80 points) Describe what it will take for one of your vendors to interoperate with a vendor implemented by another team.]

Generalized steps of inter-operating between vendors made by other group projects

The following issues need to be dealt with so that our vendor can interoperate with another group's vendor

1. First, a mutual agreement of the attribute specs on the xml format to be used between the vendors.

2. The transport has to match on both sides. i.e. clear-text over TCP, or SOAP, or something else.

3. An understanding of returned error codes must also be finalized. e.g. how to tell the upstream customer that items are not available, or available

4. A general agreement of the algorithm on how to first find if items are present (read-only) and then commit/cancel the order (write/commit) . e.g. in our case we are using the orderStatus attribute in the xml data to identify the read-only checks (itemavailable, itemNotAvailable) and in the 2nd pass, tell the vendor to commit using (commit, cancel) write signals.

5. Knowledge and agreement on the PIDs (product IDs) contained in inventory by downstream vendor, and used in orders by upstream customer (placing an order with the downstream)

Example:
ASSEMBLY = 300, 301,302,303,
VALVE = 100,101,102,103

If there was a disconnect in the above vendor, ASSEMBLY order for 301 might have been asking for 105, 106, and that would obviously not work.

Also this needs to hold true if the upstream customer (ASSEMBLY) is also a vendor to another upstream customer (AC)

6. Theoretically the 2nd level of downstream should also have the same agreement of PIDs as #5. although, it would work otherwise too, in real life, both the AC vendor and the ASSEMBLY vendor would need to agree that e.g. 1 small assembly = 1 small hose + 1 small valve.

Appendix A - References

1. **Principles of Database Systems with Internet and Java Applications by Greg Riccardi, Addison Wesley, 2001 (ISBN 0-201-61247-X)**

2. **Just Java 2 (4th Edition) by Peter Van Der Linden, Sun MicroSystems, 1999 (ISBN 0-13-010534-1)**

3. **Java in a Nutshell by David Flanagan, O'Reilly & Associates Inc. 1999 (ISBN 0-56-615021-0)**

4. **TAs' Page for CSC513: http://courses.ncsu.edu/csc513/lec/001/wrap/**

5. **http://java.sun.com**

6. **http://jakarta.apache.org**

7. **http://sourceforge.net**

8. **XML File Format**

```
<?xml version="1.0"?>
<Order>
  <Vendor>2</Vendor>
  <CustomerInfo>Foundry Networks Inc. </CustomerInfo> ←A
  <Item>
          <ProductType>200</ProductType>
          <Quantity>50</Quantity>
          <OrderStatus>1</OrderStatus> ←B
  </Item>

  <Item>
          <ProductType>400</ProductType>
          <Quantity>20</Quantity>
          <OrderStatus>1</OrderStatus>
</Order>
```

(A): This field is only used in the GUI Order. It is left blank in an XML Order.
(B): The possible values for OrderStatus are:
1 item check <---*
2 item available <--**
3 item not available <--**

11 item confirm <--*
12 item cancel <--*

13 item confirmed <---**
14 item cancelled <---**

* = set by customer (.e.g ASSM)
** = set by vendor (e.g. VALVE)

e.g. ASSM places two orders

ASSM--> valve(quantity=145, orderStatus=1), hose(quantity=145, orderSTatus=1)

valve responds back -> orderStatus=3
hose responds back -> orderSTatus=2

both valve and hose understand that orderSTatus=1 meant they only reply with 2 or 3, they don't have to remove items from their inventories as yet.

ASSM, receives valve/hose orderAcks, it issues:

ASSM--> valve(orderstatus=12) hose(orderstatus=12)

valve and hose both send back orderstatus 14, and do nothing with their DB.

say if valve and hose received orderstatus=11, they would have updated their db's and replied back with orderstatus=13.