A FrameWork for Building Financial Models

Email: yoder@cs.uiuc.edu

Table of Contents

1. INTRODUCTION 1

2. FINANCIAL MODEL SPECIFICATION 1

2.1 DuPontModel 1
2.2 Drilling-Down 3
2.3 Detailed Transactions 3
2.4 Summary Reports 4
2.5 Graphs 5

3. THE ARCHITECTURE AND DESIGN OF THE FINANCIAL MODELING TOOL 6

3.1 What happens in the Financial Model Application 6
3.2 Financial Model Architecture 8
3.3 How it all fits together 9
3.4 How to Specify a FM 11
3.5 ReportValues 13

4. DETAILS OF THE DESIGN 19

4.1 Creating DuPont Models 19
4.2 Business Logic and GUI Specification Data Model 20
4.3 ReportValue Details 22
4.4 ValueModel 24
4.5 Queries 25
4.6 QueryExpressions 28
4.7 Selection Criterion 30
4.8 Specifying Selection Criterion 32
4.9 FMState 34
4.10 ApplicationInfo 34
4.11 Sessions 34
4.12 Namespaces 35
4.13 SummaryReport Framework 36
4.14 Graphing Framework 36
4.15 DetailedReport Framework 37
4.16 Printing Framework 38
4.17 Testing Framework 38

5. SECURITY MODULE 38

5.1 Security Requirements 38
5.2 Components 39
5.3 How security requirements are met 39
5.4 Other Requirements 40
5.5 Security Data Model 40
5.6 SecurityAdmin Tools 41
5.7 FMLogin Process 42

6. FUTURE WORK 48

7. SUMMARY 48

8. PATTERNS 49

9. REFERENCES 50

10. APPENDIX - THE FARM DATA MODEL 50

Figures

Figure 1 - DuPont Model 2
Figure 2 - Drill Down on Inventories 3
Figure 3 - Prime Products Inventory Detailed Transactions 4
Figure 4 - Prime Products Inventory Summary Report 4
Figure 5 - Prime Products Inventory Graph 5
Figure 6 - Value Models in GUIs 7
Figure 7 - Financial Model Application Overview 7
Figure 8 - User View of the Financial Model Layered Architecture 9
Figure 9 - Builder View of the Financial Model Layered Architecture 10
Figure 10 - Financial Model Creational Diagram 11
Figure 11 - What Happens 12
Figure 12 - *ReportValue* 13
Figure 13 - Drill Down Values 14
Figure 14 - *ReportValues* GUI Specifications 15
Figure 16 - *ReportValues* Business Logic Specification 16
Figure 18 - ReportValues Creational Diagram 17
Figure 20 - Structural Diagram of ReportValues 18
Figure 22 - Business Logic and GUI Data Model 20
Figure 23 Business Logic and GUI Entity Relationship Diagram 21
Figure 24 - Object structure diagram for ***ValueModel*** 25
Figure 25 - Dynamic structure of *QueryObjects* 27
Figure 26 - Object diagram for *QueryObject* 27
Figure 27 - *QueryExpression's* object diagram 28
Figure 28 - Dynamic structure of *QueryExpressions* 29
Figure 29 - Aurora Selection Box 30
Figure 30 - Specifying Selection Criteria 31
Figure 31 - *SelectionCriterion* Structure 32
Figure 32 - *SelectionCriterionEditor* Editor 33
Figure 33 - Security Data Model 40
Figure 34 - Security Admin 41
Figure 35 - UserPropertiesDialog 42
Figure 36 - Login Process 43
Figure 37 - FMLoginDialog 43
Figure 38 - Login Failure Loop 44
Figure 39 - Application Setup 45
Figure 40 - Other Security Checks 45
Figure 41 - Node security 46
Figure 42 - Profile Security 46
Figure 43 - Role Security 47
Figure 44 - Password Security 47
Figure 45 - Financial Model Start Up 48
Figure 46 - Farm Data Model 52

1. Introduction

We've been working for over two years on a fairly large financial modeling project with Caterpillar. Our main result is a framework for financial modeling. It lets you quickly build applications that examine financial data stored in a database and produces profit and loss statements, balance sheets, detailed analysis of departments, sales regions, and business lines, with the ability to drill down until you hit individual transactions. It lets you budget and correct errors in the data, too. We believe that once the business specific data model has been built (which currently takes a long time in Caterpillar because most of their data has to be taken from many sources, and every business unit is different and so has to define its own data model) that it will only take a week or two to build the rest of the system. We are able to define a complete system for any business unit without any Smalltalk programming. The business model is stored in the database, not written in Smalltalk, and we plan to have a GUI for building it. The menus, reports, drill-downs, and graphs are also stored in the database. We have also implemented GUIs for defining all of the GUI's and business logic.

The financial modeling framework allows one to build a Financial Model application for nearly any type of business. Although it was hard to figure out a good architecture for building Financial Models, the actual architecture for the Financial Model is not complicated and can easily be reproduced. Most importantly, the architecture is language independent. It can be implemented in any language, and once implemented, knowledgeable users can build domain-specific applications without programming. This allows domain experts to design the applications and allows end-users to customize them.

The primary principle here is to not program in a general purpose language, rather program in a higher-level domain-specific language. You can do more with a general purpose language but with the higher level language you can program within a limited domain with fewer lines of code. This work provides a visual-object-oriented programming language [Burnett, Goldberg, Lewis 1995] to allow one to program without feeling like they are programming. They are designing the application in terms of domain specifics that they are familiar with. [reference Book on Visual Programming by Goldberg here].

2. Financial Model Specification

A Financial Model is an transactional-based application that allows multiple users to edit and view different financial aspects of a business. It is used to analyze financial data and to support business decision making. It provides a top-level profit/loss statement; the ability to drill-down to view any desired summary reports, graphs, or detailed transactions; a way to update and correct individual transactions; and a way to dynamically create new reports of interests.

The application is client-server oriented and allows users to collaborate during business decision making. Customizing and selecting views of interest is built into the system, thus the users can interactively decide the desired business logic and the view on the data.

The system creates a top-level view of the model depending on the user, typically a ***DuPontModel*** or a Profit and Loss Statement. The user can select the desired date range and products that they wish to view. They can also drill down to see how a particular value is computed. These drill-down reports also allow for many customized summary reports, graph reports, and detailed reports.

2.1 DuPontModel

The ***DuPontModel*** [Johnson & Kaplan, 1987] (see Figure 1) is a graphical model of a view of Profit/Loss statements for businesses. It provides a quick way for managers and accountants to view their return on assets. Some of the boxes in Figure 1 are simply calculations from other values while others are the result of querying values from a database. Each of the values based upon queries from a database can be viewed in more detail by simply clicking on the button above its value. This will open up a more detailed report which allows for customized views on the data of interest. The following account of the development of the DuPont

is taken directly from the book "Relevance Lost: The Rise and Fall of Management Accounting", written by H. Thomas Johnson and Robert S. Kaplan.

> "The DuPont Powder Company was an explosives firm founded in 1903 by three DuPont cousins, Alfred, Coleman, and Pierre. Upon acquisition of the company the three cousins began to develop a new adminstrative structure so that they might better evaluate and control the company operations. Thenew structure involved consolidating the company's operations. The Powder company became a centrally managed entreprise coordinating throught its own departments most of the manufacturing and distribution activities formerly mediated through the market by scores of specialized firms.
>
> A big key to the new corporate structure was the development of a centralized accounting system. The home office requested from the company's mills and branch sales offices, which were located throughout the United states, daily and weekly data on sales, payroll, and manufacturing costs. The company wanted to use this data to constantly monitor the financial performance of all the branch divisions.
>
> With the company structure and accounting system in place, all that was left to do is develop a tool. The companies intense desire to assess every aspect of the company's activities in terms of the price of capital led the founders of the DuPont Powder Company to devise an ingenious return on investment formula. That formula is what has come to be known as the DuPont model.
>
> The breakdown of the DuPont formula helps see how return on investment is affected by a change in any element from either the income statement or the balance sheet. Viewed from this perspective, the DuPont return on investment formula is an ideal tool for controlling, with accounting numbers, any verticall integrated company's opertions.
>
> On interesting side note. It appears that the idea for the DuPont return on investment formula originated with F. Donaldson Brown. Brown was a college-trained electrical engineer and one-time electrical equipment salesman. Brown had no formal accounting training. It is believed that Brown's profiency in mathematics, sales experience, and engineering and marketing skills gave him a uniques perspective on the determinants of company performance."

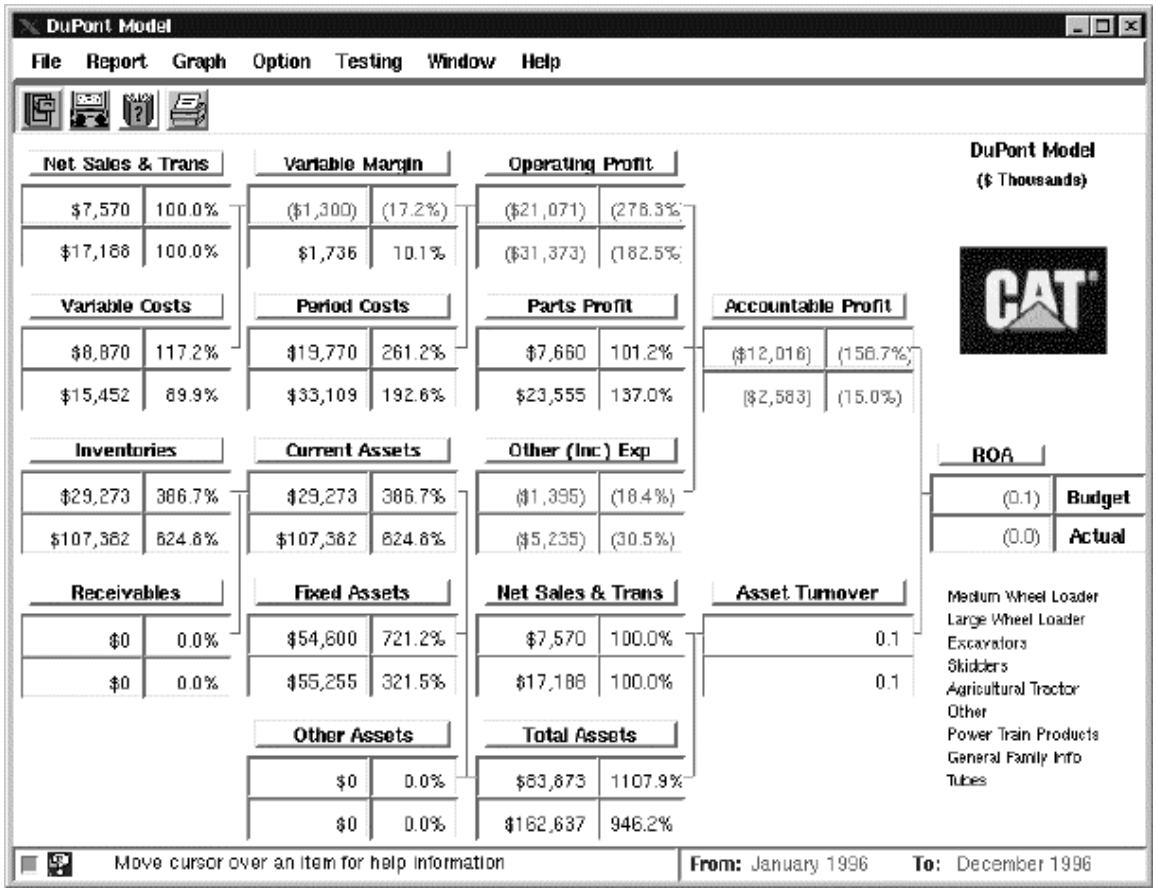

Figure 1 - DuPont Model

2.2 Drilling-Down

Viewing the source of a value is called "drilling down." When the user drills-down on a specified value, they get a report similar to the one seen in Figure 2. This drill-down on Inventories is a first level report that shows where the ***DuPontModel*** Inventories values come from. This report is created dynamically and can be customized by the user. Note that the Budget and Actual columns are derived directly from database queries while "Profit +/-" and "% Change" are extra columns that are calculated as any mathematical function of the previous columns. These calculated columns can be defined by the GUI developer and it is possible to allow the end-user to specify these. Whenever the user changes the selection criteria that the queries are based upon, the new values based upon the changed queries are automatically propagated to all open windows.

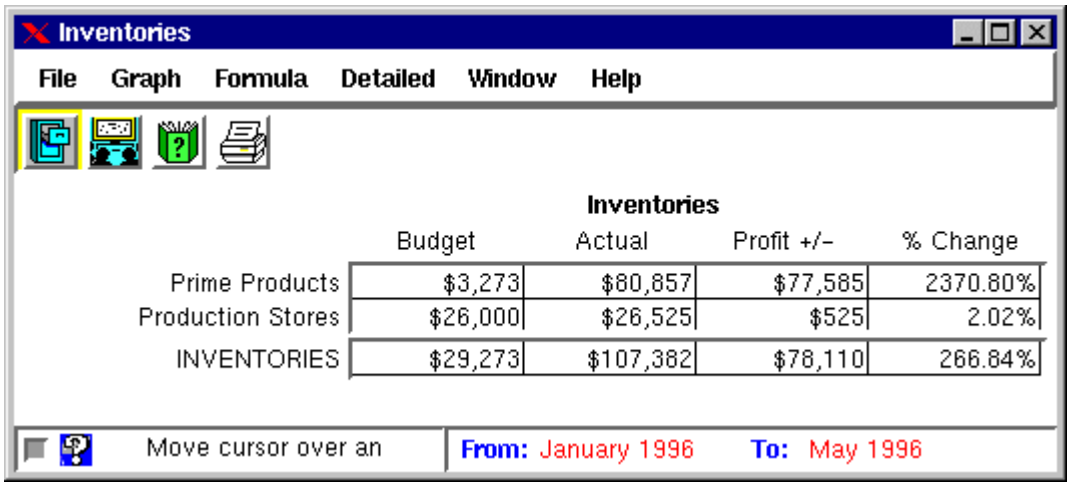

Figure 2 - Drill Down on Inventories

From here the user can open up customizable summary, graph, and detailed reports. This is done by either selecting the desired report from a user-defined menu item or by clicking on one of the rows in the table which has a user-defined report associated with it. Although our current system has the application-builder (domain-expert) define these reports ahead of time, our framework does make it possible to allow for the end-user to dynamically create these reports.

2.3 Detailed Transactions

The Detailed Transactions report (see Figure 3) allows the user to view and edit the individual transactions. The designer can select which columns are editable and which columns are viewable. There is also a GUI provided which allows the end-user to specify SQL-Queries selecting which transactions they want to view and/or edit. These reports are dynamically built and the framework allows for these to be defined by the end-user at run-time.

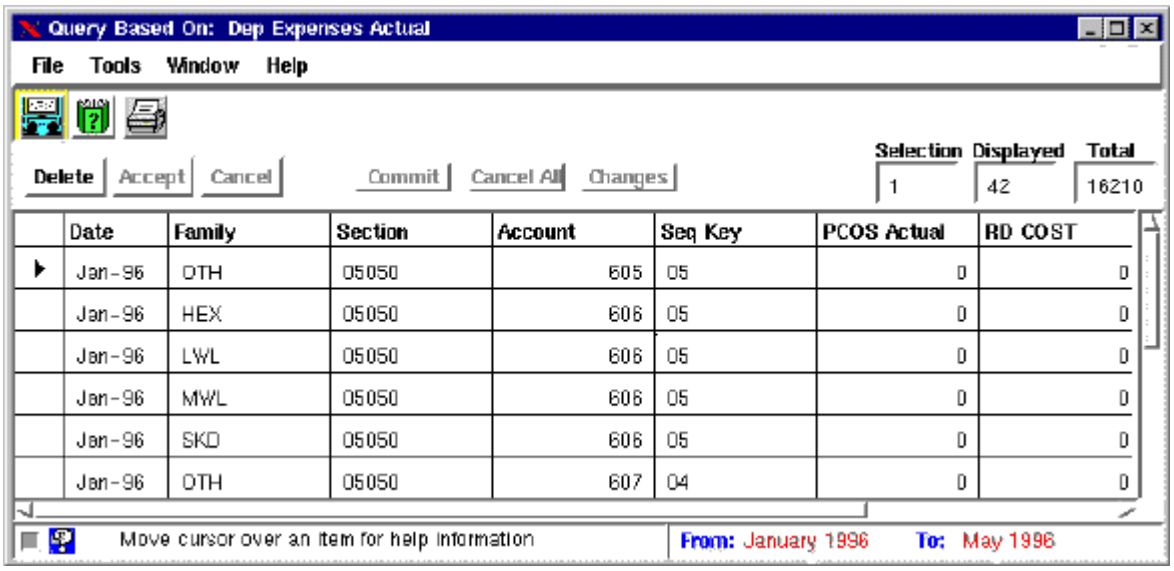

Figure 3 - Prime Products Inventory Detailed Transactions

2.4 Summary Reports

Summary Reports as seen in Figure 4 allow for any type of report to be generated that comprise the that is being drilled-down on. These reports allow for the user to view any type of summary report on the data in a spreadsheet fashion. These summaries can slice-and-dice the data in any manner. This includes allowing the user to join different aspects of the business together to search for different financial aspects of the business.

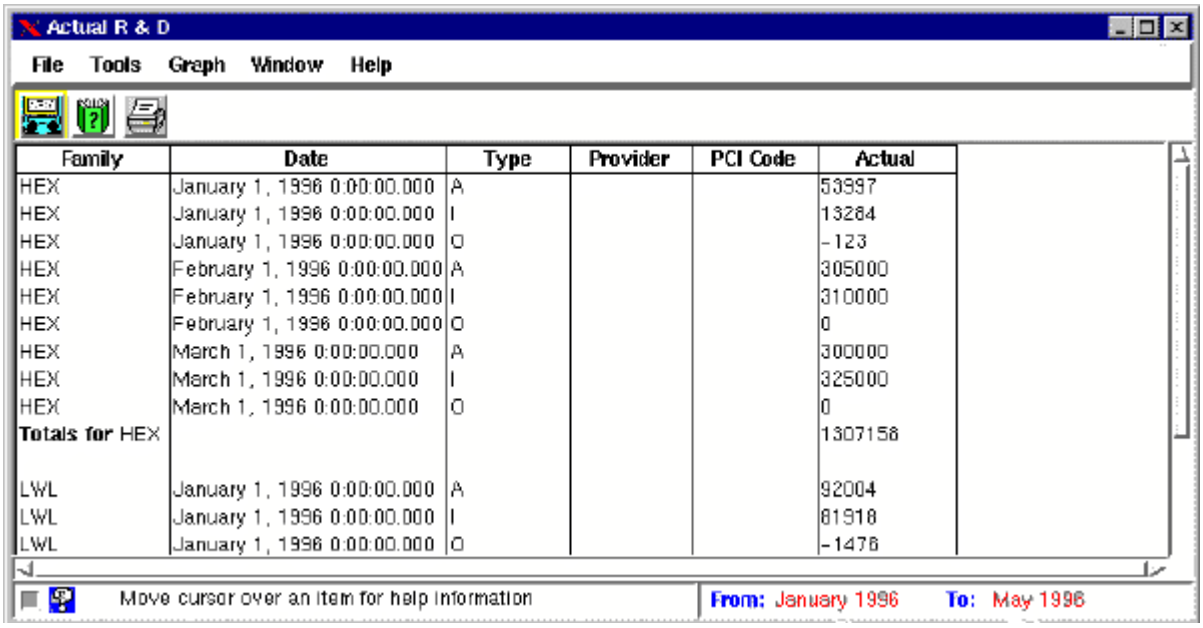

Figure 4 - Prime Products Inventory Summary Report

2.5 Graphs

Graph reports as seen in Figure 5 allow the user to specify the type of graph (bar, line, pareto, band stacked), the orientation, the legends, and other viewing options. These graphs can be built either on SQL-Queries for values from the database, or from pre-calculated values. Once again, although these are currently pre-defined by the application-builder, our framework allows for GUI extensions that could have end-users defining there graphs at run-time.

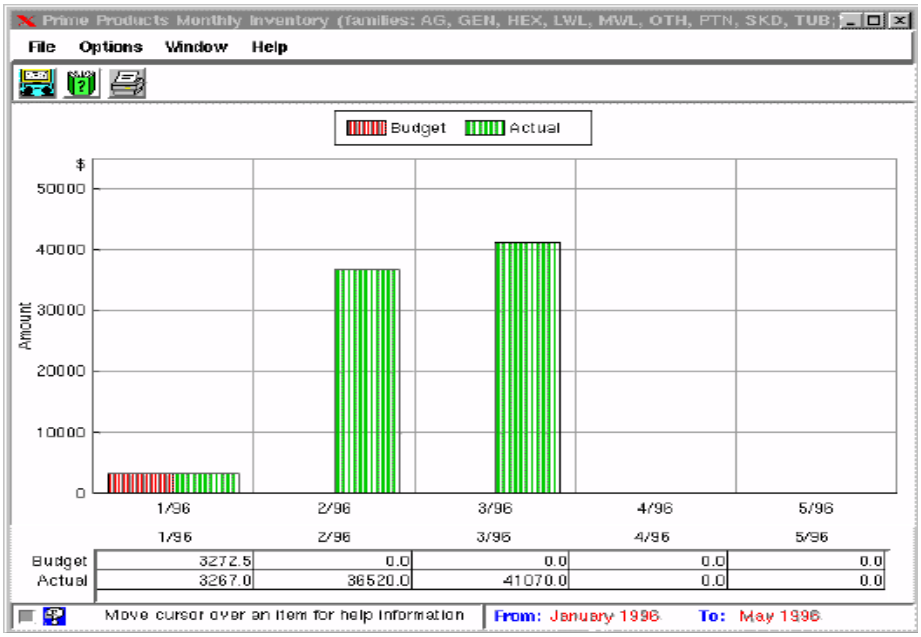

Figure 5 - Prime Products Inventory Graph

3. The Architecture and Design of the Financial Modeling Tool

The Financial Modeling Tool is a set of frameworks for creating financial models. The prototype system was developed using ParcPlace Smalltalk. It provides an architecture for creating financial models and visual tools and languages for generating these modules without knowing the specifics of the implementation language. It contains the following primary frameworks / modules.

I. User Interface Frameworks

- ***DuPontModel*** - A graphical view of the return on sales.
- ***ReportModel*** - Builds a spreadsheet interface using values and GUI descriptions from ***ReportValues***.
- ***GraphReports*** Framework - Used to graph any desired values from the database.
- ***SummaryReports*** Framework - Gives a summary view on values from the database.
- ***DetailedWindows*** Framework - Allows users to edit and view individual transactions from the database.

I. Business Logic Frameworks

- ***QueryObjects*** - Used to create queries into the database.
- ***ReportValues*** - Specifies the business logic and GUI for reports.
- ***SelectionCriteria*** - Allows for the dynamic creation of desired selection values. This works in conjunction with ***QueryObjects*** to allows users to select their values of interest.

I. Utility Frameworks

- Printing Module - Allows any window to print itself with a user selected output
- Security Module - Insures that only desired users can edit and / or view the data. This is configurable by an administrative module.
- Testing Framework - This works in conjunction with the report values and the selection criteria. It automatically selects values from the database and insures you have the desired results.

3.1 What happens in the Financial Model Application

There are four types of users of the financial modeling framework. There is 1) the end-user; 2) the GUI designers / builders; 3) the business-logic programmers (domain-experts); and 4) the system administrator who sets up the security, users, etc. This framework supports all four kinds of users.

The above is possible because we have developed a special purpose high-level visual programming language that allows the experts to build the financial application they need. This specialized language allow the application builders to build according to equations and rules of the business logic rather than coding in the specifics of the implementation language. Thus, the Financial Model can directly be represented according to its high-level organization allowing for the domain experts to build (program) it.

The users view of a Financial Model is a set of reports. These reports interact with the business logic making up the actual Financial Model. Before we look at the design of the business logic, it is helpful to see the relationship between the reports and the business logic.

As seen from a report's perspective (shown in Figure 6), a value displayed on the screen in a report actually comes from some ***ValueModel***. This ***ValueModel*** can be a function based upon a query into the financial database or any mathematical formula based upon other ***ValueModels***. Thus, queries return values that can be plugged into objects. In Figure 6 the displayed value of $5,000 comes from a query which selects all North American Sales from the database and divides it by 1000 since the view displays the amounts in thousands of dollars.

If the specified date range or products of interest changes, the query automatically re-queries the database and the changes are propagated through the ***ValueModels*** to the display. The specifics of ***ValueModels*** and ***QueryObjects*** are discussed later.

There is a set of equations making up the business logic for a company. This business logic is built upon a complicated data model representing all of the financial data. The business unit data model that we designed used the Stars Patterns [Steve Petterson]. This financial data could be relation or legacy data. The financial model application takes this business logic and builds different customizable views to the data.

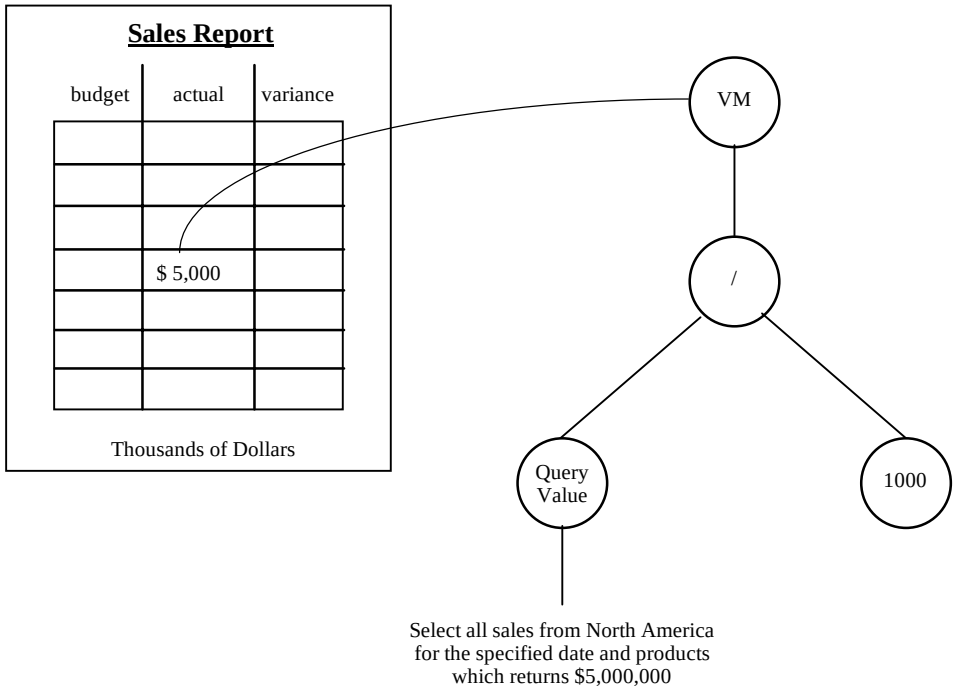

Figure 6 - Value Models in GUIs

Figure 7 gives a high-level overview of what happens. The application builds the values based upon the specified business logic and returned values from the database, and then builds an appropriate GUI and plugs the values into the GUI's.

The application does the constructing of GUI's and Values. The GUIs really take basic building block provided by the system and puts them together to provide the desired view. The Values are constructed automatically from the business logic that is stored in the database.

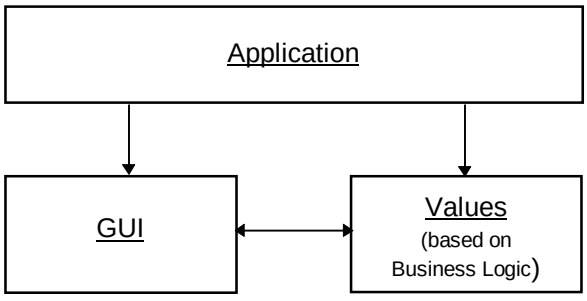

Figure 7 - Financial Model Application Overview

3.2 Financial Model Architecture

Figure 9 shows the user view of the Financial Model as a layered architecture. The three layers are the GUI Layer, a Modeling Layer, and a Composition Layer (see Figure 8). The GUI Layer is the actual application view objects that are built upon the business logic. These are the classes provided for the high-level, summary, detailed, and graph views of the business data.

The Modeling Layer builds the business logic using queries and ***ElementSpecs***. This is where all of the displayed values are created. ***ReportValues*** create the ***ValueModels*** based upon ***ElementSpecs*** descriptions queried from the database and plugs them into the GUIs. Ultimately, these ***ElementSpecs*** are tied back to value models, usually through ***BlockValue*** objects. The queries are instances of ***QueryObject***. The Modeling Layer builds values based upon the specified business logic and plugs these values into GUI's.

The Composition Layer is where the ***ValueModels*** and ***QueryObjects*** are linked together for caching in values from the database. These values are depenedent on the desired ***SelectionCriteria***. It is these ***ValueModels*** which are plugged into the reports for display. A dependency mechanism insures that if the user selects new values to be displayed, the ***ValueModels*** will automatically be updated and any changes propagated to all open views. ***FMState*** is a global object (one per instance of the application running) that knows the user's access rights, the business unit, and the selection criteria.

Each box in a ***DuPontModels*** gives budget/actual values and allows you to drill-down to view more detailed information and reports. Each box gives a different set of reports. Click on NetSales and you get the NetSales report, click on Inventories and you get the Inventories report etc. The application developer defines the break down of where the values come from and the GUI for the associated reports.

Also note that there are two types of boxes in the DuPontModel. Their are those that are calculations based upon other component boxes and there are the leaves that get their values from SQL queries. Models map to every interface widget with a view/controller. There was a commonality with these widgets, especially the model where the logic to the values is stored. Thus their needs to be an object for each box that give values and specify the drill-downs. This object was built by looking at what GUI and Business Logic specification was needed.

Another view of the layered architecture can be seen by looking at it from a builder's perspective (see Figure 9). When a ***ReportValue*** builds a new GUI, it builds the appropriate ***ValueModels*** and plugs them into the GUI. These ***ValueModels*** are dependent on the ***SelectionCriterion***. The GUI's can open up an editor to change the ***SelectionCriterion*** which then updates the ***ValueModels*** which in turn update the values presented on the screen.

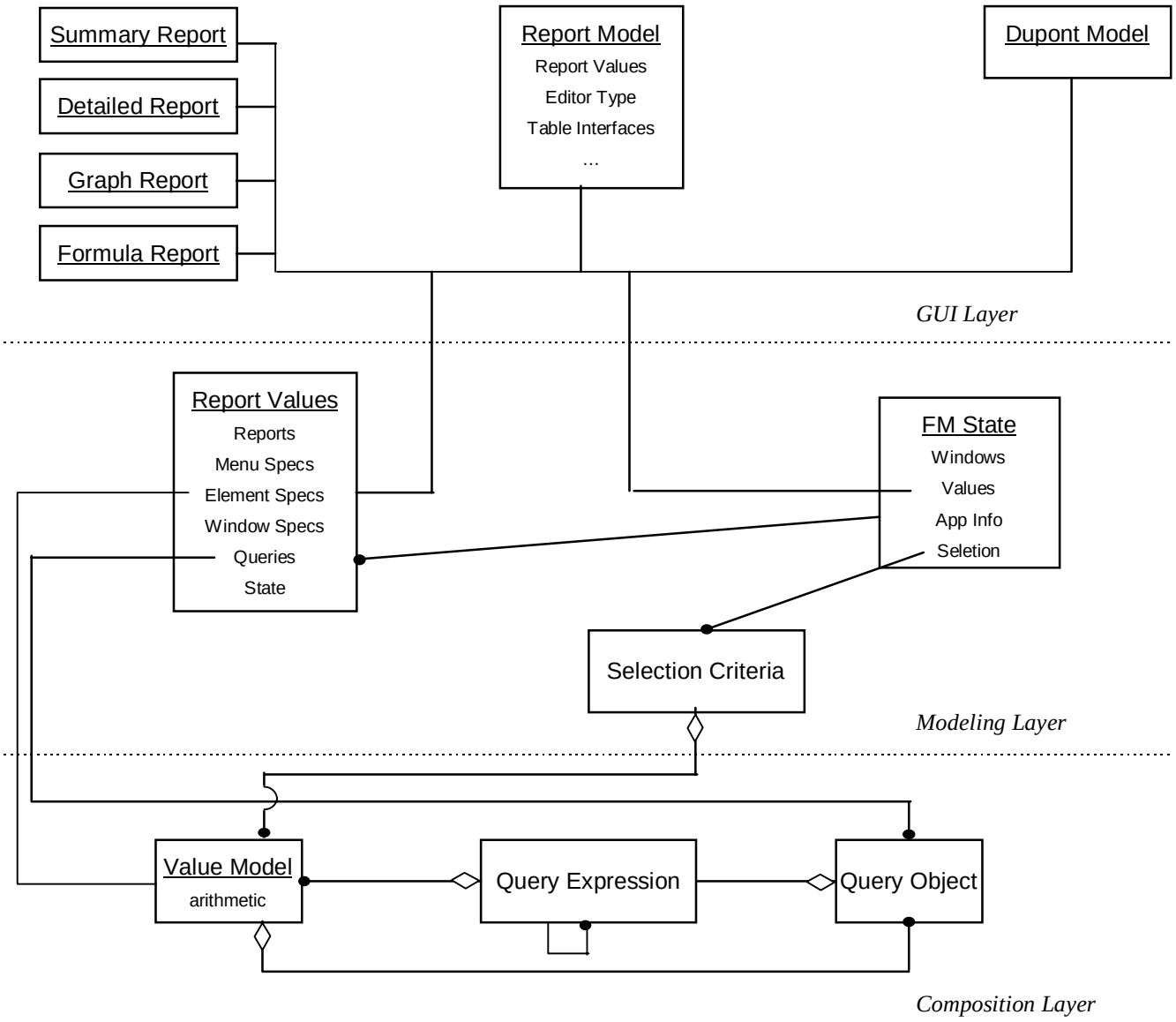

Figure 8 - User View of the Financial Model Layered Architecture

3.3 How it all fits together

A high-level view of the design, collaborations, and flow of the financial model can be seen in Figure 10. A user logs into the system by going through a Security Module that validates user information and loads the application information. This in turns creates the application session information which is called ***FMState***. ***FMState*** keeps track of all of the current application session information which allows for the accessing of values, creation of the reports, and providing security. Once the ***FMState*** has been created, a ***DuPontModel*** is opened with a reference to the newly created ***FMState***. The values presented by the ***DuPontModel*** are accessed through the ***ReportValues*** object via the ***FMState*** object. Here, lazy initialization is done to query values from the database when needed. It is the ***ReportValues*** that keep all of the domain-specific information.

A ***DuPontModel*** can also open up an report showing where each of its displayed values came from. It does this by requesting a specific ***ReportValue*** to open up a report on itself. A ***ReportValue*** knows what its view looks likes that it wants to open and also contains the business logic associated with the view to open. It takes this descriptive information and opens up a ***ReportModel*** which is the combination of the desired view and business logic.

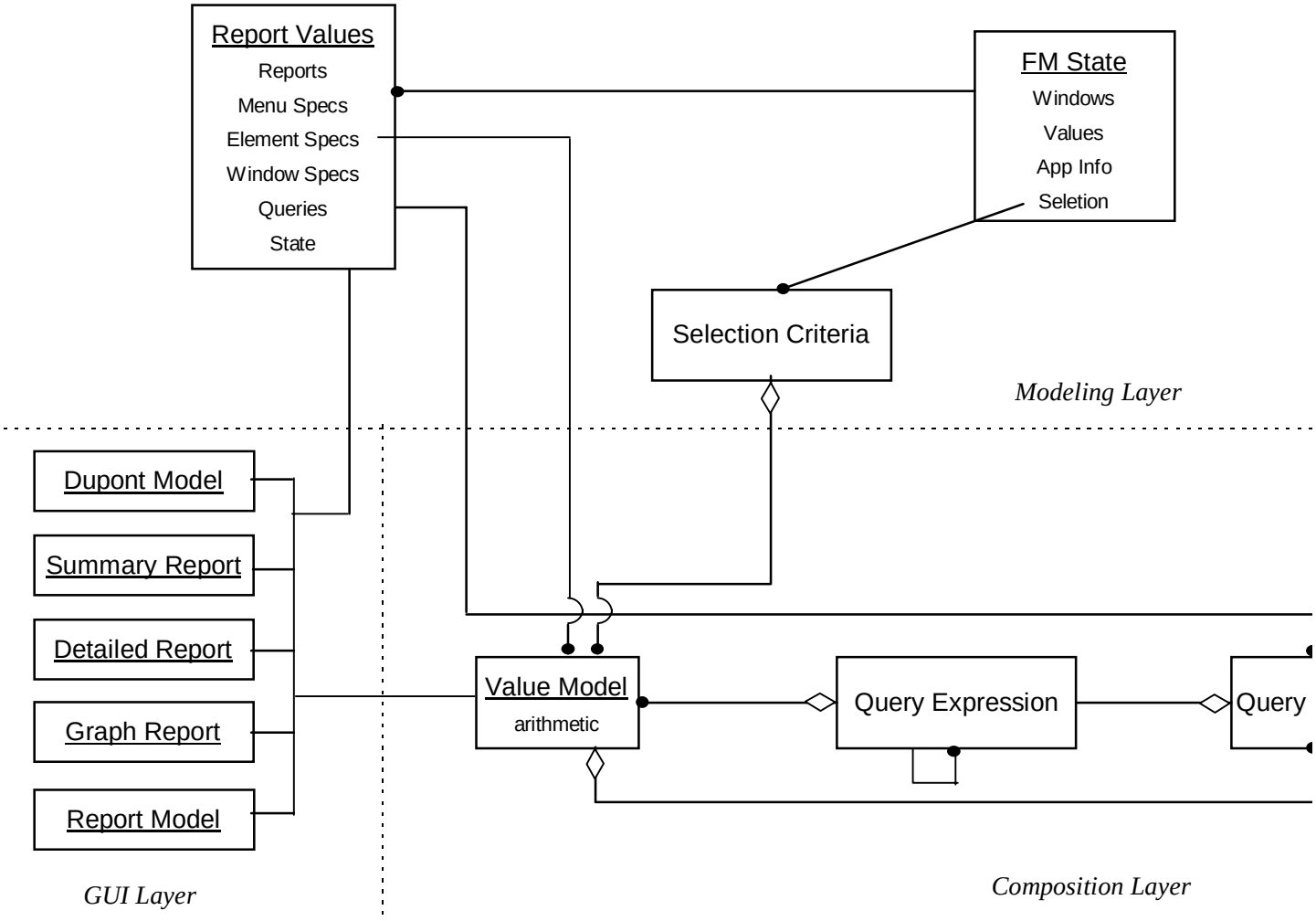

Figure 9 - Builder View of the Financial Model Layered Architecture

A ***ReportModel*** is also given information for opening up other reports for the specific ***ReportValue***. These reports consist of other summaries, graphs, and detailed transactions. This information is built up into menus and clicking actions on the reports which call the appropriate method in the specific ***ReportValue*** via ***FMState***.

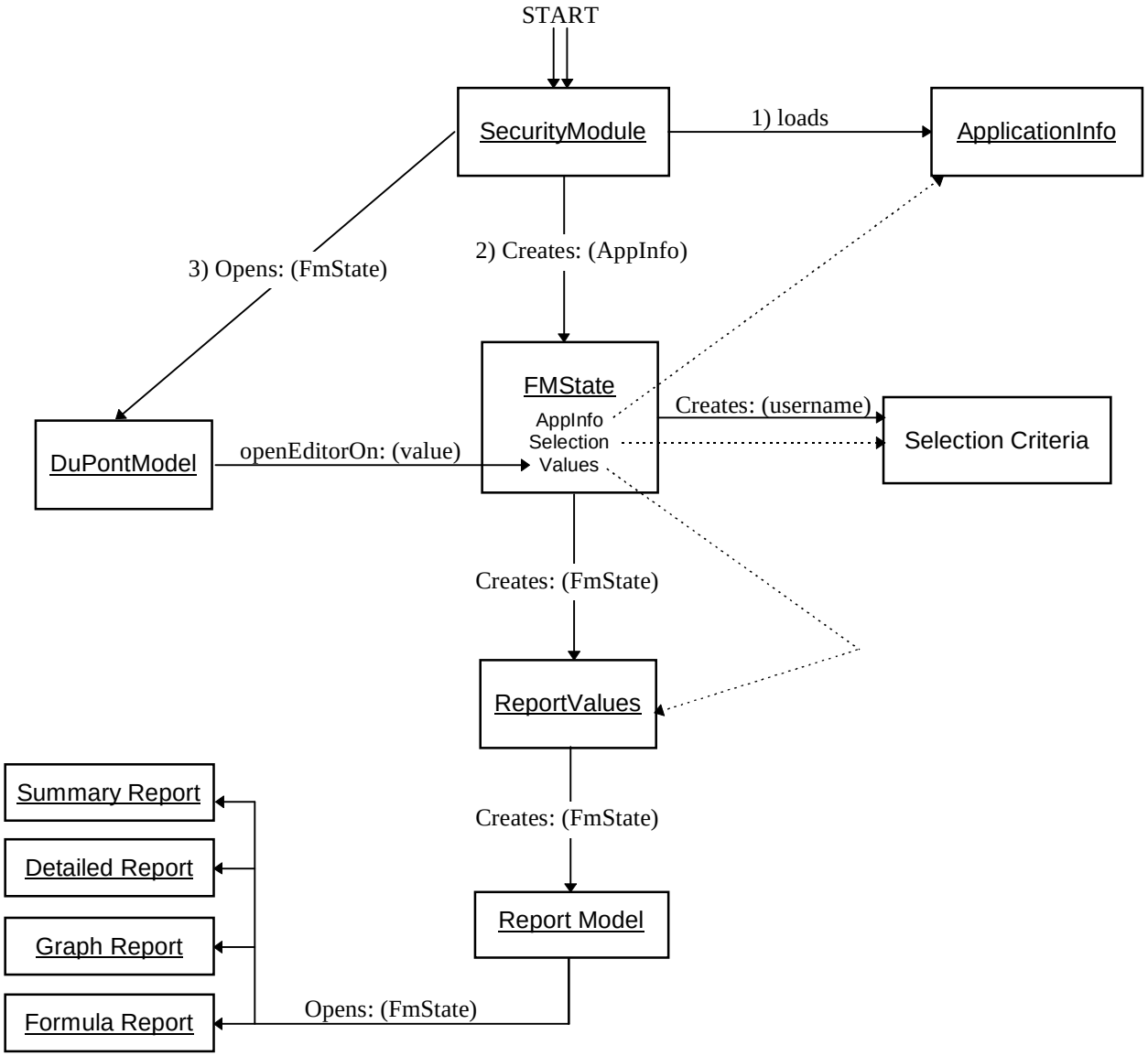

Figure 10 - Financial Model Creational Diagram

3.4 How to Specify a FM

ReportValues define both the business logic and the GUI descriptions for different reports. When the system starts up, a collection of application specific ***ReportValues*** is created by reading the GUI descriptions and Business Logic from the database (see Figure 11). Each of these ***ReportValues*** know how to query their respective values from the "Business Unit Specific Data" and build up any needed values to plug into the generated reports. ***ReportValues*** also works in conjunction with ***ReportModels*** for opening up other Summary, Graph, and Detailed reports. Since ***ReportValues*** are dynamically built, changing the descriptive information in the database for a ***ReportValue*** can change the GUI's and business logic without any programming and also provide for user-customizable views.

Although not shown in Figure 11 for the sake of clarity, sometimes there is limited communication between the "Business Unit Specific Data" and the Summary, Graph, and Detailed reports. This is primarily done when an expensive query is used to return the results or updates need to be done to individual transactions. The

queries are still obtained from the ***ReportValues*** but then a direct connection is made for any needed values and/or updates.

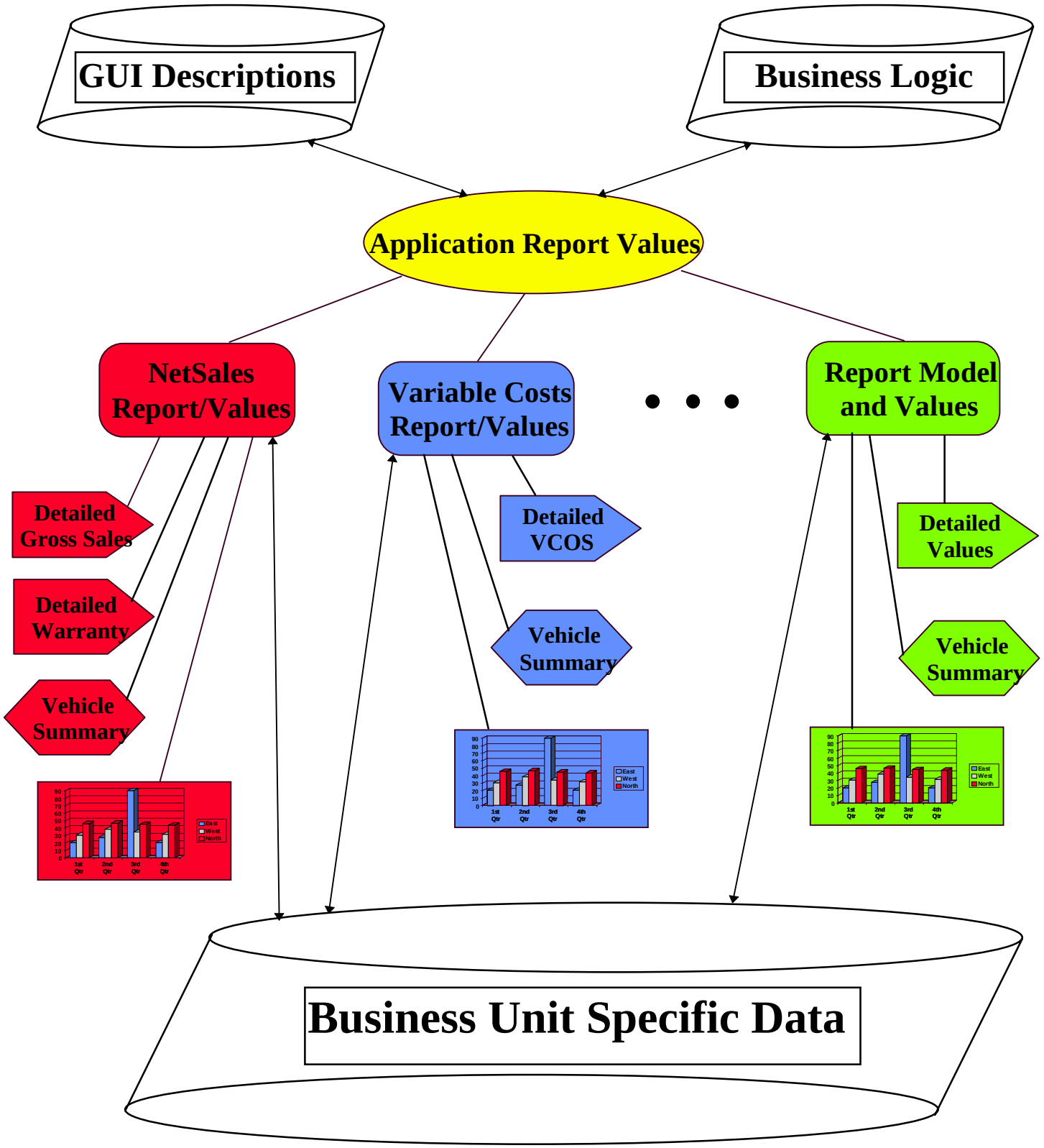

Figure 11 - What Happens

3.5 ReportValues

A ***ReportValue*** holds a specific value to be displayed and also knows the rules for calculating that value (i.e. the business logic and the GUI descriptions). A ***ReportValue*** can also drill down on its value, showing how the values are calculated. It uses the same rules for calculating its value that it does for creating the GUI, so the GUI and the values are always consistent. Usually the GUI shows a spreadsheet of some kind. It lets the user drill down on the value in the spreadsheet in several ways, perhaps by drawing graphs, perhaps by showing detailed lists of transactions, perhaps by opening the GUI for the next lower ***ReportValue***. The primary function of a ***ReportValue*** is to allow values to be built based upon the specified business logic and to allow for users to selectively slice and dice these values to get the desired summary, graph, and detailed views.

The rules for calculating values are often complex. Consider the report for Inventory. It calculates Inventory according to the formula Inventory = Prime Products Inventory + Parts Inventory. Prime Products and Parts Inventory are computed by querying the database, i.e. they correspond to SQL queries. Figure 13 shows the GUI with values plugged in. The menus are automatically generated from the Business Logic and GUI

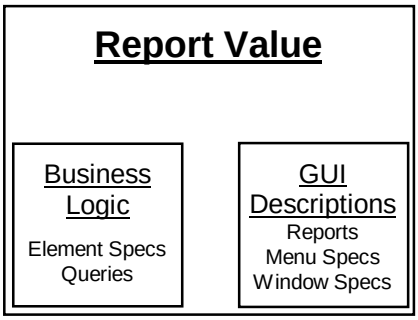

Figure 12 - *ReportValue*

Descriptions as well.

It is up to the application designer to specify what data to display

1. The ***DuPontModel*** will display the total value of the entire ***ReportValues*** group information.
2. The next drill-down level displays a more specific information in report tables.
3. The reports from the drill-down can be one of three more specific views of the data.
   a) The user can view the current or more specific information in a graphic form.
   b) The user can obtain other summary information on the current tables.
   c) The user can obtain the database table's detailed row level information. These will either return all of the rows for the specified selections or they can specify a selected list of the rows they are interested in. If the user is permited by the security module, they can edit these transactions and save them back to the database. They can also sort the rows on any columns they desire.

As can be seen in Figure 13, the drill-down report of Inventories are values obtained from two tables of the database, `prime_products_sums` and `prod_stores_sums`. The GUI interface of this window is designed following the ***ReportValues*** GUI Specifications for the Inventories report.

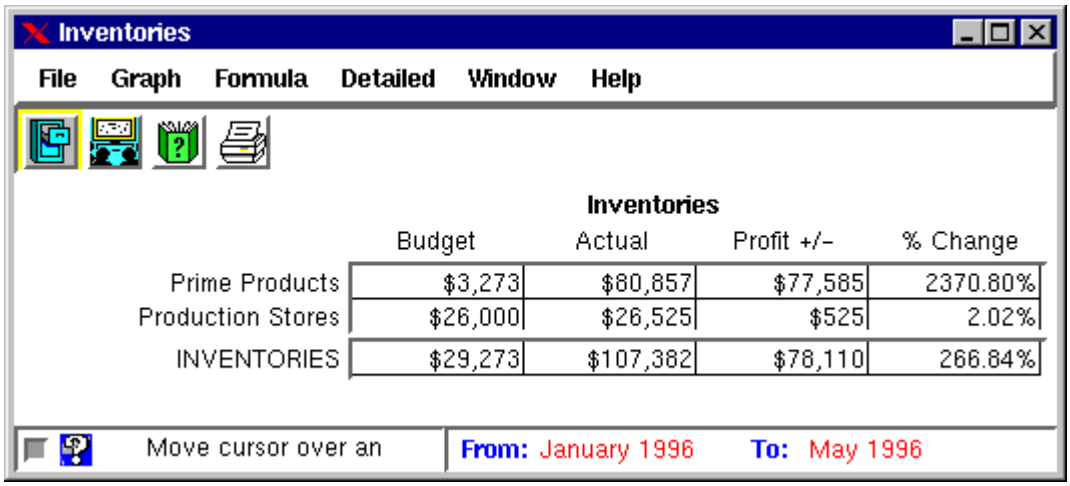

Figure 13 - Drill Down Values

ReportValues GUI Specifications in Figure 14 shows an example of the GUI specifications for the report. The window level specifications are defined in ***ReportInterfaceSpec***. The specifications include the column labels, the label of the window, the table specifications, and the functions for calculating some of the columns. The value of some of the columns are obtained from the queries, but some other columns, 'Profit +/-' and '% Change', are calculated from the values of the other columns. It is indicated in the picture below, the 'Profit +/-' is the 'Actual' column minus the 'Budget' column. '% Change' column, moreover, is the 'Profit +/-' column divided by the 'Actual' column. These calculated columns can also be stored in the business logic descriptions in the database.

Each table's specifications are in ***TableInterfaceSpec***. It holds the name of each table, which can be displayed if desired, and the collection of row specifications. It is shown in **Error! Reference source not found.** that the first table's label is shown, but not that of the second table. In actuality, the second table, which is called the total table, is generated automatically. The application builder only needs to indicate whether or not that table is to be shown and how to calculate it. The default for the total table is to add up the total of all of the other tables, but this arithmetic formula can be changed.

Each row's specifications are in ***TableRowSpec***. It holds the label of the row, which is shown on the left side of the tables, the values for each column, and the source of the values. Each value corresponds to a ***ValueModel*** object which is inserted into each element on the row. Therefore you can notice each ***ValueModel*** holding a number corresponding to each cell in the above picture. The source of these values can be obtained from various sources. They can be obtained from the ***ElementSpec*** objects. They can also be obtained from the total of other ***ReportInterfaceSpecs*** or ***TableInterfaceSpecs*** or ***TableRowSpecs***. In fact, although it is not shown in this picture, arithmetic operations can also be performed on the totals of these specs. For example, you may want another table which holds one row that divides the 'Prime Products' row by the 'Production Stores' row.

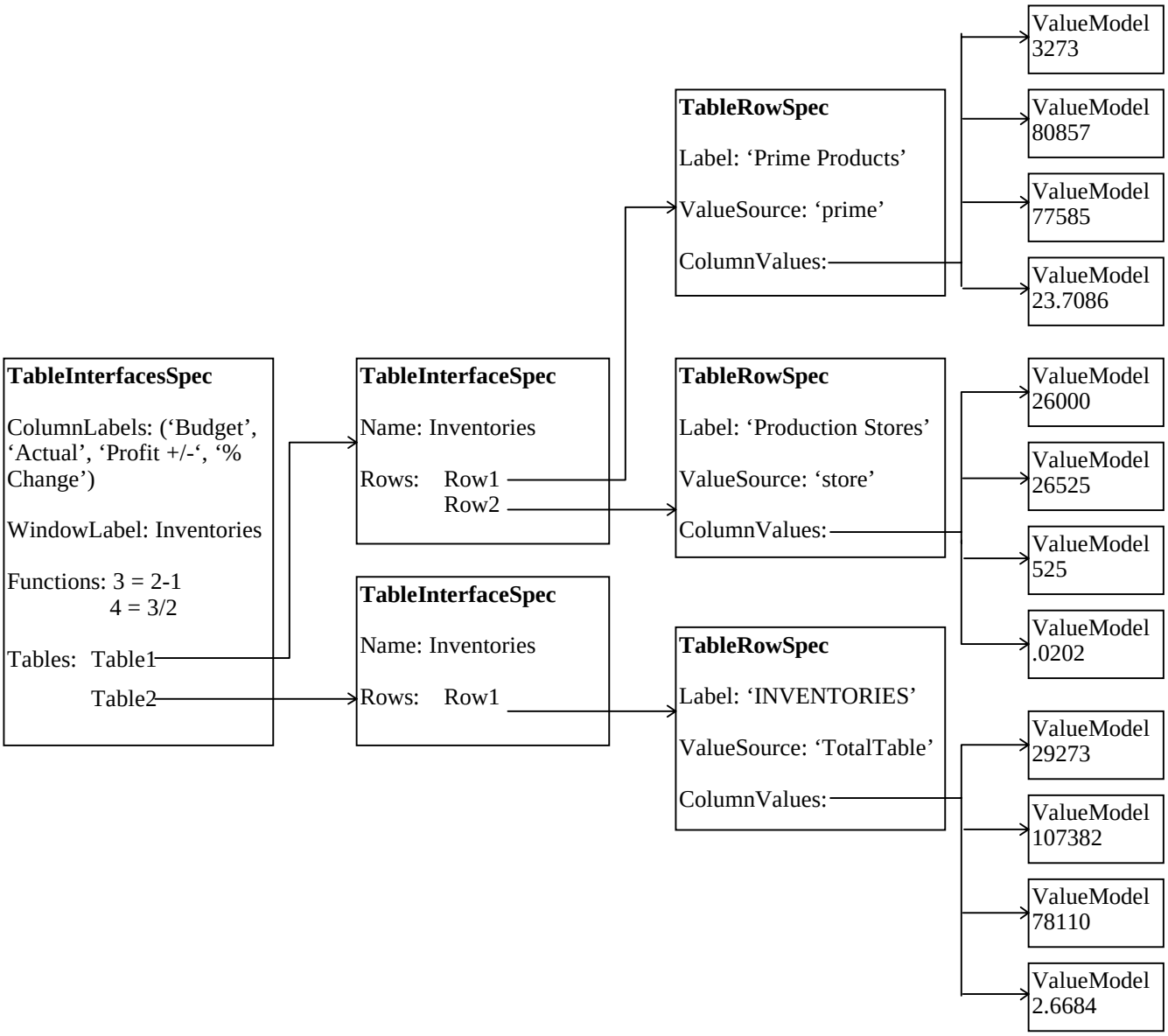

Figure 14 - *ReportValues* GUI Specifications

ReportValues Business Logic Specification in Figure 15 shows the structure of the collection of ***ElementSpecsHolder*** and ***QueriesHolder***. As you can see from below, the ***ElementSpecs*** are ultimately dependent on the ***QueryObjects***. But they can also be an arithmetic operation of two other objects. In the case below, the ***ElementSpecsHolder*** has 'prime' pointing to a ***CompositeElementSpec***, which is an ***ElementSpec*** divided by the number of months selected. The child ***ElementSpec*** in turn points to a ***QueryObject*** with the information about how to sum up the rows returned from the query.

A direct dependence on the queries is undesirable because database queries are so expensive. If there are many queries, then it can take a long time to get all the values from the queries. In fact, there are many redundancies, where several queries may be accessing the same database table to obtain only slightly different values. So it is preferable to query as little as possible. One effective way of minimizing queries is by getting all the needed values in few large queries and then selecting the desired values from those query values. The selection of the query values is done by the ***ElementSpecs***. Although heavy arithmetic calculations are now performed on the client side, the reduction of the number of queries results in a performance improvement. It is up to the user to determine the optimal division between queries and the element specifications. Also, query values are cached

into memory and held there until the desired values are no longer needed or the selection criteria the values are based upon changes. This minimizes the amount of network traffic since if we already have queried for the desired value, we can just use it without re-establishing a connection to the database.

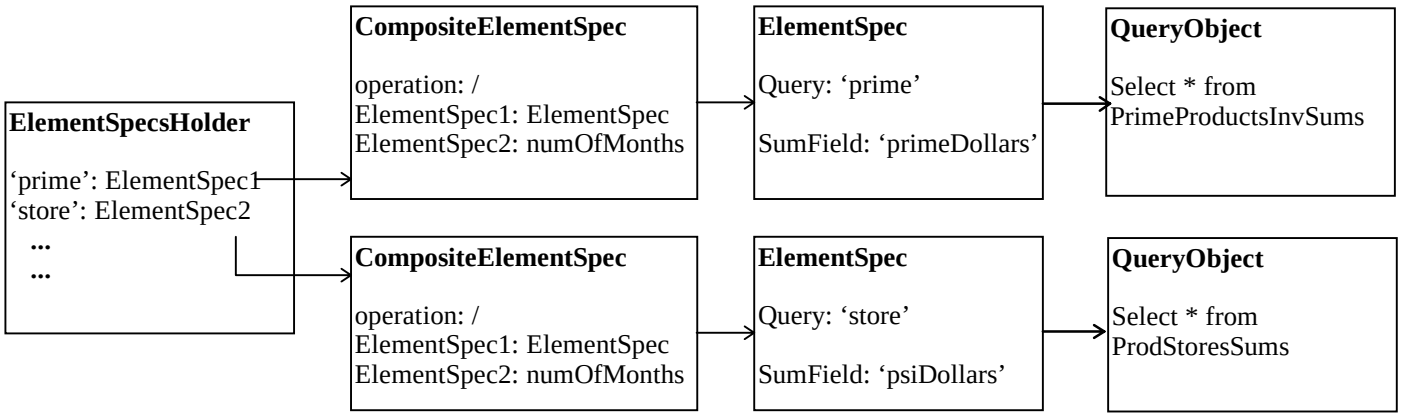

Figure 15 - *ReportValues* Business Logic Specification

Figure 16 shows the creational diagram of ***ReportValues*** and ***ReportModel***. ***ReportValues*** must first create its specs by querying for them from the database. This includes ***ReportGraphSpecs***, ***ReportDetailedTableSpecs***, and ***ReportSummarySpecs***. Other objects are also created from the specifications in the database. They are the menu which is used for the report, ***QueriesHolder***, and ***ElementSpecsHolder. QueriesHolder***, and ***ElementSpecsHolder*** need ***ReportValues*** for proper creation.

Once these are created, the ***ReportModel*** can be opened. The ***ReportModel*** needs information about ***FMState***, and ***Menu*** for the ***ReportValues*** to be properly created. It is the ***ReportModel*** that prompts the opening of the ***DetailedTable***, ***TableGraph***, and ***SummaryReportModel*** windows. The creation of these objects is actually done on the ***ReportValues*** side since the menu holds blocks from ***ReportValues*** that are evaluated when selected.

One exception is ***TableGraph***. ***ReportModel*** automatically creates a graph for each table in the drill-down report. In this case, the ***TableAdaptor*** of the table interface must be passed for display to ***TableGraph***. The other way of creating a graph is through the ***ReportGraphSpecs***, which makes a graph based on an ***ElementSpec***.

SummaryReportModel needs a ***QueryObject*** which will provide the view on the summary data. Similarly, ***DetailedTable*** needs ***QueryObject*** and ***QueryDescription*** with which the most detailed row-by-row data of tables can be obtained. ***QueryDescription*** determines the type of ***DetailedTable*** that is to be opened.

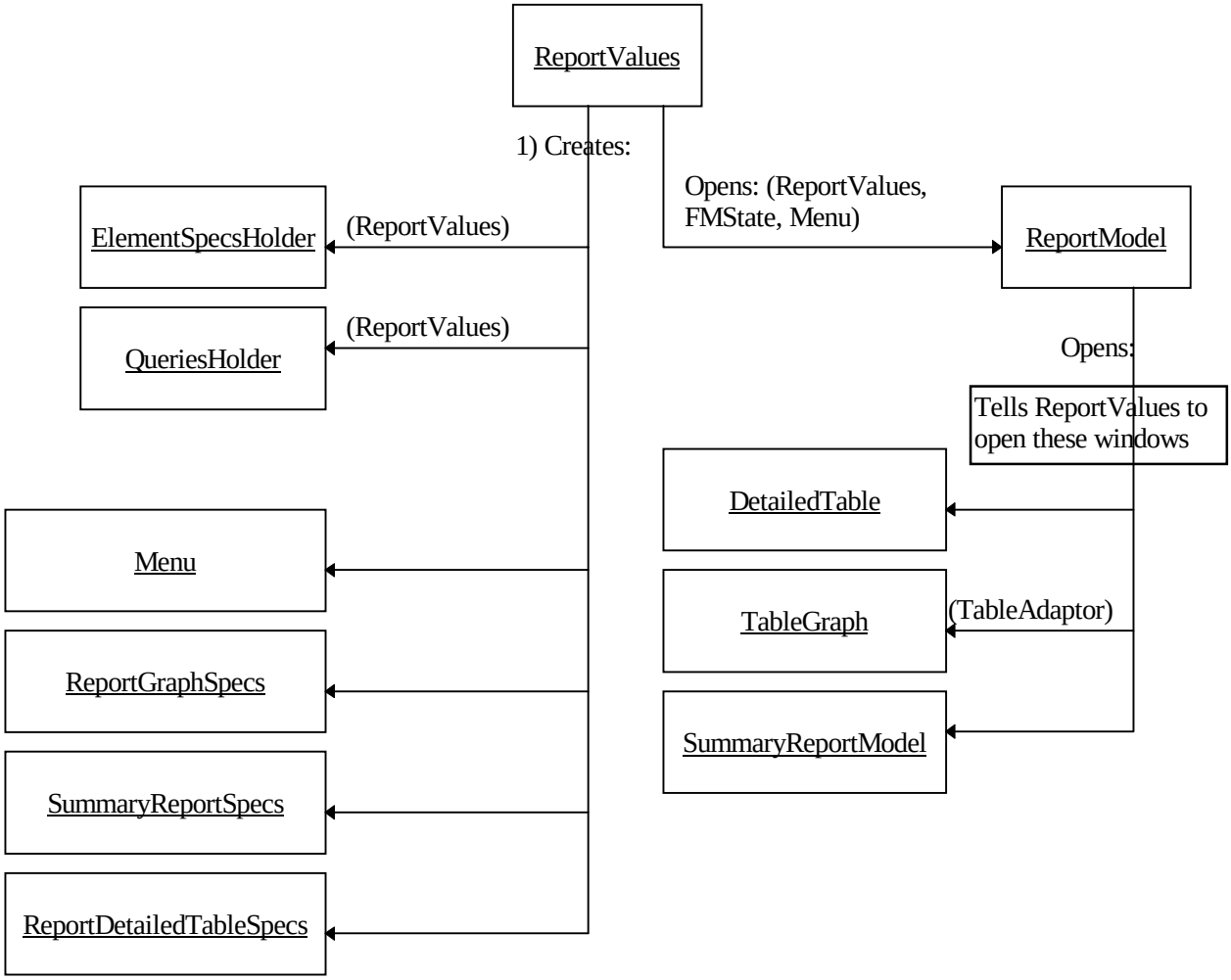

Figure 16 - ReportValues Creational Diagram

Figure 17 shows a structural diagram of ***ReportValues***. It shows a detailed structure of the objects that ***ReportValues*** needs. It first needs the ***FMState*** in order to pass around the information. The queries are built up using ***QueryObjects*** for each query needed. The ***ElementSpecs*** can point to a ***SingleElementSpec*** or a ***CompositeElementSpec***. ***SingleElementSpec*** makes its results by directly putting constraints on the values returned from a ***QueryObject***. ***CompositeElementSpec***, on the other hand, is an arithmetic function on two other ***ElementSpecs***.

The specification of the reports are stored in the following manner. Each report, in ***ReportInterfaceSpec***, consists of a label, window-level formats, such as the specification of the functional columns, and the specifications of the tables. It also has a total value. The total value is a ***RowInterfaceSpec***. It indicates the total value of the entire report. This total value is needed when an arithmetic operation of two reports is performed. This is basically the arithmetic operation of the the two total values. The ***TableInterfaceSpec*** is similar to the ***ReportInterfaceSpec*** in having its own total value. The functional column specifications are passed down to this object. But it points to a collection of ***RowInterfaceSpecs***. Each ***RowInterfaceSpec*** actually holds the ***BlockValues*** that are inserted into the cells of the tables. It makes the ***BlockValues*** based on the result of the ***ElementSpecs***. It inserts the values for the functional columns based on the specifications. It can also be tied to an action that is executed when the cell or row is double-clicked.

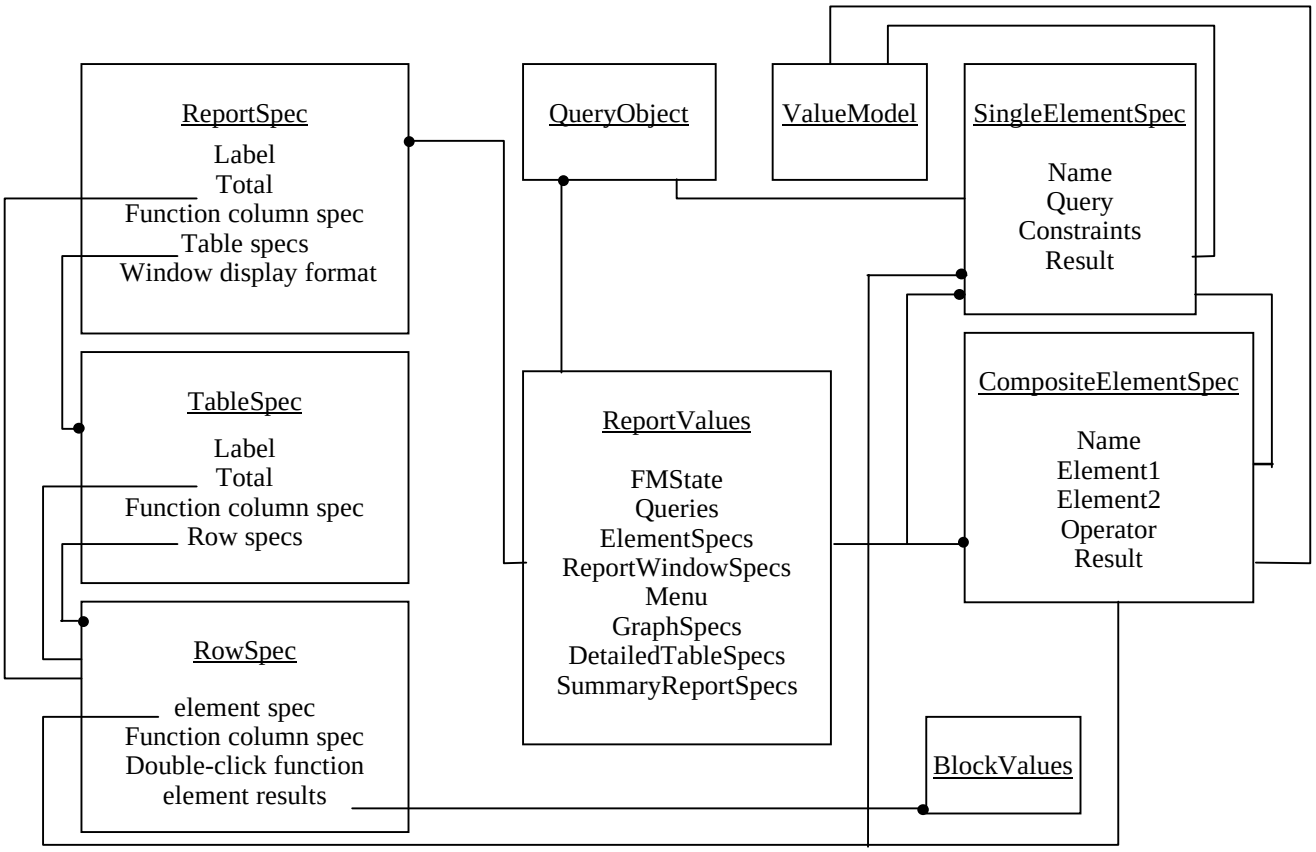

Figure 17 - Structural Diagram of ReportValues

4. Details of the Design

This sections describes the more intricate details of the design. Occasionally, implementation details will be described. This is only done to helps clear up the design or a very complicated implementation that the details are needed for understanding how to implement this architecture.

4.1 Creating DuPont Models

Each box in a ***DuPontModels***, has formulas and database queries associated it. A typical way to build GUI's like this is to use an interface builder to draw the text fields and buttons and then write code to the associated behaviors. This code would include such things as defining all of the queries/formulas along with all of the associated constraints. If this only needed to be done once, then it probably would be an efficient way to develop the software. However, different business units will have different business logic and different views on their data which included different boxes, constraints, formulas, and database queries. Moreover, the database structure layout will be somewhat different for each business unit. This prompted us to extend the Visual Builder framework to automate the construction of ***DuPontModels***.

As was seen in the ***DuPontModel*** example (Figure 1), there is a common interface widget that is used many times. By creating a "DuPont widget" to be used with the GUI builder along with methods for the automatic generation of related code, the developer can quickly tailor different ***DuPontModels*** to meet the needs of different users.

We were able to extend the VisualWorks Smalltalk Visual Builder framework to re-use code and easily extend the ***DuPontModel*** by creating another interface widget like the DuPont box which has all of the fields and the associated button. The developer can then draw the DuPont boxes quickly on the screen and use the property editor along with the builder's automatic define method to automatically generate the formulas, the default accessing methods, and the needed constraints. This extension to the Visual Builder can be done in other languages but may be harder. For example in Visual Basic you can define new widgets, but then you have to go off and write "C" code for the behavior and compile and test. You need to work with two different languages to extend the builder.

The ***DuPontModel*** is really a collection of ***DuPontSpecs*** and ***DuPontPctSpecs***. These are special sub-classes of ***SubCanvasSpec*** which are pre-built to define the default look of a DuPont widget. Once given a name it will get its ***ReportValues*** object by concatenating its with "Values" and "ValuesPct" and asking ***FMState*** for that object. Therefore, whatever application model these widgets are placed on need to understand the state message which returns a ***FMState*** object. "ValuesPct" is for the percentages. Once it gets the ***Values*** object budget and actual are sent for the top and bottom boxes respectively. Whenever the button is pressed, the openEditor message is sent to the values object which defaults to "No further drill-down available". The only difference between ***DuPontSpec*** and ***DuPontPctSpec*** is that ***DuPontPctSpec*** only deals with percentages while the ***DuPontSpec*** has percentages and whole dollar values.

In order to meet the above, the ***DuPontModel*** requires an object that knows its values and can create GUI's to examine more specific details of these values. These objects are known as ***ReportValues.***

4.2 Business Logic and GUI Specification Data Model

Business Logic and GUI Specification Data Model

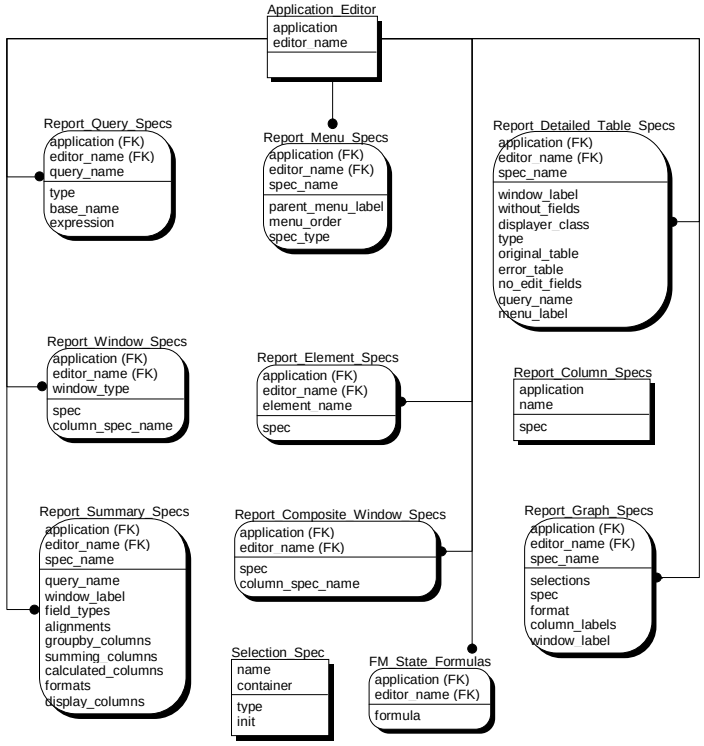

Figure 18 - Business Logic and GUI Data Model

The data model that contains the descriptive data (meta-data) needed to build a financial model is called the Business Logic and GUI Data Model. The foreign key relationship are shown in Figure 18 - Business Logic and GUI Data Model. Figure 19 depicts the entity relationships. To build a GUI with a given report element requires information detailing the look and feel of the GUI as well as information regarding how the data is to be displayed.

`Report_Query_Specs` holds the queries used to retrieve data from the financial database for all the report elements.

`Report_Detailed_Specs` stores the data necessary for building a table based report.

`Report_Summary_Specs` stores the data necessary for building a summary report. Both it and `Report_Detailed_Specs` have a `query_name` field which relates them to `Report_Query_Specs`. This provides the corresponding report elements with means to retrieve the data they are to display.

`Report_Graph_Specs` stores data for building graph reports, but it does not have a reference to `Report_Query_Specs`. Rather it is associated with `Report_Element_Specs` which contains the information needed to get the query from `Report_Query_Specs`.

Report_Composite_Window_Specs contains information for putting one or more report windows together.

Report_Window_Specs' data specifies how to build a report window. A report window contains one or more element specs found in Report_Element_Specs. Report windows also have menus which are defined by Report_Menu_Specs.

Report_Menu_Specs contains data for building a menu. Each menu item is associated with a detailed table report, graph report, summary report, or another menu.

Selection_Spec has the data for building a selection criteria box. It references the data in Report_Query_Specs necessary for specifying its queries. Selection_Spec also has data referenced by Report_Query_Specs to augment the query specification in Report_Query_Specs.

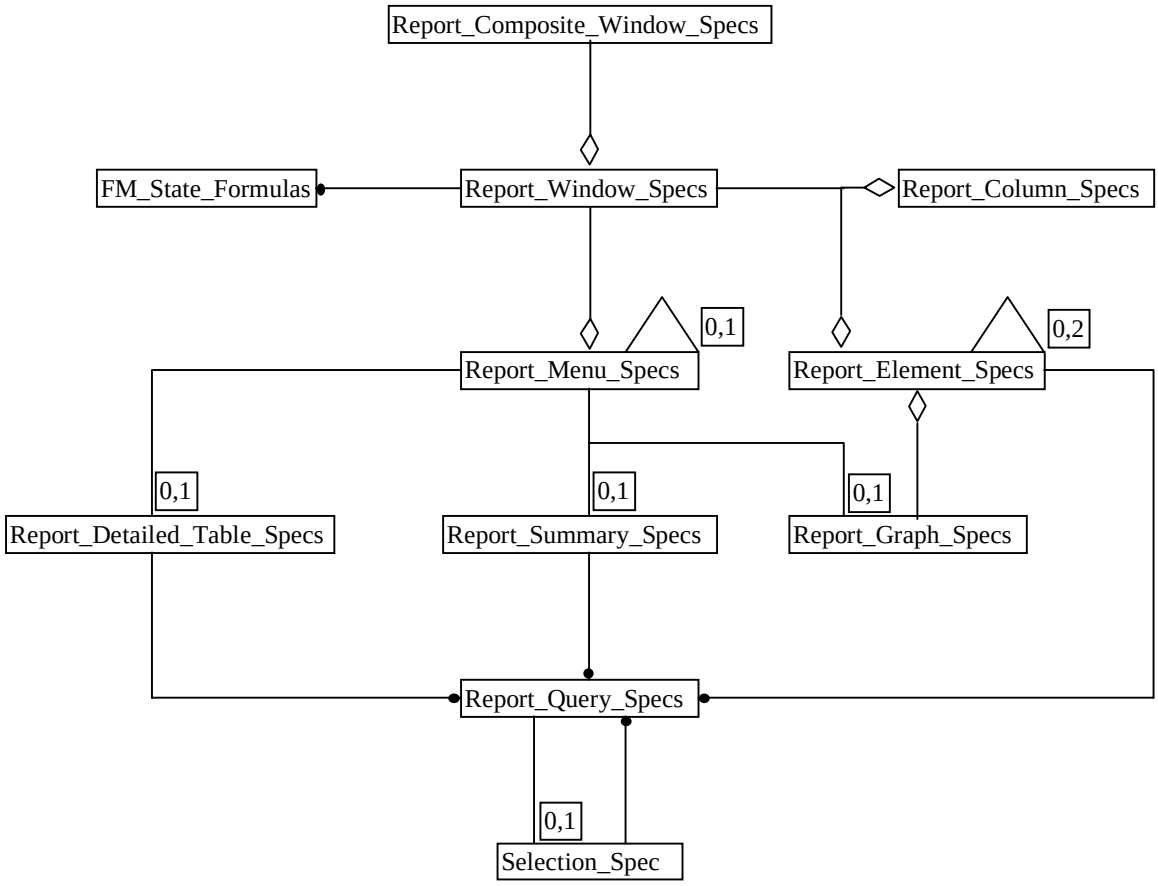

Figure 19 Business Logic and GUI Entity Relationship Diagram

4.3 ReportValue Details

ReportValues is an object that holds the values from database queries that are needed for producing reports. It has the following capabilities.

I. **Keeping track of the queries needed for the report, as well as the values returned from the queries**. In reality, this is split into two parts, ***QueriesHolder*** and ***ElementSpecs***. The ***QueriesHolder*** holds the queries as well as their cached values. The ***ElementSpecs*** holds specifications for manipulating the query values of the ***QueriesHolder*** to obtain the desired result.

1. ***QueriesHolder*** holds a collection of associations mapping query names to queries and their results.
2. ***ElementSpecs*** holds a collection of associations mapping names to selection specs. For building the report window, two types of selection specs are available. It always holds query selections specs called 'total'. They should represent the total value of the report. The total is currently being used by the ***DuPontModel***.
   a) ***SingleElementSpec*** returns the sum of all the rows that satisfy the given constraints. It returns a block value which returns the desired sum when evaluated. It holds the following specifications.
      i) Spec name.
      ii) The name of the query value, which it obtains from the ***QueriesHolder***.
      iii) The selection constraints that determine which rows of the result to choose.
      iv) The specification for calculating the value of each row. The specification basically involves arithmetic operations on the columns of the row that hold numbers.
   b) ***CompositeElementSpec*** returns the result of an arithmetic operation on two other selection specs. This spec is indirectly dependent on the query values. It holds the two ***ElementSpecs*** and the operator. The second operand can also be a number or a symbol that stands for a method call to the ReportValues object that holds the element spec.

II. **Specifying the report interface**. ***ReportSpecsHolder*** keeps a collection of associations mapping report names to the report interface specifications, called report specs. The interface also has a menu associated with it. More than one interface can be specified, but there should always be a default interface. The report interface's look is basically a collection of tables lined up either horizontally or vertically, each with a specified set of columns and rows. All tables of a given report have the same number of columns. The tables specify the rows, which in turn hold the values that are to be put into the table cells. Instead of inserting numbers, value models (more specifically ***BlockValues***) are inserted so that when the results of the queries change, the cells obtain the new values automatically.

1. The report specs hold the following information.
   a) Report label. This label is used if the report is displayed as a window.
   b) Function column spec. There are columns that are dependent on the query values, and there are also function columns, which are functions of other columns. The arithmetic function for each of the function columns must be specified. The query specification for the query-based columns is not necessary at this level because the specifications of these columns are assumed to be constant throughout all the report specs. Each function column needs to specify two functions, one for a cost table and another for a profit table. The report spec picks the desired one for each table (each table spec should do this).
   c) A collection of table specs.
   d) The total value of the totals of the table specs. The default is to sum up the table totals, but it can be changed. Also a total table can be displayed below all other tables if desired.
   e) Display format. The values can be displayed after being divided by an integer value.
2. The table specs hold the following information. They correspond to the tables of a report.
   a) Table label. This label can be displayed if desired.

b) Function column spec. This spec is obtained from the parent report spec.
c) A collection of row specs.
d) The total value of the row specs.

3. The row specs, which correspond to the rows of a table, hold the following information.
   a) Function column spec, obtained from the parent table spec.
   b) An array that is specifies the action when the row is double-clicked.
   c) A formula for obtaining the values. The formula is an arithmetic specification of ***ElementSpecs*** and operators. Instead of obtaining values directly from the ***ElementSpecs***, it is also possible to obtain values from other row, table, or window specs.
   d) A collection of value models (more specifically block values), corresponding to each column. They are obtained by using the formula and are inserted into each cell of a row in the report interface.

III. **Specifying the graphs**.

I. **Specifying the detailed table information**.

I. **Specifying the summary reports**.

I. **Specifying the menu of the window interface**. The actions for opening graphs, detailed information, and summary reports will be inserted there.

Using *ReportValues*

In order to use the report specs, you must know the queries needed and the desired view of the report interface.

1. It is recommended that you first determine the database tables and queries needed. Then determine the optimal split between the queries and the query value selections.
1. Then determine the look of the report interface. Determine what value each cell of the tables will hold.
1. Then determine the other specifications.
1. From these specifications of (3), create the menu specs. Also insert double-clicking actions into the cells of the tables if desired.

We have developed GUIs to allow the application developer to easily specify all of the above.

4.4 ValueModel

We usually think of values as being the attributes of objects, or sometimes we think of them as being special classes. For example, an Employee object will have attributes like salary and hiring date, and these values will be instances of classes like Money or Date, which we might consider a value class.

But there are other ways that objects can represent values. VisualWorks has a class called ***ValueModel*** that represents a single value. Not only can clients of a ***ValueModel*** read and (usually) write its value, they can become its dependents and be notified when it changes. GUI widgets usually depend on a single ***ValueModel***. So, if an Employee object stores its salary in a ***ValueModel*** then a text widget can depend on the salary and be notified when it changes.

The most common ***ValueModel*** is a ***ValueHolder***, which is just a container of a value. Instead of storing its attribute in an instance variable, an object can store its attribute in a ***ValueHolder***, which can be stored in an instance variable. A read or a write to the instance variable must then get converted to a message to the ***ValueHolder***. This lets clients depend on the attribute and be notified when it changes.

Most other ***ValueModels*** are adapters. For example, a date adapter converts a value that is a date to a value that is a string. Thus, a date adapter might translate between a text widget, which expects a string, and a ***ValueHolder*** containing a date.

The most interesting ***ValueModels*** are ***ComputedValues***, which define one value in terms of others. ***ComputedValues*** are often defined as a function of other ***ValueModels*** and as such use the dependents feature of ***ValueModels*** to stay up-to-date with the other ***ValueModels***. There are many items that can be based on other values. For example, value PROFIT is really a function based on the amount of total SALES minus the total COSTS; whenever SALES or COSTS change so does PROFIT. The PROFIT value is represented by a ***BlockValue*** which is a specialization of ***ComputedValue*** that calculates its function based on a Smalltalk block. Whenever the value is needed the block is evaluated.

In addition to ***BlockValues*** who compute their functions through Smalltalk blocks, there are also ***QueryValues*** which compute their values from queries. This allows the model to be directly hooked into the database without the need of writing specific code to transfer values from queries to ***ValueModels***.

The inheritance diagram for the ***ValueModels*** is shown in **Error! Reference source not found.**. Although the actual VisualWorks ***ValueModel*** hierarchy has several additional classes, only the important ones for the framework are shown.

The default protocol for ***ValueModel*** is very small. It only contains the value message to return its value. But since many ***ValueModels*** are also used in formulas by ***ComputedValues***, it should be easy to combine them to form the formulas. We added methods for arithmetic functions like + and - to ***ValueModel*** that automatically create a ***ComputedValue***. The result is that instead of creating a ***BlockValue*** for PROFIT with code such as:

```
BlockValue
    block: [:sales :costs | sales - costs]
    arguments: (Array with: salesHolder with: costsHolder)
```

we can define the profit as "salesHolder - costsHolder". The definition of - in ***ValueModel*** is

```
aValue
    ^BlockValue block: [:x :y | x - y]
        arguments: (Array with: self with: aValue)
```

We added to ***ValueModel*** all of the basic arithmetic operations and a few for operations for string and date manipulation. As a result, it is easy to define new ***ValueModels*** based on other ***ValueModels***.

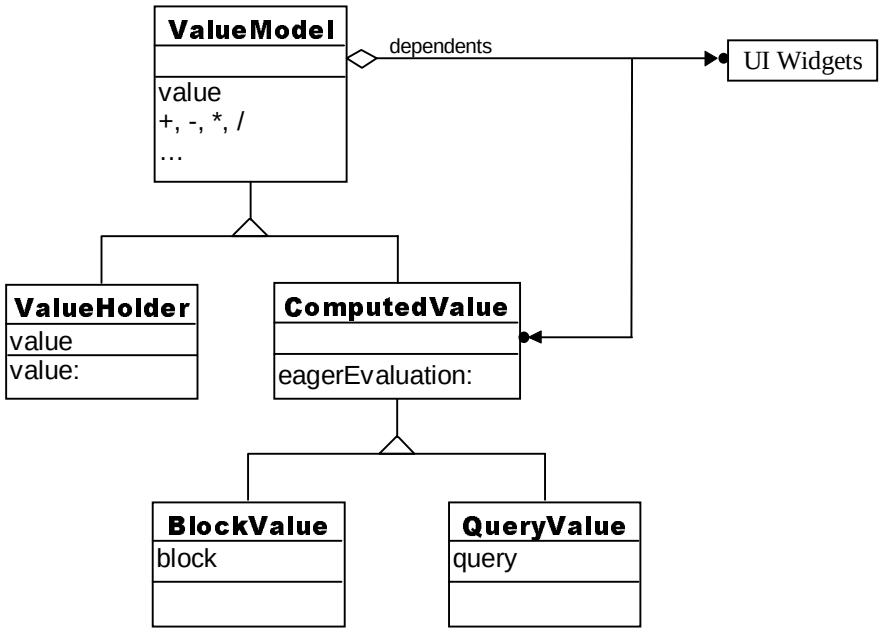

Figure 20 - Object structure diagram for ***ValueModel***

ValueHolders have a direct reference to their value, and as a result have a value: method to set this value. This message is mainly used by interface widgets, but can be used programmatically to change the value. ***ComputedValues*** also add a few messages to the basic ***ValueModel*** protocol that determine how it computes its value. The value can be computed eagerly or on demand by sending the eagerEvaluation: message. Values which are associated with queries turn-off eager evaluation since queries can take seconds to compute. Both ***BlockValue*** and ***QueryValue*** only extend ***ComputedValues*** interface by adding messages to initialize their blocks and queries.

4.5 Queries

Not all information for an application will be stored in memory. Instead, this information is stored externally in a database, and values are fetched by querying the database. For example, consider a payroll system. There might be one function that lists the number of hours worked by an employee during a time interval. For such a system, there would be an interface that allows the user to enter the employee id and date range. Once they are entered, the database would be queried for the number of hours worked.

Relational databases have their own language for specifying queries. Many times the language is SQL. For our application to query a SQL database, it must send its commands as SQL statements. A default SQL statement might look like:

```
SELECT fields to return
    FROM tables in query
    WHERE clause to select
    GROUP BY fields to group on
    ORDER BY fields to sort on
```

The WHERE, GROUP BY, and ORDER BY parts of the statement are optional. For "our hours" worked example above, we would have an SQL statement such as:

```
SELECT SUM(hours)
   FROM time_cards
   WHERE employee_id = '12345' AND
            date < '1/1/98' AND
            date >= '1/1/97'
```

While we could model each query as a string, many queries have similar parts and these parts might change over time. For example, we might have several queries that have the same WHERE clause that specifies that the records returned should be within a date range. If we needed to change the condition to add another condition, we would need to change all the strings in the code. Clearly, this is undesirable.

Another way to model the queries might be to make an object that holds each query part as a string and then concatenates them together when we execute the query, but we would like to include other Smalltalk objects besides strings in our expressions. Instead of constructing a string from code like: "'orderNumber = ', orderNumberHolder value printString", we would rather construct the SQL expression from code like: "salesTable orderNumber = orderNumberHolder". When the query is evaluated the expression is turned into the appropriate SQL statement string. Otherwise, we must update the string whenever the orderNumberHolder ***ValueModel*** changes.

Instead of using strings to model the queries, we chose ***QueryObjects*** to model queries and ***QueryExpressions*** to model the individual expressions (e.g., WHERE clauses, GROUP BY clauses, etc.). ***QueryObjects*** then construct their SQL statements at runtime using the ***QueryExpressions***.

A query returns something that behave like a table; it has rows that are separated into fields. Many databases even allow you to create database views from a query. These views can then be used in other queries as if they were real tables. We want to have a similar feature in our ***QueryObjects***, but instead of having to create database views for each query, we want to be able to use ***QueryObjects*** in other ***QueryObjects***.

The most primitive type of ***QueryObject*** is a ***TableQuery***. ***TableQueries*** represent tables in the database, and are the basic building blocks for all other queries. They correspond to the tables listed in the FROM clause of a SQL statement. Evaluating a ***TableQuery*** by itself just returns all the records in the table. Whenever multiple tables are listed in the FROM clause of a SQL statement, they are "joined" together. This operation is represented by a ***JoinQuery***. ***JoinQueries*** join two ***QueryObjects*** to form one ***QueryObject***.

The other clauses of a SQL query are also ***QueryObjects***. For the SELECT part, we use a ***ProjectionQuery*** since the SELECT part tells the database what fields to project in the result. The WHERE clause is modeled by a ***SelectionQuery*** since it tells the database which rows to select. The ORDER BY and GROUP BY clauses are represented by ***OrderQuery*** and ***GroupQuery*** respectively. Since all of these queries also have expressions, they have ***QueryExpressions*** that will create their SQL code for their respective clauses.

In addition to the SQL syntax described above, there is an additional keyword that can appear in a SELECT statement. The "DISTINCT" keyword specifies that all records returned by the query should be unique. This is modeled in our system by a ***DistinctQuery*** which wraps another query and returns only the unique rows.

There are a couple additional ***QueryObjects*** that don't have a direct SQL mapping. A ***RenamingQuery*** renames the fields of another query. This is useful for achieving consistency with the field names. An ***ImmediateQuery*** evaluates and caches the results of its wrapped query. Instead of letting the database compute the values for the overall query, ***ImmediateQueries*** signify that part of the calculation should be performed in Smalltalk. These can be used for performance optimizations and also when values from one database must be merged with values from another database. Since ***ImmediateQueries*** signify that part of the query calculation should be performed in Smalltalk, we can present a unified query model that can straddle several databases without the developer needing to write special code.

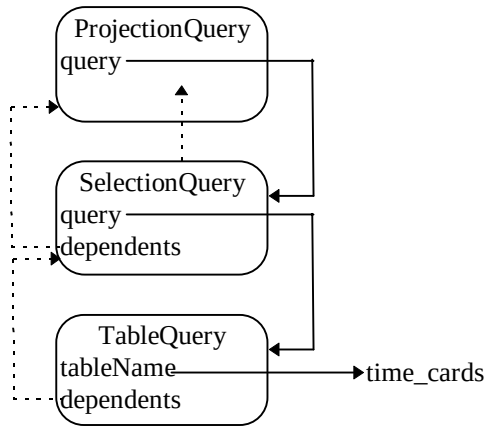

Figure 21 - Dynamic structure of *QueryObjects*

A single SQL statement is represented by a set of ***QueryObjects***. The SQL statement above is shown using ***QueryObjects*** in **Error! Reference source not found.**. Both the ***SelectionQuery*** and the ***ProjectionQuery*** also have ***QueryExpressions*** which are not shown.

Figure 22 shows the ***QueryObjects*** object diagram. The query operations have been split-out under a ***WrapperQuery*** which defines some common behavior for all operational queries. Also, the queries that need a ***QueryExpression*** have been further split-out under ***ExpressionWrapperQuery***.

QueryObjects support a protocol to retrieve values from the database through the value, valueIfAbsent:, values, and valuesAsObject messages. Both the value and valueIfAbsent: messages expect to return zero or one row from the database whereas the values and valuesAsObjects can return zero or more rows. If value or valueIfAbsent: query returns more than one row, then an error is raised. The valuesAsObjects message is used when you wish to return the values from the query as Smalltalk data model objects, and not as arrays.

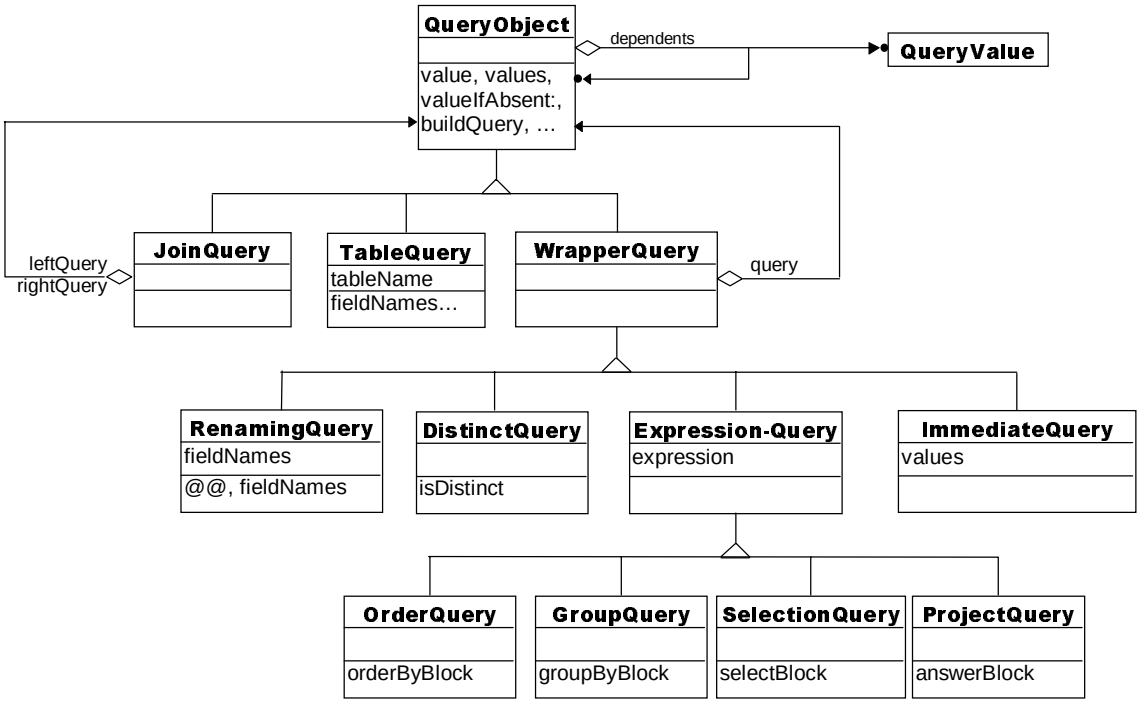

Figure 22 - Object diagram for *QueryObject*

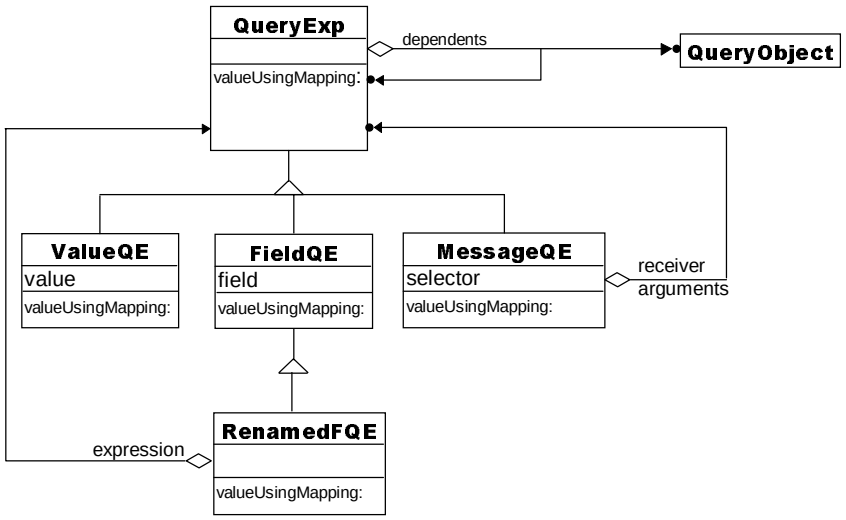

Figure 23 - *QueryExpression's* **object diagram**

In addition to the value retrieval protocol, there are also methods that return fields from the query so that they can be used to create ***QueryExpressions***. There are two main methods that are used for this support: @@ and fieldNames. The @@ message returns a ***QueryExpression*** that represents the field for argument name, and the fieldNames message returns the list of field names that are available to the query.

There is also several "helper" methods that are defined by ***QueryObjects***. These methods allow you to create new ***QueryObjects*** based on the receiver. For example, you can join two query objects together by using the join: message.

In addition to the public protocol for retrieving values, creating ***QueryExpression***, and creating new ***QueryObjects***, there is also a private protocol for converting the ***QueryObjects*** into SQL by interpreting them. The main method that builds the query is the buildQuery *Template Method* in ***QueryObject***. It uses the answerBlock, selectBlock, orderByBlock, and groupByBlock methods to help build the query. Each of these build specific parts of the query.

4.6 QueryExpressions

As mentioned in the previous section, ***QueryExpressions*** specify the expressions for the different parts of the SQL query. A query such as

```
SELECT employee_id, SUM(hours)
   FROM time_cards
   WHERE (date < '1/1/98') AND (date >= '1/1/97')
   GROUP BY employee_id
   ORDER BY employee_id
```

has four expressions (the FROM line is formed from ***JoinQueries*** not expressions). Looking at each expression closely we see that they are almost in a Smalltalk syntax. Renaming a few of the logical operators to their Smalltalk equivalent (e.g., AND → &), we can convert everything except for the functions. Functions with one argument are easily converted to unary messages, and functions with more arguments are converted to keyword messages. Once we convert the functions into messages, we see that each expression is a series of message sends to a field in a table. Therefore, we can represent each expression as a parse tree of message and field nodes. Since queries can also refer to constant values, we also need parse nodes for constant values. These nodes can either hold a constant such as 100, or hold a ***ValueModel*** which holds the constant. If the value node holds a ***ValueModel***, then when the query is evaluated, the current value of the ***ValueModel*** is used.

Error! Reference source not found. shows the object diagram for ***QueryExpressions***. In addition to the three types of parse nodes, there is also a ***RenamedFieldQueryExpression*** class that is used together with the ***RenamingQuery***. Since each field of the answer of a ***RenamingQuery*** can refer to many fields of its wrapped query, we need a reference to the expression that created that field. A ***RenamedFieldExpressionQuery*** holds onto the original expression for the renamed field. For example, given the query above, we might want to rename the fields of the answer to be "`employee_id`" and "`hours_worked`". For such a query we would need two ***ReanmedFQE's*** one for the "`employee_id`" expression and one for "`SUM(hours)`" expression.

As an example of the runtime configuration of the ***QueryExpressions***, **Error! Reference source not found.** shows the expression for the WHERE clause (i.e., the expression that is used by the ***SelectionQuery***).

There are three different protocols for ***QueryExpressions***. One is used for easily forming the expressions. It consists mainly of a redefinition of the doesNotUnderstand: message. This makes it easy to construct the parse trees simply by executing Smalltalk code. Whenever the doesNotUnderstand: message is received, the ***QueryExpression*** constructs a ***MessageQueryExpression*** with itself as the receiver. Although the doesNotUnderstand: mechanism can handle most messages, there are a few that must be overridden since they are defined by ***Object*** (e.g., isNil).

Another protocol is responsible for converting the expression into SQL code. Although we could generate our own SQL code, we rely on the VisualWorks Lens framework to generate it for us. Since we use the Lens framework, this protocol consists of only one message: valueUsingMapping:. This returns the Lens object that is equivalent to the expression, since the blocks used in the Lens queries are similar to our ***QueryExpression***, the valueUsingMapping: method simply evaluates the ***QueryExpression*** in the context needed by the Lens blocks.

The final protocol supports the *Observer* pattern. Since ***QueryExpressions*** can also have ***ValueModels***, they need to update their dependents when they change. These dependents can either be other ***QueryExpressions*** or ***QueryObjects***. Whenever an expression changes, the query that it is contained in must be re-computed.

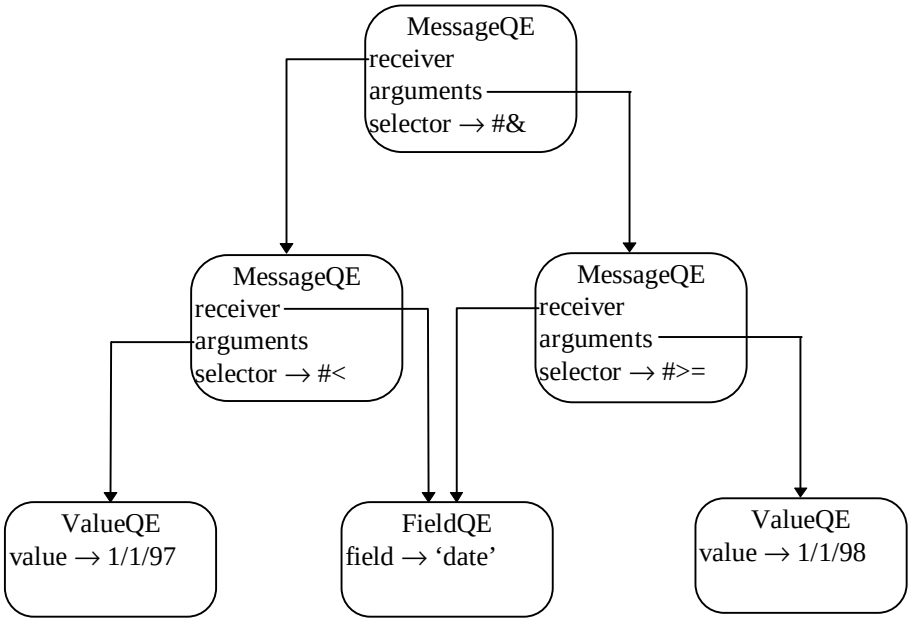

Figure 24 - Dynamic structure of *QueryExpressions*

4.7 Selection Criterion

A financial application queries a relational database to extract data for financial reports. Usually users of the application want to set constraints and view the data satisfying the restriction. Therefore, the queries need to depend on user constraints.

SelectionCriterion is an object connecting queries and a user interface for specifying constraints.

4.7.1 Example

We consider an example from the Aurora financial model application (see Figure 25). In the application users select a list of product families, for which they want to view the data, the period of time they are interested in, and, possibly, the kind of records(internal sales, effect of internal sales).

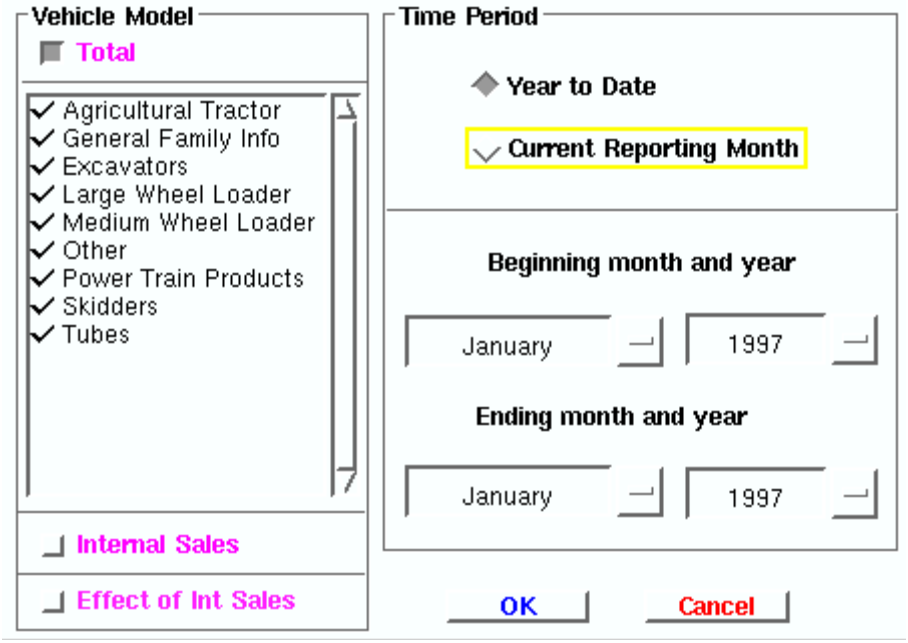

Figure 25 - Aurora Selection Box

The selections a user can make map to value models within the model of the interface, ***SelectionCriterion***. In this case we have the following value models:

- familyList
- isInternalSales
- isEffectOfInternalSales
- startingDate
- endingDate

4.7.2 Implementation

A usual way of implementing ***SelectionCriterion*** is to have one instance variable for each of the value models. However, this approach requires a new class to be created for each new selection interface.

Instead, we would like to have one ***SelectionCriterion*** class to work for every selection interface. Therefore, we use a dictionary of the value holders as an instance variable in ***SelectionCriterion***. In the dictionary each

value holder is assigned a name(key in the dictionary). The name is then used for accessing the corresponding value holder(see examples in *Specifying Selection Criterion*).

Since the value holders are value models, they can be used in query expressions. The query expressions will be then dependent on the value holders. The queries that use the query expressions will also depend on the value holders, and, hence, be recalculated whenever the value holders change.

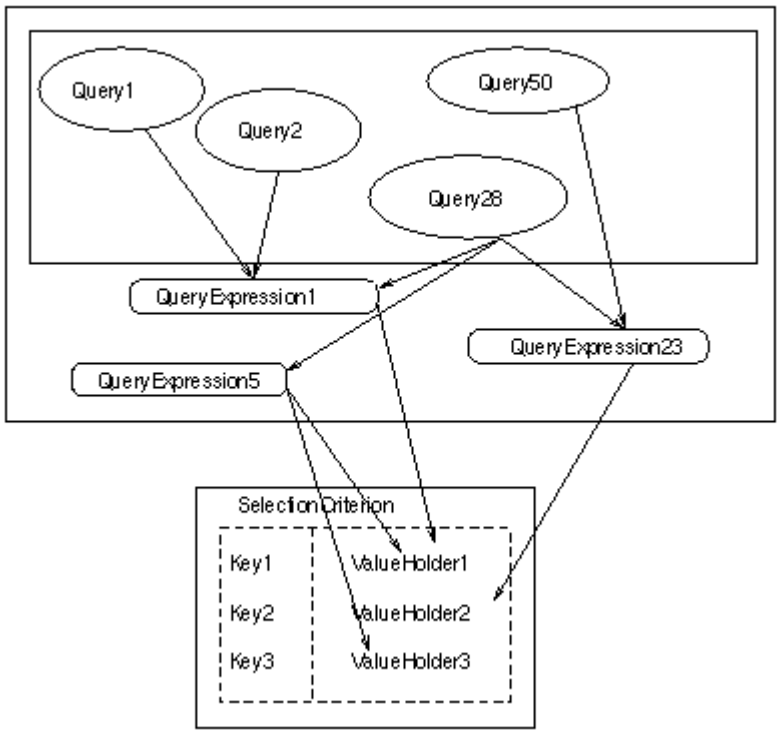

Figure 26 - Specifying Selection Criteria

4.7.3 Query Specification

A problem appears, however, when one tries not only to specify the queries, but also save them in a database. Since value holders are transient objects, it makes no sense to store them in a database.

We solve the problem by have the query specifications use not the value holders themselves, but their names in ***SelectionCriterion***. Then the queries retrieved from the database during runtime look up corresponding value holders in selection criterion by their names, use them to make query expressions, and become dependent on them.

4.7.4 Selection Interface

SelectionCriterion also contains the name of the selection interface class. For example, ***SelectionBox*** represents Aurora financial model selection interface, which is depicted above.

One could also automatically build the interface from the specifications of a particular ***SelectionCriterion***. There are several problems with this. First, it is hard to layout nicely widgets for value holders without putting constraints on the value holders. Second, there may be widgets not corresponding to any of the value holders. Third, some of the value holders need not be displayed. Therefore, a programmer will still have to modify the selection interface.

4.8 Specifying Selection Criterion

The figure below presents the structure and the process of constructing ***SelectionCriterion*** for a financial model application.

As shown, an instance of ***SelectionCriterion*** has a dictionary of value holders(selections). A key in the dictionary is the name of the corresponding value holder. The key is also the name of the method called to

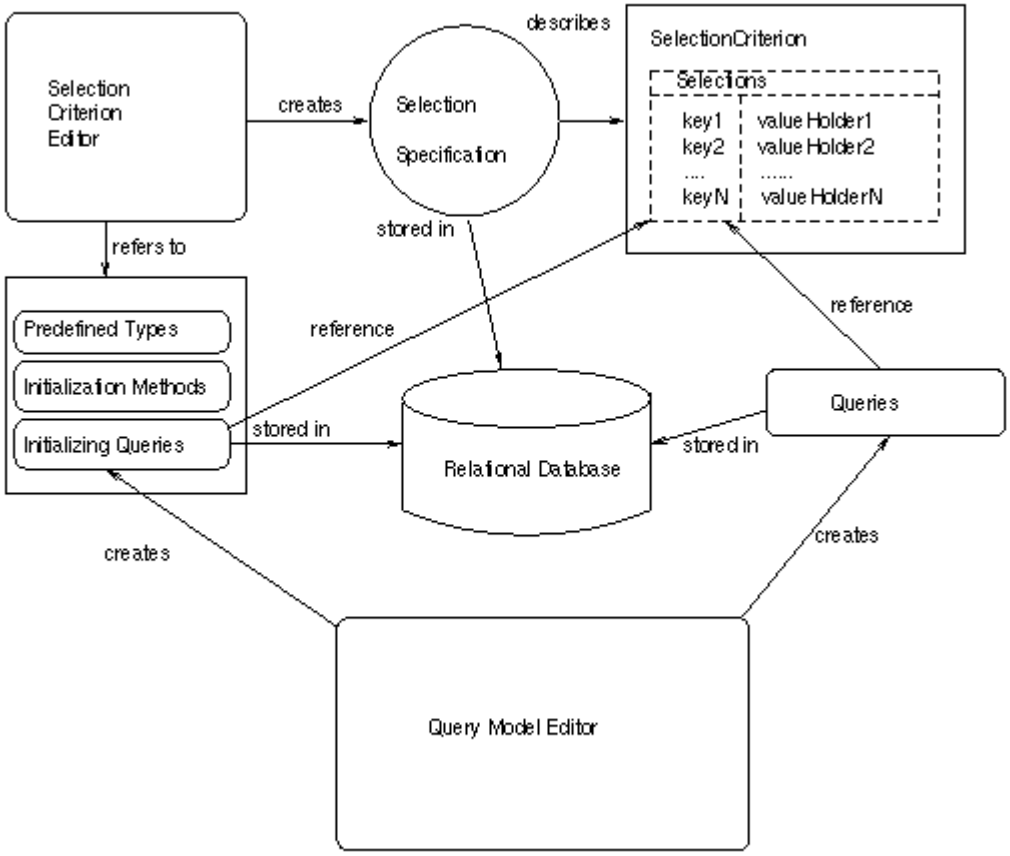

Figure 27 - *SelectionCriterion* Structure

access the value holder. For example, to access value holder whose key in the dictionary is "familyList", one simply calls *selectionCriterion familyList*, where *selectionCriterion* is an instance of the ***SelectionCriterion*** class. The class of a selection held by a value holder as well as its initial value at the time of financial application startup is specified using ***SelectionCriterionEditor***. A view of ***SelectionCriterionEditor*** is shown below.

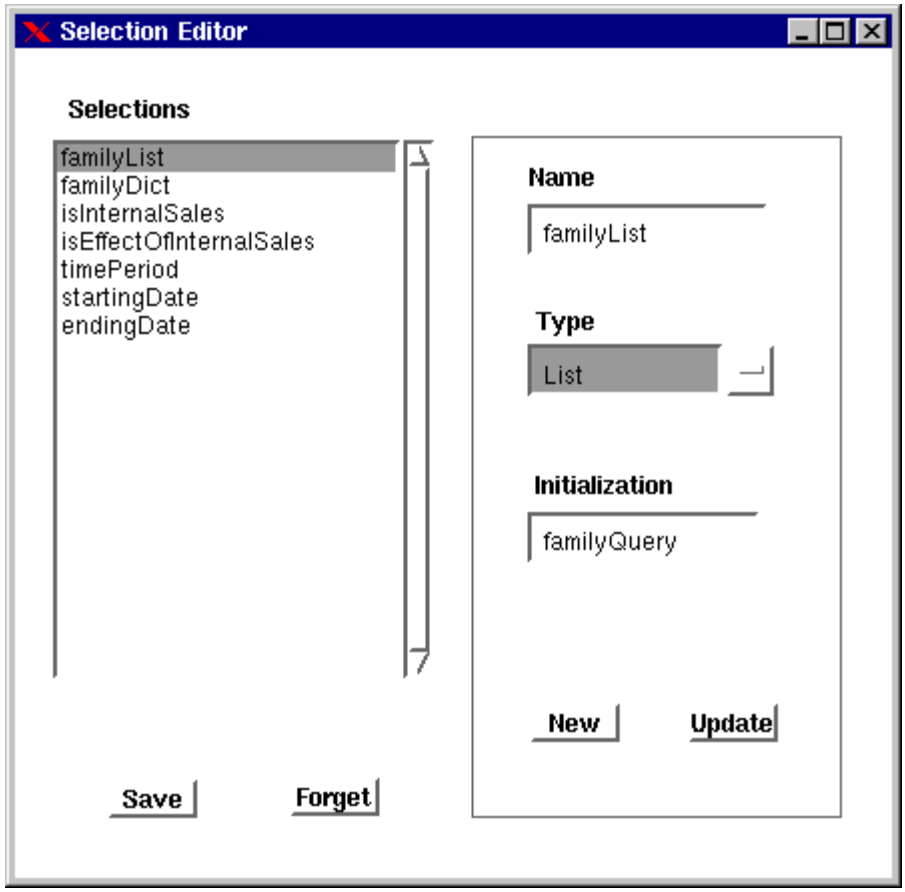

Figure 28 - *SelectionCriterionEditor* Editor

There are several conventions for specifying the initial value in ***SelectionCriterionEditor***.

1. If the type of the value is primitive (*Number*, *Boolean*, *String*, *Symbol*), the initial value is to be entered in the input field.
   For example, *false* is entered for the selection isInternalSales whose type is *Boolean*.
2. If the type of the value is ***Timestamp***, the initial value is specified by the name of an initialization method on the class side of the ***Timestamp*** class.
   For example, entering yearStart for the initial value of the startingDate selection specifies that "***Timestamp*** yearStart" will return the initial value for the startingDate selection.
3. In other cases, it is assumed that the type of a selection is a kind of *Collection*. That is, it needs to be initialized with a list. This is done by specifying in the initial value field the name of a query stored in the ***ReportQuerySpecs*** table. At the system startup, the query will be executed and the corresponding selection will be initialized with its results.
   For example, we specify familyQuery as the initial value for the familyList selection.

The queries referenced as initial values are specified declaratively using the query model editor. The query model editor provides an interface for constructing complex queries from simpler ones. The queries are then stored in the database and retrieved as need for them arises. Since queries comprise a significant part of business logic, such treatment makes it possible to modify and construct new business models while preserving application code intact.

4.9 FMState

ReportValues often need access to "global" information about other ***ReportValues***. These objects keep this information accessible by keeping a reference to a ***FMState*** object.

The state object is created by the security module when all of the security checks have passed correctly. The state is then passed to the startup window (***DuPontModel***, ***FMEditor***, etc.). The startup window passes the state the each drill-down, and each drill-down passes it to any window it creates. That ensures that window will have the state information, if needed.

A ***FMState*** holds a variety of types of information. To keep the information more manageable, it is stored in four different namespaces. The namespaces are values, selectionBox, windows, and applicationInfo.

values: This is where the business logic is cached. Although values is actually a dictionary, it is the default namespace for a ***FMState***. For example, state netSalesValues will search for #netSalesValues in the values dictionary. This trick uses Smalltalk's doesNotUnderstand: mechanism and greatly improves code readability, but is not necessary part of an ***FMState*** implentation.

w**indows:** This keeps a list of all windows currently open for that session. These windows are listed in under the Windows menu of a ***CatModelWrapper***. This lets the user easily just between windows of the same running application.

applicationInfo: This stores the ***ApplicationInfo*** object selected in the ***CatLoginDialog.***

selectionBox: This stores the ***SelectionCriterion*** object for this state's session.

4.10 ApplicationInfo

Not all of the application's description can be put in the database. The data which tells how to setup security is needed before a connection to the database has been made. Also the location of the database and the name of the data model are needed to make the connection to the database. Finally, if something goes wrong, the user will need to know the system administrator's name and phone number. This data is stored in an ***ApplicationInfo***.

In the ***CatLoginDialog***, the user selects which ***ApplicationInfo*** to use. This startup data is loaded from a configuration file stored on disk. This file is created by the system administrator through the ***SecurityAdmin*** tools, and it is encrypted to prevent malicious users from tampering with the file by hand.

During the login process the ***CatLogin*** security module store additional important information in the ***ApplicationInfo***. Examples include a ***QueryDataManager***, the user's list of editable families, and the user's login name.

After the login process the ***ApplicationInfo*** is passed to the ***FMState*** for future access. ***ApplicationInfo***'s values are set up as a namespace so the code state applicationInfo username looks up #username in the the ***ApplicationInfo***'s values

4.11 Sessions

Sessions are used to hide data from many simultaneously running applications while sharing the data among many objects in one other application. This is achieved by passing a reference of the *session* object to every new object in the same *session* which might need it. In the financial model, ***FMState*** objects serve as *session* objects

Using the state in this way involves some tradeoffs.
Drawbacks:

- The code is more complex because the *session* must be passed around to each object that is created instead of having the code refer to a global location.
- Passing the whole *session* around means some objects have access to extra information which they do not need.

Benefits:

- The user can run multiple applications simultaneously. This is achieved because the "global" information for a running application is always stored locally. This allows each running application to have its own private "global" information, shared throughout that application.
- The user can have multiple *sessions* of the same application running simultaneously. Each has independent, local data.
- Passing the whole *session* simplified the interface between classes. Otherwise, each value that is need by a new object, *plus* the each value needed by objects which the new object will create, have to be passed individually to the new object.

In the financial model, a *session* is created whenever a user successfully logs in for particular application. A user can open multiple *sessions* for the same application by specifying a new ***SelectionCriterion***. A new ***FMState*** will be built for the new selection and the old state's ***ApplicationInfo***. The windows for the new ***FMState*** will be empty and its values will be re-queried from the database. This way, the new ***FMState*** holds all of the data for the new session. A user can also open multiple *sessions* for different applications simply by going through ***CatLogin*** again.

Session pattern in the financial model some pitfalls which are not necessarily drawbacks. They are just things to be careful about when applying the pattern.

- Since classes like ***QueryObject*** are application independent, they should not be constrained by adding a state instance variable. If information from the state's ***ApplicationInfo*** is needed, then that information must be made truly global. Since this information is normally only used when saving query specs to the database, this causes the restriction of specifying only one application at a time. An alternative solution is to make an application specific extension which takes the necessary information as arguments.
- In any case, the ***QueryObjects*** must be created specifically for the ***QueryDataManager*** stored in the application's state. This means the creator must know the current *session* and must call a different instantiation for the ***QueryObjects***.

4.12 Namespaces

Namespaces are used to separate information. Typically, a *namespace* will hold a set of information that has the same type, is used together, or is created by the same object. *Namespaces* are an organizational tool that put conceptual walls between objects which might be indistinguishable or otherwise confusing if stored together. Sometimes *namespaces* simply prevent a naming conflict so multiple, unique objects can share the same base name. For example, "NetSales" could refer to a window, a query, or a set of values, depending on which *namespace* is searched.

This is achieved by storing the values in separate dictionaries. Optionally doesNotUnderstand: can be redefined. There are several tradeoffs with this technique.

Drawbacks:

- The code is misleading. In the example below, in appears that the ***ApplicationInfo*** has a username: method, when actually doesNotUnderstand: is actually catching the exception.
- Its not immediately clear which values will be put in a *namespace*. This information is clear when instance variables are used.
- There is no guarantee that only relevant items are put in the *namespsace*.
- doesNotUnderstand: incurs a minimal performance overhead. The performance is most likely too small to be an issue, however.

- Items which have names equivalent to important system messages cannot be put in a *namespace* unless values at:put: is explicitly used.

Benefits:

- The code is much more readable.
- All of the drawbacks can be avoided by using a dictionary without redefining Smalltalk's doesNotUnderstand: for the *namespace* object.
- The code is more flexible. If more information needs to be stored, it can be put in a *namespace* without writing new code for the object holding the *namespace*. This single benefit is the principle reason for using this pattern. *Namespaces* are the solution for a flexiable, dynamically changing system in which many values have to managed.

In the financial model, a ***FMState*** holds four *namespaces*: values, windows, applicationInfo, and selection**.** Each *namespace* is implemented as a dictionary. Undefined messages sent to the object which hold the *namespace* are forwarded to the dictionary. This makes the code more readable (state applicationInfo username: 'jack' instead of state applicationInfo values at: #username put: 'jack').

4.13 SummaryReport Framework

The SummaryReport Framework uses the Model-View-Controller pattern [BMRSS 96]. This framework provides an easy way to build a spreadsheet-type report for viewing high-level summaries of the transactions

You specify the list of values, label, columns to group on, columns to sum on, formats, alignments, extra calculated columns. A query can also be provided that is automatically converted to the list of values.

Given the above, a row-column report is generated with the necessary constraints for automatically updating itself if the query or values it is dependent upon changes. This spreadsheet allows for individual rows to be selected, columns to be resized, ordering of columns, and allows for printing. You can also add pop-up menus to get a desired action for a specify row.

The printing package does nice printing with headers, footers, page-numbers, smooth column/row breaks.

There are three classes that collaborate to make it work. SummaryReportModel, SummaryReportView, and SummaryReportController. The view and controller classes are just like the normal view and controller from MVC in that the controller handles input and the view displays it. The model of a SummaryReportView is a SelectionInList.

The SummaryReportModel is an ApplicationModel that contains a view holder for the SummaryReportView. It also contains several support methods for accessing menus, printing, etc.

What happens is that the SummaryReportView is created using a ReportSummarySpec. This creates the summary rows that will be displayed. A RerpotSummarySpec contains all of variable parts specified above. This gets pluggged into the SummaryReportModel for later display.

The view lazily creates the controller when it is needed. Once the view is opened by displaying the SummaryReportModel's window, the models values are drawn on a GraphicsContext using the standard MVC provided by VisualWorks.

4.14 Graphing Framework

This framework extends the VisualWorks Business Graphs to dynamically display any two-dimensional collection of numbers in graph form. In addition, the labels for the X and Y axes must also be given (the Y axis labels are the legend). In addition to this basic feature, there are several extensions.

1. Create a dependency between the graph and the collection, so that when the collections values change, the graph automatically updates itself.
1. Pass an entire table instead of a collection of values, in which case the dependency between the table and the graph is set up automatically.
1. Display the results of a query in a graph. Again, the dependency between the query and the graph is set up automatically.
1. Display the results of a query for each individual month. The dependency is set up automatically.

Several classes handle the graphing framework. ***TableGraph*** handles the graph display. It is able to handle the first two features listed above. This class holds another class called ***GraphHolder*** as a subcanvas, which in turn holds the VisualWorks graphics view. ***TableGraph*** makes a dynamic version of the VisualWorks graphics package. Because the package requires the specifications of legend, type of graph, legend placement, among other things, to be determined as part of the spec, the specifications cannot change. ***TableGraph*** allows all these features to be changed at run time.

QueryGraph is a subclass of ***TableGraph*** . It displays the graphs that are calculated through ***GraphMaker***, which makes a graph that is dependent on element specs, which in turn are dependent on queries. ***GraphMaker*** handles the last two features above. The ***GraphMaker*** holds a two-dimensional collection of block values to display. It is also possible to perform simple arithmetic operations on ***GraphMaker***. The values of the two ***GraphMakers*** must have the same dimensions

Its subclass, ***SingleGraphMaker***, gets the value from the result of a set of element specs. The resulting values of each element spec is on the X-axis. Each element spec should return a group of values. The group should match the Y-axis. ***MonthlyGraphMaker*** is a subclass of ***SingleGraphMaker*** that must use one element spec whose value returns the date of each row along with the value to graph. Based on the current selections of date in ***CatState***, it calculates the values for each month. The X-axis is ordered by the months.

Composite***GraphMaker*** is the result of applying arithmetic operations on ***GraphMaker***.

4.15 DetailedReport Framework

This framework handles the display of the lowest-level query, where each row of the tables of the database is displayed. Its features include:

1. Display the rows returned from a query.
1. Let the user edit the rows.
1. Before letting the user change the rows, pop up a dialog that asks the user to specify some information.
1. Search for a string in all the data, just specific columns.
1. Sort by a column.
1. Go to the nth row.

DetailedTable displays the rows returned from a query. It actually has all the methods for row editing as well, but they are turned off by default. It does not query all the rows at once, but more rows are obtained as the scrollbar is pulled down. The user can specify the number of rows to return. The information that the detailed table uses is passed through a class called ***QueryDescription***.

The subclasses tend to be somewhat query specific. Each one assumes that certain fields exist in the query. ***IncurrenceRatesTable*** is used for incurrence rates tables. It allows the rows to be edited. ***EditableDetailedTable***, on the other hand, is a more general class for editing rows of tables. ***SimpleErrorsTable***, on the other hand, is more extensive because it not only edits the original table, but it also creates a row in the error table corresponding to the original table. It is assumed that the error table has at least all the fields of the original table. ***DialogErrorsTable*** is even more extensive because it requires the user to enter some information which will be stored in the error table along with the original row information. ***ErrorCorrectionTable*** does the reverse of ***DialogErrorsTable*** by displaying an error table, and when a row is

edited, the error row is deleted, and the edited row is written to the non-error table. The error table's fields must have all of the fields of the non-error table.

4.16 Printing Framework

The Printing framework is design to print up all of the reports except the Summary Reports which know how to print themselves.

PrintFileInterface is a standard print dialog that is opened when the end-user wants to print something. It collects some of the basic information from the user about page-numbering, header, footer, orientation and then calls ***PrintFile*** to print the view. ***PrintFile*** does the printing work. It takes the model requested to print, the page orientation, the print medium, the title, the footer, the report date, and possibly disclaimer and filename for printing. ***PrintFile*** then initializes everything and prints on the specified graphics context which could be a printer device or file

4.17 Testing Framework

Testing framework is designed to perform testing of DuPont values and different level "drill-downs". In the core of the framework is the TestObject class. The class keeps both TestCriterion, which virtually stores a SelectionCriterion instance, and a TestData collection, a collection of TestData subclass instances.

To test new kind of report values, it is necessary to perform the following operations:

- Create a subclass of TestData. The class instances should know how to create themselves from the current state of the financial application.
- In the default portfolio create a test TopTest for testing the new TestData subclass.

Aside from the TestObject, TestData subclasses, and TestCriterion, there are several support classes: TestSelectionDialog, EqualityTestInfo, TestResultViewer. They are usedfor selecting tests from a TestData collection, storing test result information, and viewing test results, respectively. See each class for detailed descriptions.

5. Security Module

The security module follows some specified requirements to restrict access to the financial data. Different applications could have different security requirements, so the security modules was designed to make it easy to plug in additional security checks and easy to turn some security features on and off. The default security checks include forcing the user to enter a new password after a certain period of time and disabling access from a machine or account after a certain number of login. After passing the active security checks, the security module starts the process of extracting all of the information from the database and organizing it into independent sessions.

5.1 Security Requirements

This list provides some of the types of security which Caterpillar was looking for in the financial model.

1. Passwords must be at least five characters long.
1. Passwords should not be viewable in clear text form.
1. Passwords should expire after a specified period, forcing the user to select a new password.
1. The new password should not be a repetition of any of the user’s recently used passwords.
1. Users can only view a specified list of products.
1. Users can only edit a subset of the specified list of products.
1. A user’s account is disabled after 3 consecutive failed login attempts
1. Login is only possible from specified machines.

1. A machine is disabled after 3 consecutive failed login attempts.

5.2 Components

FMLogin manages the security and login process.

Several user interfaces for system administration and user login have been designed with security in mind. They are in the CAT-SecurityGUI category. ***ChangePasswordDialog***, ***NewUserDialog***, ***FMLoginDialog***, ***ValidNodesDialog***, and ***UserPropertiesDialog*** are some examples of security support user interfaces. Most are used with the ***SecuirtyAdmin*** tool or during the ***FMLogin*** Process.

A ***FMState*** is created at the end of the login process. It serves as the basis for separating sessions and sharing data among a session's ***ReportValues***.

The ***ApplicationInfo*** class stores all of the information necessary to configure the security process, connect to the database, and start the application. It also includes some extra information such as the name and phone number of the system administrator. Because this information is needed before a connection is made to the database, it is persistently stored in .cfg files in a specific configuration file directory.

The ***Cryptor*** prevents users from changing ***ApplicationInfo*** configuration files and security tables by hand. It has encrypt: and decrypt: methods which take a string as input and return encoded and decoded strings, respectively. Using this class forces administrators to use the Administrator Application to make changes.

Several database tables have been added to the reusable part of the data model to persistently store user and machine specific information related to security. They are: `Accessible_Products`, `Editable_Products`, `Old_Passwords`, `Cat_User_Prof`, and `Valid_Nodes`.

5.3 How security requirements are met

1. Passwords length is required to be > 5 by the ***ChangePasswordDialog*** and ***NewUserDialog***.
1. ***FMLoginDialog***, ***NewUserDialog***, and ***ChangePasswordDialog*** use password text fields that display asterisks instead of what the user is typing. Old passwords are processed by the ***Cryptor*** class before they are written to the `Old_Passwords` table. Database account passwords are used for current passwords, so they have the visibility protection provided by the database.
1. ***ApplicationInfo*** stores a passwordAgeLimit. If the `change_date` in the user's `Old_Passwords` record is more than passwordAgeLimit days old, ***FMLogin*** opens a ***ChangePasswordDialog*** to force the user to change the password.
1. The new password is compared to the user's 12 most recent records in `Old_Passwords`. The password change is only successful if the password is unique.
1. ***SelectionCriterion*** generates the list of families for the selection box from the user's records in `Accessible_Families`. All querying is based on a ***SelectionCriterion*** and thus is restricted to this list of families.
1. Detailed windows are the only places where data can be saved to the database. Before an editable detailed window is opened, it checks to make sure all of the selected families are in the user's `Editable_Families` record. If not, the window is opened in read-only mode. state isEditable determines if the editable families are a subset of the selected families.
1. A user's `Cat_User_Profile` record keeps a list of `login_failures` which is cleared when a login is successful. If `login_failures` is greater than 3, the enabled column is set to false and the current time is stored in the `disabled_time` column. The user's account will be automatically re-enabled after a specified elapsed time.
1. ***CatLogin*** makes sure the machine's IP address is stored in the `Valid_Nodes` table.
1. A machine's `Valid_Nodes` record keeps a list of `login_failures` which is cleared when a login is successful. If `login_failures` is greater than 3, the enabled column is set to false and the

current time is stored in the - column. The machine will be automatically re-enabled after a specified elapsed time.

5.4 Other Requirements

Users can run several, possibly different, applications, in the same image. Each application executes independently.

5.5 Security Data Model

There are five security tables in the data model: `Accessible_Products`, `Editable_Products`, `Old_Passwords`, `Cat_User_Prof`, and `Valid_Nodes`.

`Accessible_Products: username(FK), products`
- This table is used to restrict which products or families a user can select and view.

`Editable_Products: username(FK), products`
- This table is used to restrict which products or families a user can edit.

`Old_Passwords: username(FK), password, change_date`
-This table is used to force the user to create new passwords after a specified time has passed

`FM_User_Prof: username, enabled, disabled_time, login_failures`
- This table is used to temporarily disable user accounts

`Valid_Nodes: node_address, enabled, disabled_time, login_failures`
- This table is used to temporarily disabled access to the data model from a machine

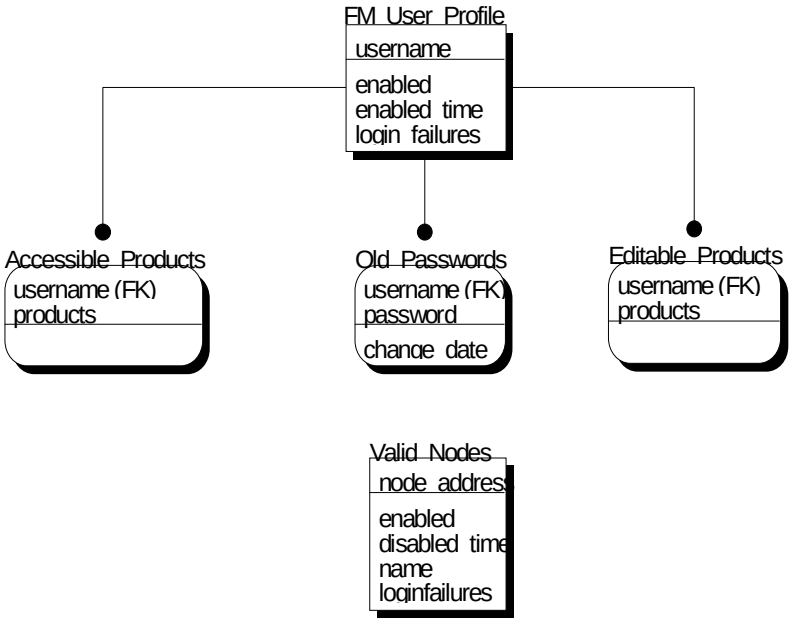

Figure 29 - Security Data Model

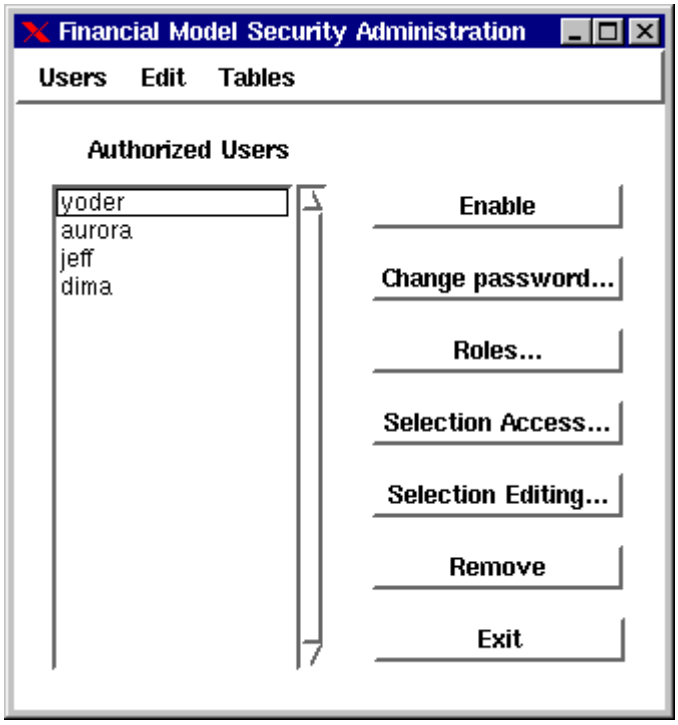

Figure 30 - Security Admin

5.6 SecurityAdmin Tools

This set of tools is available to any user with a 'dba' role. With these tools, the security administrator can:

1. Create a user (***NewUserDialog***)
1. Remove a user
1. Enable a user's profile
1. Edit a user's roles (***UserPropertiesDialog***)
1. Edit a user's viewable and editable products (***UserPropertiesDialog***)
1. Add, remove, and enable a machine (***ValidNodesDialog***)
1. Grant role access for new tables (a sql script)
1. Create public synonyms for new tables (a sql script)
1. Create/edit/remove application initialization files (***ApplicationInfoDialog***)

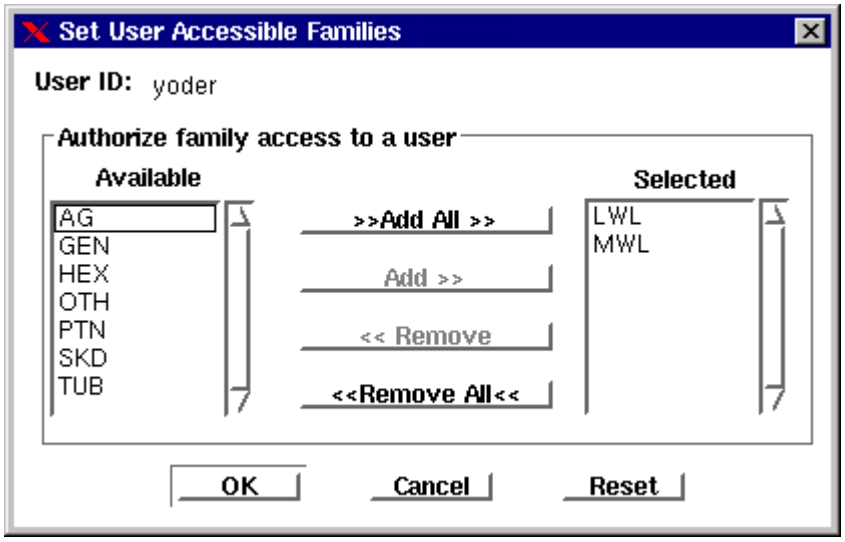

Figure 31 - UserPropertiesDialog

5.7 FMLogin Process

The login process involves several stages. A failure at any stage raises a signal to stop. Depending on the timing and severity of the cause, the login process could be halted immediately, the user's account or the machine's access privileges could be disabled, or the process might simply restart.

FMLogin first locks itself as a critical section. Only one ***FMLogin*** security verification process is permitted at a time. After the security verification has completed (successfully or unsuccessfully), another login can be attempted. The process then proceeds to a login failure loop, other security and the financial model startup routines.

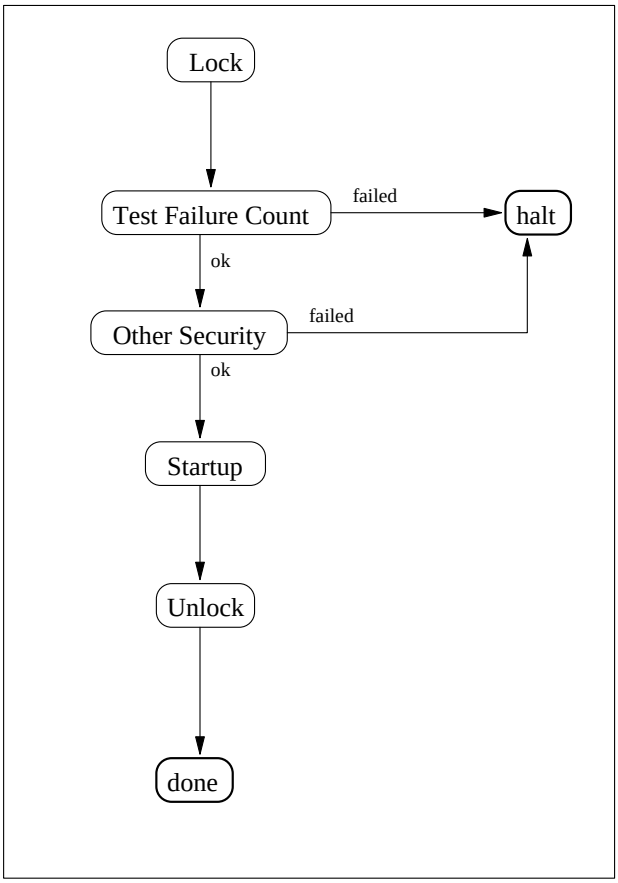

Figure 32 - Login Process

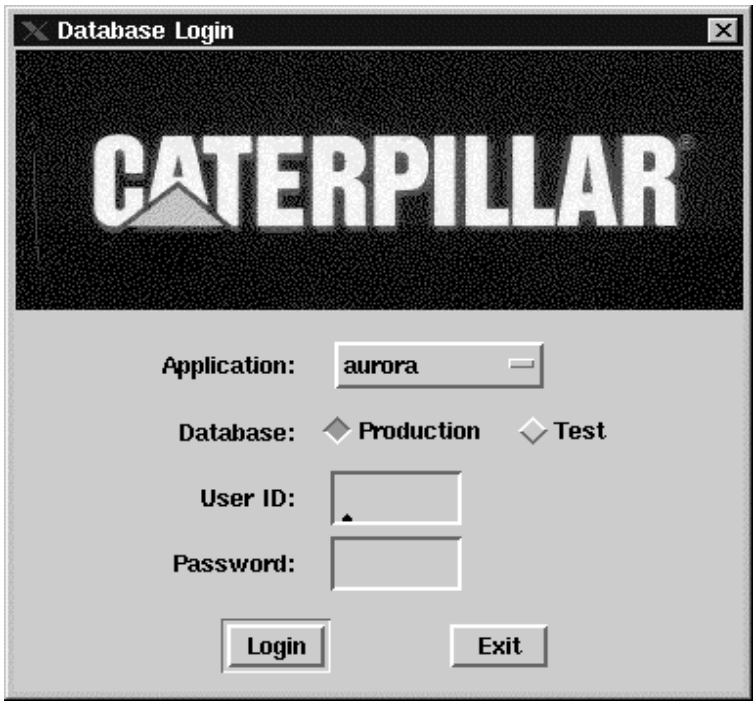

Figure 33 - FMLoginDialog

Initially, the user sees a dialog window for logging in (***FMLoginDialog***). The Application is the name of the ***ApplicationInfo*** configuration file to load. Many applications could support a production and a test database, so easy access to two databases is provided in the login dialog. The user must also type in a valid user id and password.

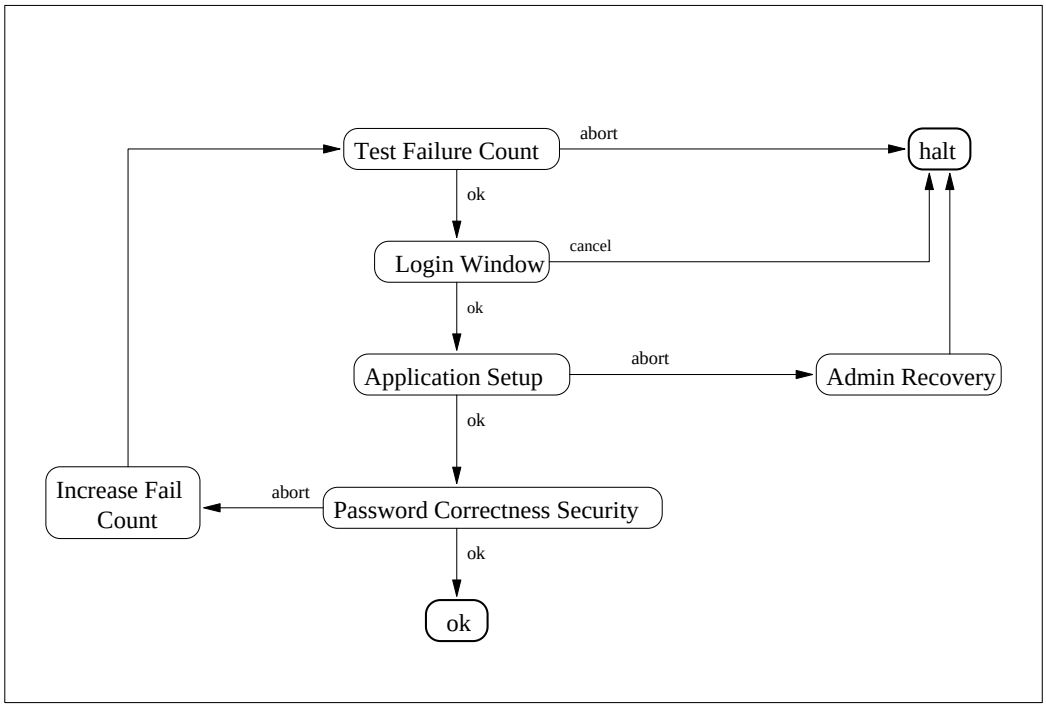

Figure 34 - Login Failure Loop

After entering this information, ***FMLogin*** attempts to setup the system and login with the username and password. After three consecutive failures, the login process aborts immediately.

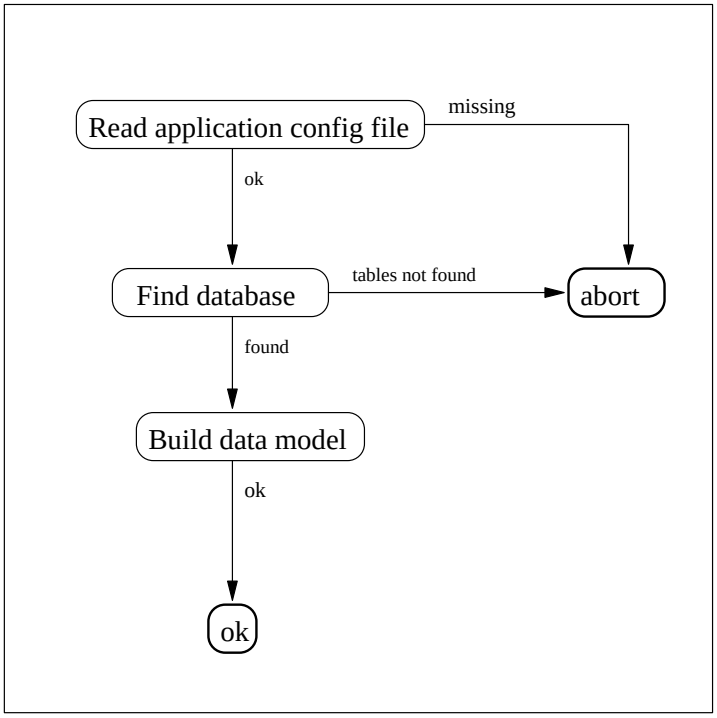

Figure 35 - Application Setup

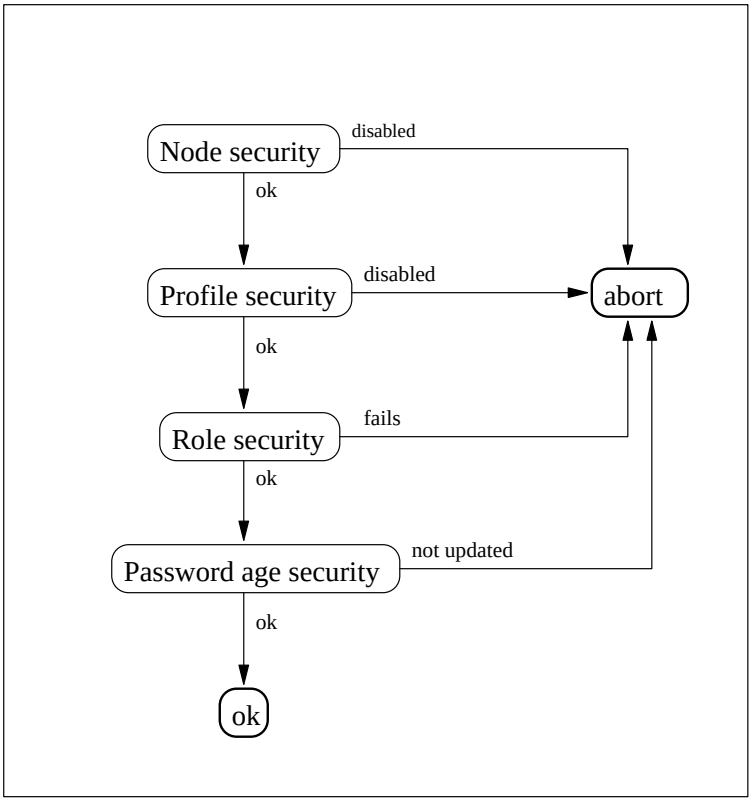

Figure 36 - Other Security Checks

Application setup read a .cfg file from the config directory and stores the information in an ***ApplicatoinInfo***. Then it verifies that some tables exist in the database. If the database's data model has changed, it will rebuild the client's data model to match

.At this point, the user has a successful database connection. During the next stage of the login process, ***FMLogin*** tries employs other security techniques to valid the user before giving them full access to the financial model. The ***ApplicationInfo*** has a parameter which determines which checks are performed

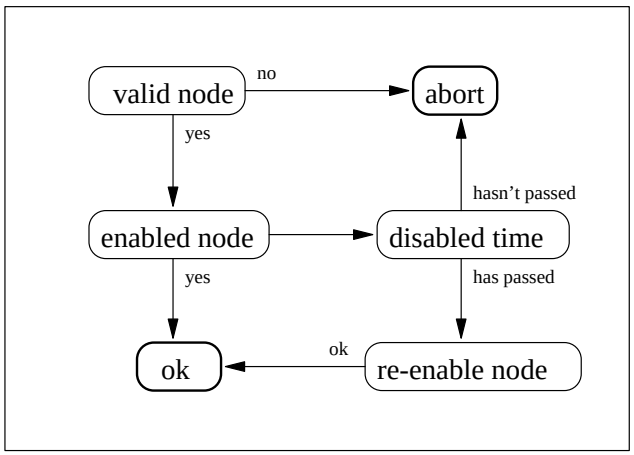

Figure 37 - Node security

Node security ensures only designated computers can keep a connection to the database. (A node is an IP address). This check is done by first retrieving the record for node from `Valid_Nodes` database table. If the node is not in the table the login process is aborted. If the retrieved record's enabled column is false, the disabled time is checked. The node can be re-enabled if enough time has elapsed since the disabled time. This has the advantage of providing disrupting a hacker's attempts while not necessarily forcing a system administrator from intervening to re-enable a node. If not enough time has passed, abort the login process. When a successful login occurs, `login_failures` is reset to 0.

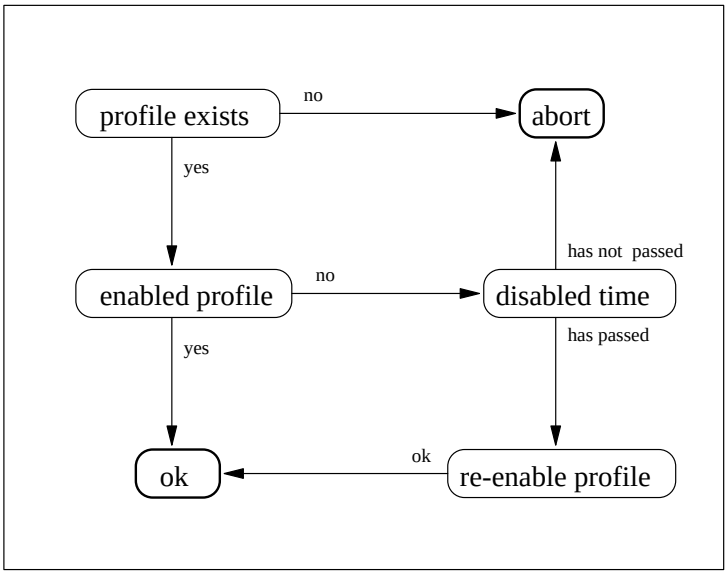

Figure 38 - Profile Security

Profile security is similar to node security except that instead of disabling machines, it disables a user's account. The user's profile is retrieved from the FM_User_Profile table. Otherwise, this security check behaves exactly like Nodes security.

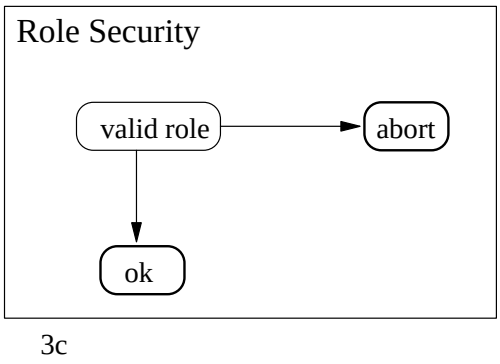

Figure 39 - Role Security

Role Security allows different types of users to access different parts of the financial model. Roles are actually implemented by Oracle, but the financial model extends the roles' purpose while limiting the roles which actually play a part in security. The four primary roles are: dba, accountant, modeler, and developer. The dba can read, write, and change any data model and can use the ***SecurityAdmin*** tools. The developer can read, write, and change any of the data models. The modeler has read and write the business logic and GUI specification data models and can run the ***FMEditor***. The accountant can read the business login and gui specification models and can read and write the business unit specific and security data models. Write privileges are constrained by the application (only personal information in the security model, only editable products in the business unit specific model). The accountant can also run the financial model.

Other security checks can readily be added at this point of the login process.

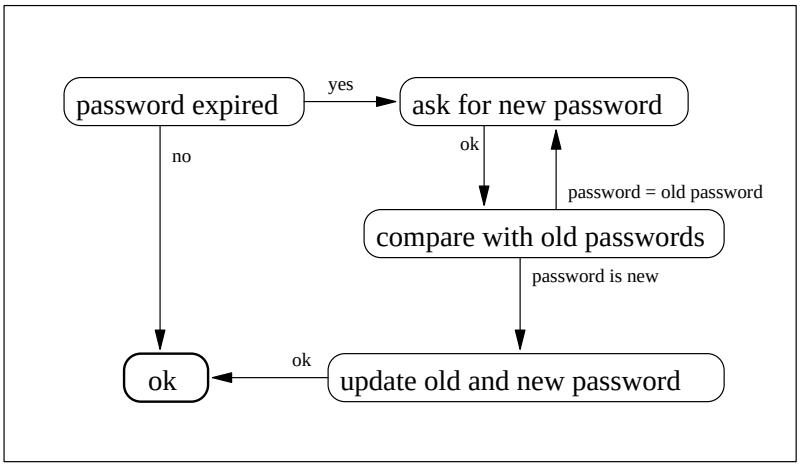

Figure 40 - Password Security

If the password has expired, open a ***ChangePasswordDialog***. If the new password is listed for that user in the Old_Passwords table, force the user to select a new password by reopening a ***ChangePasswordDialog***. When the new password is entered add the old password to the Old_Passwords table and set the new password for the user.

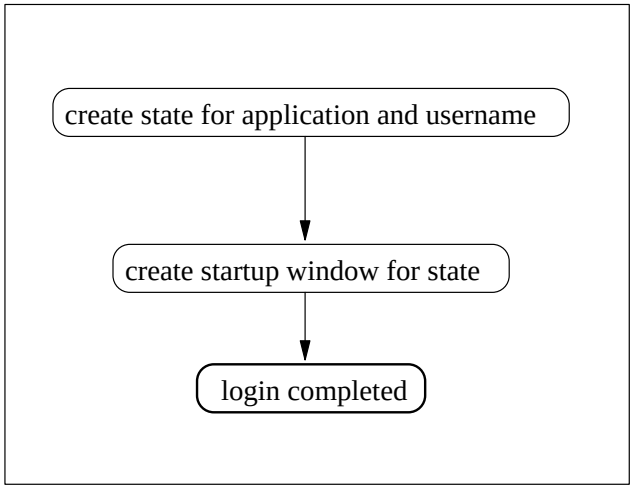

Figure 41 - Financial Model Start Up

At this point, ***FMLogin*** has finished its security checks. Using information from ***ApplicatoinInfo*** and a user's selected active role, ***FMLogin*** creates a ***FMState*** passes it the startup class (***DuPontModel, FMEditor,*** or ***SecurityAdmin***), opens the startup class, and unlocks itself.

6. Future Work

There are many extension to the domain-specific visual language that could be added. For example, you might image that you want to be able to reuse tables, element specs, and report values more globally. This would provide for a finer grain of development. Thus, the developer can make a set of ReportsValues used in different applications but associate different menus, calculated columns etc. They could also create queries that can be reused in different ReportValues. Of course there are tradeoffs with this as the developer will have to know more and try to keep track of global information during the design.

We also took into consideration that the application could be distributed. We believe that the architecture allows for this, but there may be some needed changes to allow for distributing the business logic correctly, especially if you want to share these and have views automatically updated for multiple users when the business logic or GUI descriptions change.

This application has logically and architecturally been developed as a three-tiered application. The current implementation in Smalltalk only used two real tiers since we could easily let the business logic and GUI's live in the same place. However, it would be easy to either use Gemstone or DST and move the ReportValues to a middle-tier.

7. Summary

What we have described here is the architecture and design of a domain-specific visual language for building financial models. There has been many reusable components integrated into the system. The overall architecture is language independent and could be developed using any general purpose language desired.

Our architecture was developed around a object-oriented framework written in Smalltalk [Goldberg & Robson 1983]. Smalltalk allowed us to quickly develop working prototypes, and get immediate feedback from our users. Since VisualWorks is robust enough for production use, these prototypes could evolve into production applications.

8. Patterns

The following is an incomplete list of the patterns that being used in the Financial Modeling Framework.

All of the Reports patterns by Brant and Yoder
Sessions/Namespaces patterns by UoI to be developed
Adapter by GOF BETWEEN VALUES AND QUERY OBJECTS
Builder by GOF REPORT VALUES BUILD QUERIES AND REPORTS AND VALUE-MODELS
Command by GOF QUERY OBJECTS ARE SOMEWHAT COMMANDS
Composite by GOF QUERY OBJECTS and POSSIBLY THE GUIs
Decorator by GOF QUERY OBJECTS
Factory Method by GOF REPORT VALUES
Interpreter by GOF REPORT VALUES AND QUERY OBJECT
Observer by GOF QUERY OBJECTS
Singleton by GOF DONE BY STATE SOMEWHAT...SINGLE ACTIVE STATE OBJECT and a SINGLE ACTIVE SECURITY MODULE
Patterns by Kent Beck -Used throughout....just good smalltalk patterns
Template Method by GOF QUERY OBJECTS REPORTVALUES
Visitor by GOF DuPont Model to build code and write to database....walks through interface widgets.
Constraints by Johnson USED IN THE VALUEMODELS AND QUERY OBJECTS
Evolution, *Architecture*, and *Metamorphosis* patterns by Foote and Yoder
Some of the *Selfish Class* patterns by Foote and Yoder
Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks Authors: Don Roberts and Ralph Johnson, University of Illinois USED DURING THE DESIGN AND DEVELOPMENT OF THE SYSTEM
Checks by Ward Cunningham I believe we use patterns 1-6 of his paper.
Understanding ValueModels by Bobby Woolf
Null Object by Bobby Woolf and Ralph Johnson USE FOR NULLREPORTVALUES
Stars: A Pattern Language for Query Optimized Schema by Steve Peterson are patterns that we did use in our data modeling of the business. We really don't describe that in this paper but it was done and might be interesting to note as a side-point especially when we talk about designing a database for the company.
Ward Cunningham. "The WyCash Report Writer," OOPSLA '92 Workshop, "Towards an Architecture Handbook." http://c2.com/doc/ooplsa91.html DOING SIMILAR THINGS IN OUR SUMMARYREPORTMODEL

9. References

[Brown & Whitenack 95] Kyle Brown and Bruce G. Whitenack. "Crossing Chasms - A Pattern Language for Object-RDBMS Integration," *PLoP'95 Proceedings.*

[Cunningham 91] Ward Cunningham. "The WyCash Report Writer," OOPSLA '92 Workshop, "Towards an Architecture Handbook." http://c2.com/doc/ooplsa91.html

[Foote & Yoder 95] Brian Foote and Joseph Yoder. "Evolution, Architecture, and Metamorphosis," *PLoP'95 Proceedings.*

[Brant & Yoder 96] John Brant and Joseph Yoder. "Reports" *PLoP'96 Proceedings.*

[BGL 96] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis. *Visual Object Oriented Programming, Concetps and Environments*, Manning, 1995.

[BMRSS 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern - Oriented Software Architecture, A System Of Patterns*, Manning, 1996.

[Fowler 97] Martin Fowler. Analysis *Patterns, Reusable Object Models*, , Addison-Wesley, 1996.

[Foote & Yoder 96] Brian Foote and Joseph Yoder. "Attracting Reuse," To appear in *PLoP'96 Proceedings.*

[Goldberg & Robson 1983] Adele Goldberg and David Robson, *Smalltalk-80:The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983

[GHJV 95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Johnson 92] Ralph E. Johnson. "Documenting Frameworks with Patterns," *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10): 63-76, Vancouver BC, October 1992.

10. Appendix - The Farm Data Model

The farm data model was created as an example of our framework being used (see Figure 42). The farm data model handles the financial transactions that would occur on a typical farm. To keep the model simple, several assumptions have been made.

- All components of the farm are considered assets including the farm itself.
- The farmer and his family comprise the entire work force. The cost of any work done by an outside party is related to the asset on which the work was done rather than to the individual or organization providing the service.
- Income comes solely from the sale of assets.

- The only products produced are crops.

The collection of `assets` constituting the farm have been separated into four categories: `Land`, `Crops`, `Supplies`, and `Capital_Goods`. Each type of asset has an identifying field, an adjustment date, a quantity, and a value. The identifying field allows for the categorization of the asset types. The adjustment date keeps track of when the quantity or value of an asset has changed. The value field represents the total value invested in all the assets with a given identification field as opposed to the unit value. The complete inventory of all assets is kept in another table called `Inventory`. All data in `Inventory` can be accessed on a monthly basis. Any time a transaction occurs involving `Land`, `Crops`, `Supplies`, or `Capital_Goods`, `Inventory` is automatically updated.

The `Supplies_Used` table manages the depletion of supplies used in production. All supplies must be directly related to the production of crops. This constraint simplifies handling the cost of using a supply and attributing it to some `plot_id`. `App_date` is the date of application and qty specifies the amount of the supply used. `App_date` provides a means of keeping a history of supply use. `Qty` is necessary for updating the appropriate asset record and calculating the cost.

The `Production` table is very similar to the `Supplies_Used` only it keeps track of the amount crops grown on a plot in a given season. Notice that `crop_id` is not part of the primary key so only one type of crop can be grown on a given plot of land in any one season. This was done to keep the profit analysis based on a plot of land simple. Any asset produced which is not sold becomes part of the inventory resulting in an increase in the quantity field of the corresponding asset record .

Income is generated based on the sale of an asset. The `Income` table handles such transactions. The number of assets sold with a common identification as well as the date of sale and total revenue generated is stored.

There are two types of costs modeled. The `Period_Costs` table stores the data pertaining to a cost which occurs periodically such as taxes. The other type of costs, variable costs, is handled by the `Var_Costs` table. The only difference in the data stored in the two tables is a quantity field (`qty`) inthe `Var_Costs` table. It allows for the purchase of several assets at the same time. Costs can be either production or non-production related. The type field determines which is the case. The `dollars` field represents the total cost in the case of `Var_Costs` rather than the cost per `qty`.

Farm Data Model

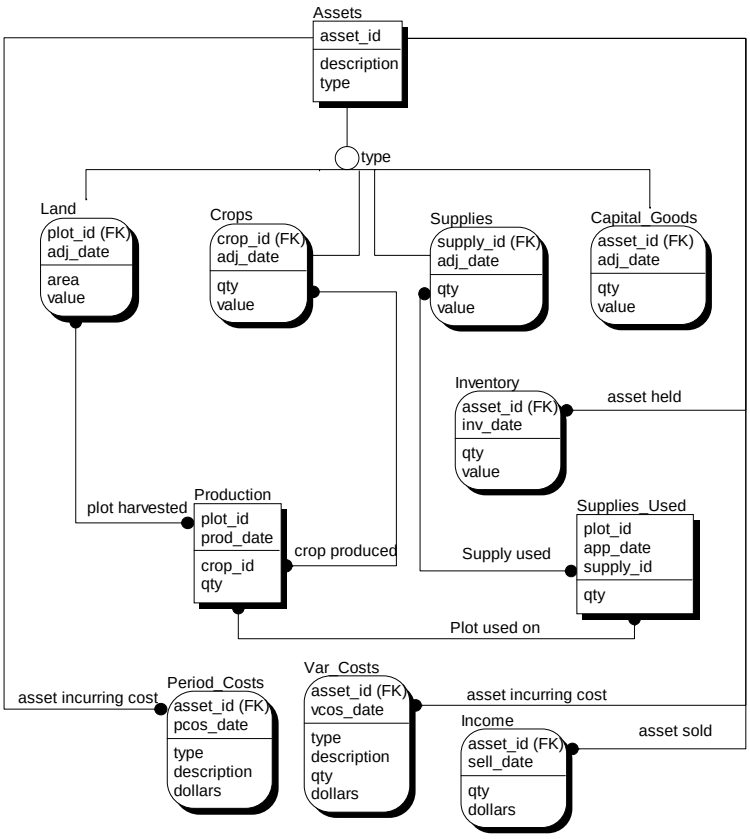

Figure 42 - Farm Data Model

Queries may be constrained by selection criteria chosen by the user. If the selection criteria is used, the queries may be constrained by any combination of a *crop_id*, *start_date*, and an *end_date*.

Income Queries:

Income Generated from a *crop_id* from *start_date* to *end_date*.

```
SELECT sum(dollars)
   FROM Income
   WHERE asset_id = crop_id AND sell_date >= start_date
      AND sell_date <= end_date
```

Income Generated from the sale of Supplies between *start_date* and *end_date*:

```
SELECT sum(dollars)
   FROM Income,Assets
   WHERE Income.asset_id = Assets.asset_id AND
      Assets.type = 'supplies' AND Income.sell_date >=
      start_date AND Income.sell_date <= end_date
```

Similar queries can be made for the sale of Land or Capital_Goods by replacing the Income table with either Land or Capital_Goods.

Total Income Generated from all sales between *start_date* and *end_date*:

```
SELECT sum(dollars)
   FROM Income
   WHERE sell_date >= start_date AND sell_date <= end_date
```

Cost Based Queries:

Production Costs from start_date to end_date:

```
SELECT sum(dollars)
FROM (SELECT asset_id, vcos_date, type, description, dollars
      FROM Var_Costs
      UNION ALL
      SELECT *
      FROM Period_Costs)
WHERE type = 'production' AND vcos_date >= start_date
   AND vcos_date <= end_date
```

Non-Production Costs can be found by substituting the comparison of type against 'production' with 'non-production'.

Supply Costs for a plot of land, *plot_id*, between dates *start_date* and *end_date*:

```
SELECT max(Supplies.adj_date),
   supplies_Used.qty/Supplies.qty*Supplies.value
FROM Supplies,Supplies_Used
WHERE Supplies.supply_id = Supplies_Used.supply_id AND
   Supplies.adj_date <= Supplies_Used.app_date AND
   Supplies_Used.plot_id = plot_id
GROUP BY Supplies.supply_id
```