A Framework for Financial Modeling

Joseph W. Yoder
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave
Urbana, IL 61801
j-yoder@uiuc.edu
http://www.uiuc.edu/ph/www/j-yoder

Sponsored by Caterpillar Inc. through the National Center for Supercomputer Applications

 1

Goals

- Case study of developing a framework
- Case study of using design patterns
- Learn a framework for financial modeling

 2

Overview

- A Framework Overview.
- What is a Financial Model?
- How we developed our framework.
- The design of our framework.
- Patterns in our framework.

Frameworks

- Interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or re-create than code.

  Peter Deutsch

- Difference between framework and component library for framework (components get plugged in and are concrete sub-classes that do all the work).

For detailed information about Frameworks attend Ralph Johnson's tutorial or see:
http://st-www.cs.uiuc.edu/users/johnson/

Frameworks Encode Domain Knowledge

- Frameworks solve a particular set of problems.
  - get different points of view
  - explain/defend current design
- Some frameworks are more technology (horizontal) frameworks verses application domain (vertical) frameworks.
  - Are we solving GUI's, object persistence verses more application domain related such as insurance or manufacturing.

Frameworks

- Framework is:
  - reusable design of an application or subsystem
  - represented by a set of abstract classes and the way objects in those classes collaborate.
- Use framework to build application by:
  - Creating new subclasses
  - Configuring objects together
  - Modifying working examples

Frameworks

- Framework prescribes how to decompose a problem.
- Not just the classes, but the way instances of the classes collaborate.
  - shared invariants that objects must maintain, and how they maintain them
  - framework imposes a collaborative model that you must adapt to.

Relevant Principles

- Frameworks are abstractions: people generalize from concrete examples
- Designing reusable code requires iteration
- Frameworks encode domain knowledge
- Customer of framework is application programmer

Generalize from Concrete Cases

- People think concretely, not abstractly.
- Abstractions are found bottom-up, by examining concrete examples.
- Generalization proceeds by
  - finding things that are given different names but are really the same,
  - parameterizing to eliminate differences,
  - breaking large things into small things so that similar components can be found, and
  - categorizing things that are similar.

Finding Abstract Classes

- Abstract classes are discovered by generalizing from concrete classes.
- To give two classes a common superclass:
  - give them common interface
    - + rename operations so classes use same names
    - + reorder arguments, change types of arguments, etc.
    - + refactor (split or combine) operations
  - if operations have same interface but different implementation, make them abstract
  - if operations have same implementation, move to superclass

Frameworks Require Iteration

Reusable code requires many iterations.

Basic law of software engineering

If it hasn't been tested, it doesn't work.

Corollary: software that hasn't been reused is not reusable.

White-box vs. Black-box

White-box ⟷ **Black-box**

Customize by subclassing	Customize by configuring
Emphasize inheritance	Emphasize polymorphism
Must know internals	Must know interfaces
Simpler, easier to design	Complex, harder to design
Harder to learn, requires more programming.	Easier to learn, requires less programming.
Easier to customize because you can overwrite the code	Harder to customize because you need to learn how objects collaborate

What is a Financial Model?

- reports
- answer “why”
- correct errors, enter budget
- depends on database
- ensure security

Answering Why

- answers questions about finances
  - profit, return on assets
  - detailed costs
  - compare actual, budget, predicted
- high-level and detailed
- fixed reports and ad-hoc queries

What is a Financial Model?

- business logic is equations
  - variable margin = net sales - variable cost
  - net sales = gross sales - warrantee
  - gross sales = sum *sales* column from *sales_and_transfer* table
- User interface just as important

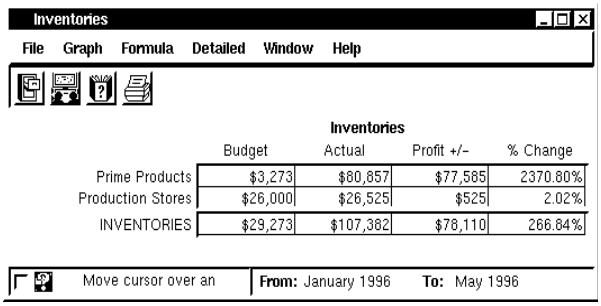
Top Level

Dupont Model

Warning!
All numbers are fake.

Inventories Drilldown

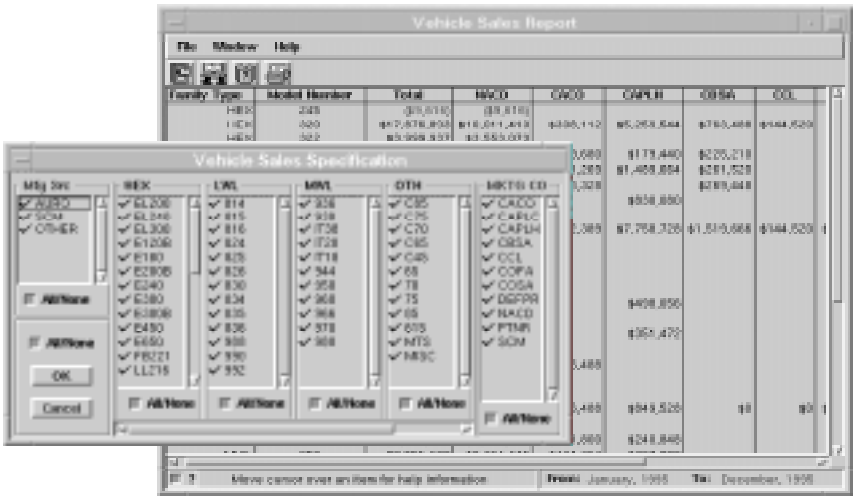
"Show calculation for Value"



 17

Summary Report

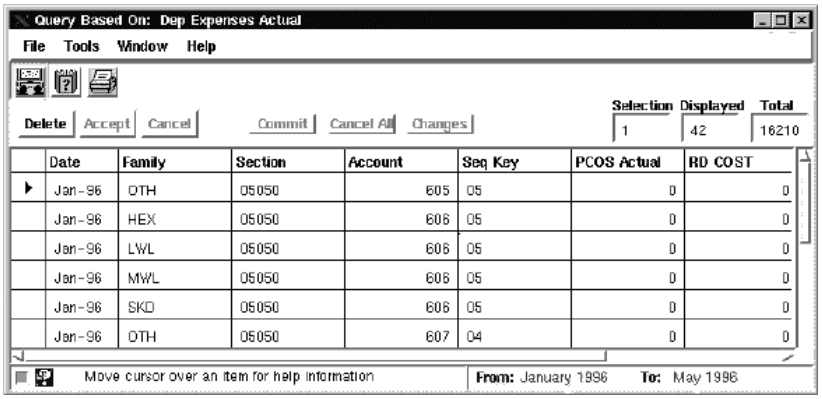
Vehicles by marketing company.



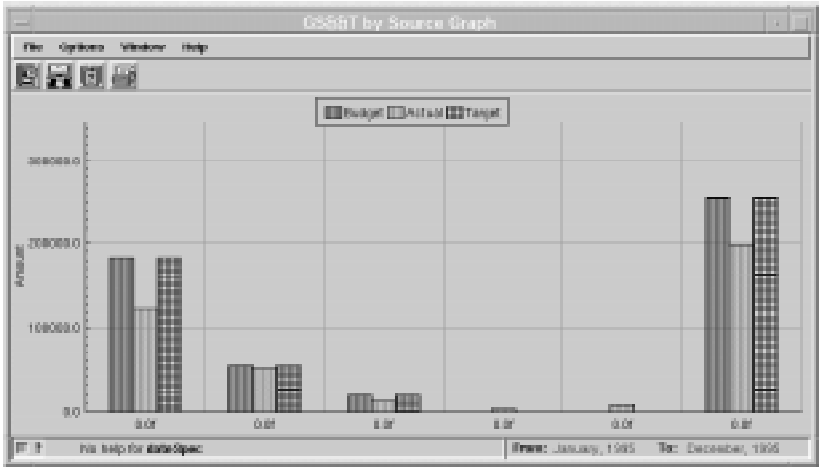
 18

Detailed Transactions

Inspect and edit the individual transactions.



Graphs



Summary of Reports

- Top level (Dupont or P&L)
- Drill down (ReportModel)
- Summary report
- Detailed transactions
- Graphs

Patterns for Developing Frameworks

1) Three Examples
2) White-box Framework
3) Component Library
   – Build applications and add components to library
4) Hot Spots
   – Separate Changeable from Stable Code
   – Design Patterns

Patterns for Developing Frameworks

5) Pluggable Objects
6) Fine-grained Objects
7) Black-box Framework
8) Visual Builder
9) Language Tools

http://st-www.cs.uiuc.edu/users/droberts/evolve.html

Three Examples

- Models for three business units
- Seemed completely different at first.
- Only one was fully implemented

White-box Framework

Five kinds of ApplicationsModel, with lots of subclasses

- DupontModel
- ReportModel
- DetailedModel
- SummaryModel
- GraphModel

User Interface Frameworks

- ***DuPontModel*** - Top Level View
- ***ReportModel*** - Builds a spreadsheet interface using values and GUI descriptions from ***ReportValues***.
- ***SummaryReports*** - Detailed Reports
- ***DetailedWindows*** - Edit and view individual transactions
- ***GraphReports*** - Graph Reports

White-box Framework

- New window = new subclass
- Subclass has methods for
  - reading database
  - computing values
  - stuffing them in GUI
- Initialization registers with dependents

Component Library

- First, just abstract superclasses
- Second, query objects
- Third, GUI objects

Hot Spots

- Find aspects that change, and make them objects
- Often are patterns from *Design Patterns: Elements of Reusable Object-Oriented Software*
- QueryObjects: Interpreter pattern

Interpreter Pattern

- Need to represent SQL to manipulate query:

```
SELECT SUM(sales) FROM sales_and_transfer
WHERE family='MWL' AND date >= '1/1/96'
   AND date < '1/1/97'
```

- Problem: how do you represent a simple language?

Interpreter Pattern

1) make a class hierarchy that represents nodes in abstract syntax tree (SELECT, AND, <, tables, field names)

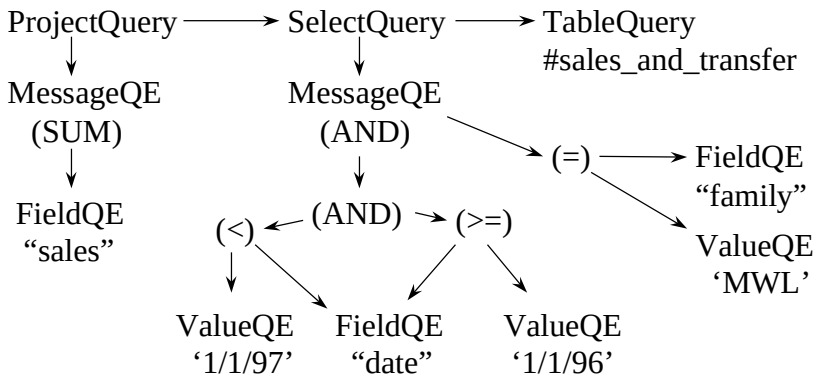
2) define methods to construct and manipulate tree

3) define method to compute value of query (this is the "interpreter")

Instance Hierarchy

SELECT SUM(sales) FROM sales_and_transfer

WHERE family='MWL' AND date >= '1/1/96' AND date < '1/1/97'



QueryObjects

QueryObject
- TableQuery
- JoinQuery
- WrapperQuery
  - RenamingQuery
  - ExpressionQuery
    - SelectQuery
    - ProjectQuery
    - OrderQuery

QueryExpression
- ValueQE
- MessageQE
- FieldQE
  - RenamedFieldQE

QueryObject Protocol

- values - answer collection of tuples
- fieldNames
- join: aQueryObject
- select: aQueryExpression
- project:, renameColumnsTo:, outerJoin:, groupBy:, orderBy:, asDistinct

Creating a QueryObject

```
salesQ := #sales_and_transfer asQuery.
dateQ := salesQ select:
        ((salesQ @@ 'family') = 'MWL') &
        ((salesQ @@ 'date') >= '1/1/96') &
        ((salesQ @@ 'date') < '1/1/97').
dateQ project: (dateQ @@ 'sales') Sum
```


QueryExpression Protocol

+, -, <, =, &, |, Sum, Average, Count, …

Sending one of these messages to a QueryExpression builds a MessageQE with the appropriate operands, and with the message as the operator.

Leading to Black-box

- Component Library
- Hot Spots
- Pluggable Objects
- Fine-grained Objects
- Black-box Frameworks

First Design

- Class hierarchy of ReportModels, ReportModel creates QueryObjects.
- Improvement: separate logic and GUI
- Two hierarchies: ReportModel and ReportValues.
  - Result: twice the classes, some reuse

First Separation

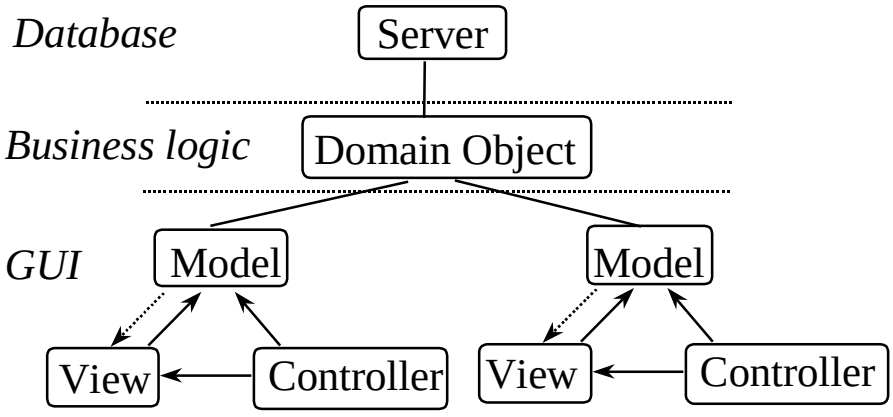
- ReportModel
  - SalesModel
  - InventoryModel
  - ...

Uses QueryObjects

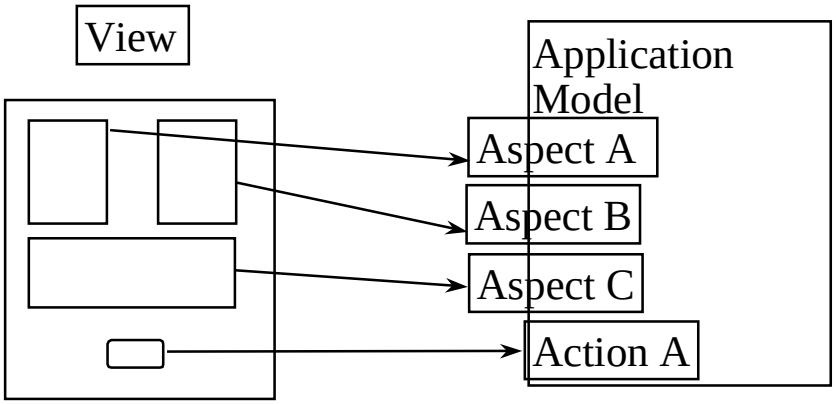
- ReportValues
  - SalesValues
  - InventoryValues
  - ...

Makes QueryObjects

Three-tiered Client-Server



Structure of Application

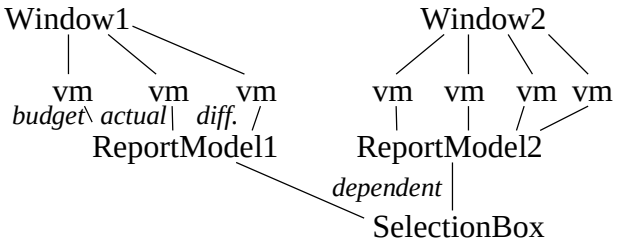


View is a dependent of the aspect.
Aspect is a ValueModel.

ValueModel

- Example of Observer pattern
- View (observer) registers with ValueModel (subject) and is notified when it changes.
- ValueModel protocol
  - value, value:
  - addDependent:, removeDependent
- Observer protocol
  - update:

Using ValueModels



budget value: (query1 values first first).
actual value: (query 2 values first first).
difference value: budget value - actual value

A query returns a collection of tuples.

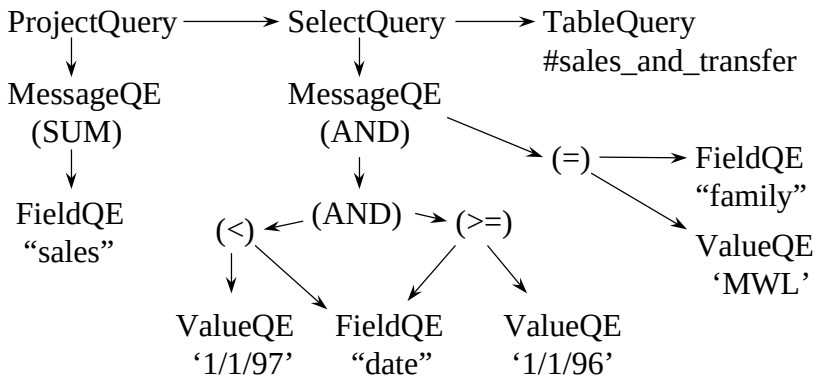
Alternative Solutions

- ReportModel depends on SelectionBox.
  - updates for too many changes
- ReportModel depends on ValueModels from SelectionBox used in QueryObject
  - hard to manage dependencies
- ReportModel depends on QueryObject, QueryObject depends on ValueModels from SelectionBox

Observer and QueryObjects

Let ValueQE refer to a ValueModel.

Let each QueryObject observe its components.



Old way - route change through report

budget value: (query1 values first first).
actual value: (query 2 values first first).
difference value: budget value - actual value

Update

New way - route change directly to ValueModel

budget := QueryHolder on: query1
actual := QueryHolder on: query2
difference := budget - actual

Initialization

Requires:

QueryHolder - adapts QueryObject to ValueModel
ValueModel understands +, -, *, /, etc

QueryHolder

Adaptor pattern - subclass of ValueModel that lets QueryObject act like ValueModel.

instance variables: query, values

```
query: aQuery
   query := aQuery.
   aQuery addDependent: self
```


QueryHolder

```
update
   values := aQuery values
   self changed

value
   ^values first first
```


Arithmetic on ValueModels

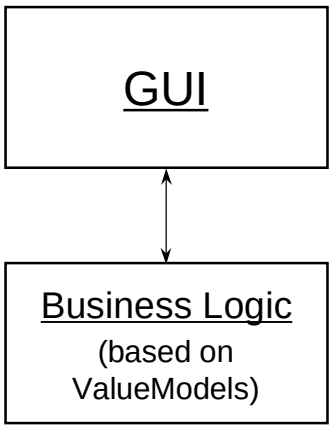
ValueModel implements arithmetic by creating ValueModels that compute function.

```
+ anObject
    ^BlockValue
        on: [:a :b | a value + b value]
        with: (Array with: self with: anObject)
```

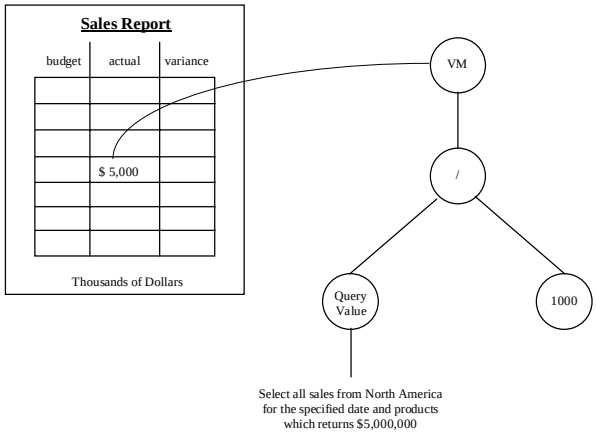

Result of Refactoring

Reuse GUIs, change ValueModels.

Hard part is creating ValueModels and connecting them to GUI.



Typical Values in a Report



Business logic is equations expressed with ValueModel and QueryObjects

- Values = functions of other values
- Values = queries from the database

variable margin = net sales - variable cost

net sales = gross sales - warrantee

gross sales = sum *sales* column from *sales_and_transfer* table

Problems

- How do we go from one report to the next?
- How do we connect report to business model?

- Must define business model flexibly
- Must define GUI flexibly

Specifications

- A ReportSpec
  - has name
  - has parameters
  - has menus, which name other reports
- DetailedReportSpec and SummaryReportSpec are parameterized with QueryObjects.
- GraphReportSpec is parameterized with ValueModels

ReportValues

Many tables.

Many columns

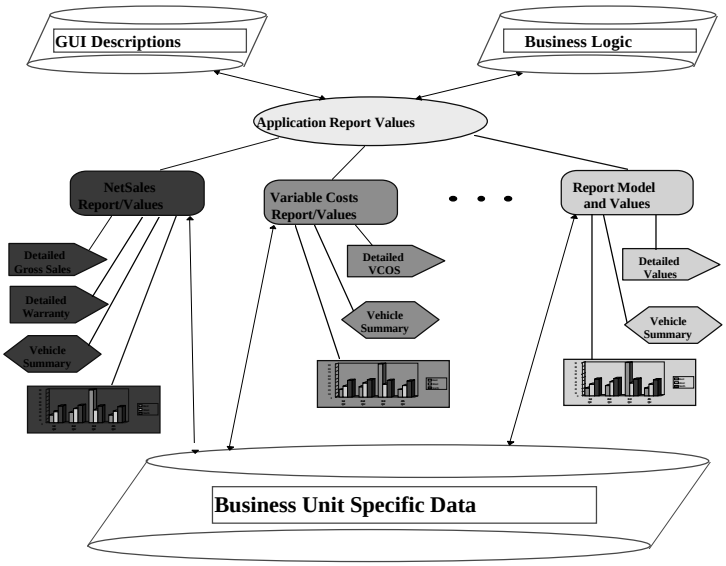
Each table has a sequence of valueModels

Total at end.

Solution

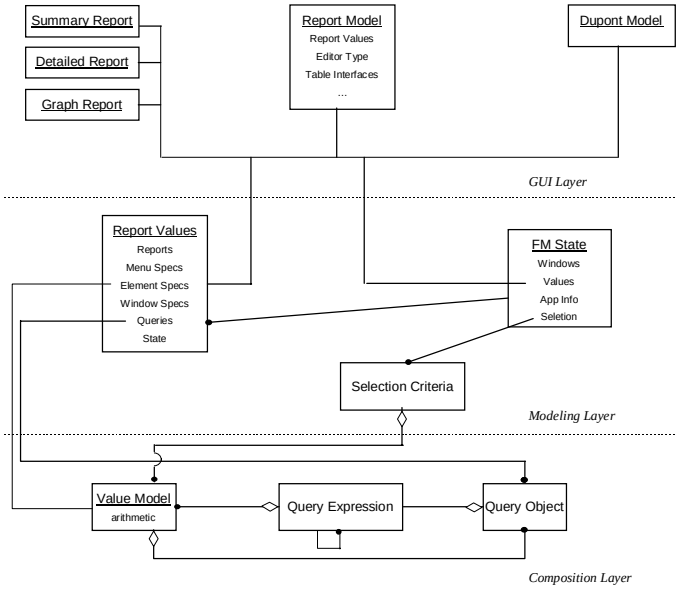
- ReportValues responsible for
  - knowing values
  - knowing how to compute values
  - knowing how to display more detail (drill-down) on values
- Top level starts up ReportValues which starts up next.

Report Values



 57

User View Architecture

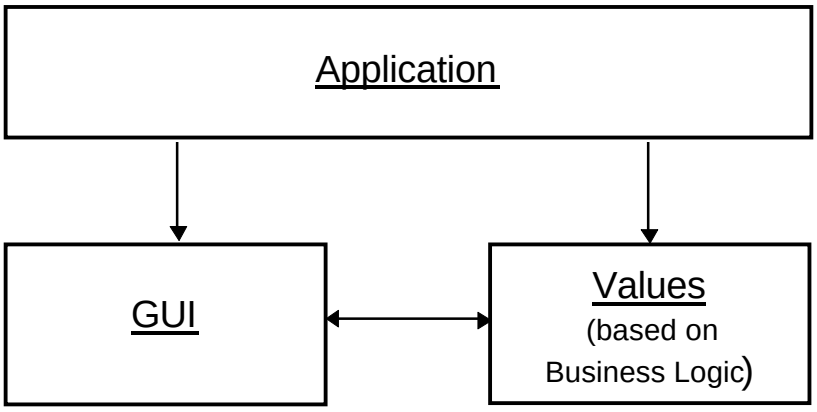


 58

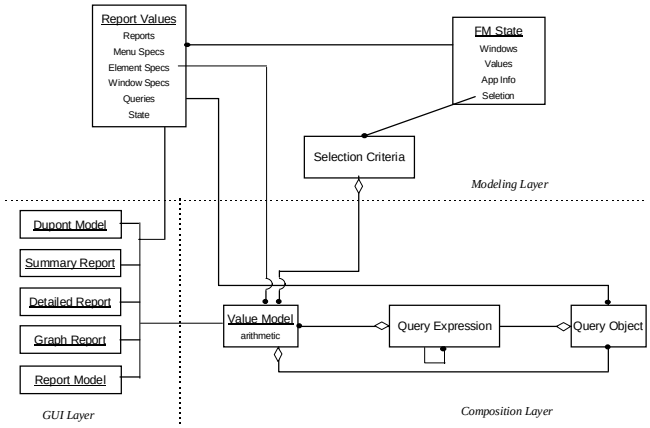
ReportValue protocol

- budget, actual - ValueModels
- openEditor - open “drill down”
  - specify spreadsheets, ValueModels to go in the spreadsheets, menus, reports on menus
- openWith: aSymbol - opens named report

Layered Architecture



Black-box Framework



Visual Builder

- Make a GUI to define Specs.
- This GUI is a language for defining financial models.

Builders

- Equations in a ReportValue
  - expressions
  - queries
- GUIs
  - ReportValue (Drill down)
  - Graphs - specify business logic, labels
  - Detailed - specify query, labels, editing
  - Summaries - specify query, grouping, columns to sum and calculate
- Selection

Language Tools

- Languages need debuggers, profilers, version control, etc.
- So far only built a testing tool.

Summary of Architecture

- Builders
- ReportValues, Selection Criterion, FMState
- GUI frameworks
- ValueModel, QueryObject

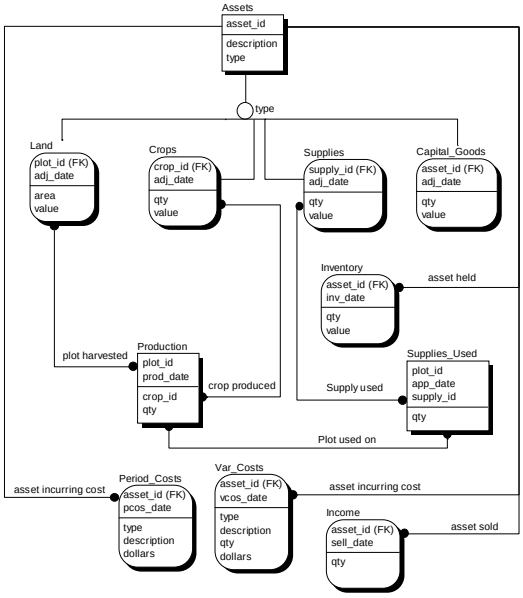
Summary of Architecture

- business model is not object-oriented, just a bunch of equations
- object model is the language for specifying business model, not the business model

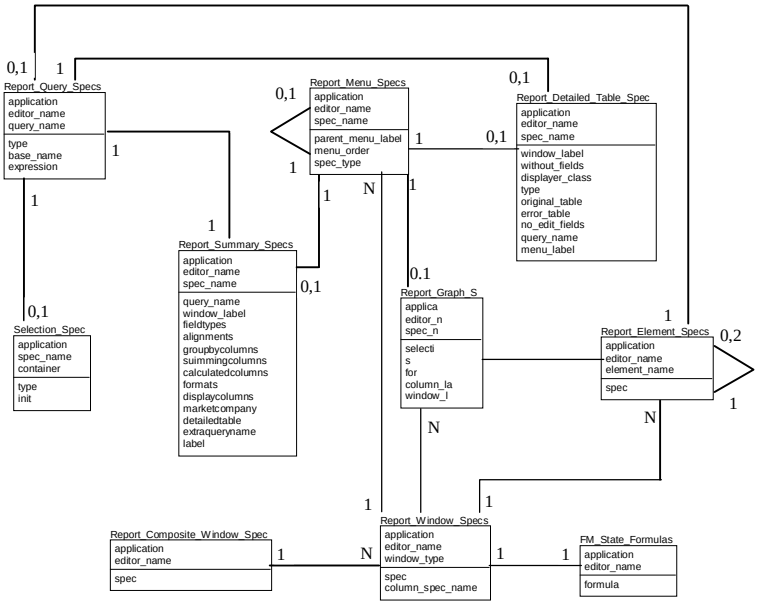
Data Model

- Application Specific
  - holds “real data”
  - changes with every business model
- Generic
  - specifies business logic and GUI
  - never changes

Farm Data Model



Specification Data Model



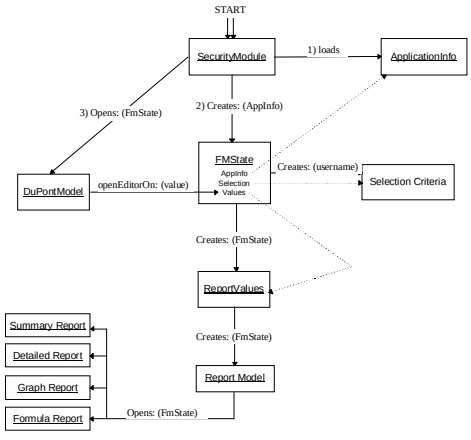
Other Features

- Any window can print itself
- Automated testing support
- Security for editing and/or viewing the data; configured by administrators.

Security Requirements

- Control passwords
- Control login
- Users have roles
- Role can only view a specified list of products.
- Role can only edit a subset of the specified list of products.
- All security features can be controlled by administrators

FM Creational Diagram



How to Develop a New Financial Application

- Analyze business unit
- Build business-unit data model
- Specify GUI and business logic
- Install and Test

Analyze Business Unit

- Questions to ask a new business unit
  - Values to be calculated (netsales, vcos, pcos, ...)
  - User interface
    - top level
    - Drill Downs (summary and detailed)
    - Graphs of values
  - Error-Correction/Analysis modules

Summary

- We have developed a reusable design for financial applications
- Domain specific “Visual-Language”
- Framework emerges by repeatedly refactoring system to eliminate complexity and create flexibility

Related Links

- The following link discusses the details of the framework
  http://www.uiuc.edu/ph/www/j-yoder/financial_framework
- Good Object-Oriented page with framework references
  http://st-www.cs.uiuc.edu/users/johnson/
- The Evolutionary Patterns Paper - PLoP ’96
  http://st-www.cs.uiuc.edu/users/droberts/evolve.html
- The reporting patterns describing query-objects - PLoP ’96
  http:// www.uiuc.edu/ph/www/j-yoder /papers/patterns/Reports/

Related Links

- The security patterns used in this framework - PLoP ’97
  http:// www.uiuc.edu/ph/www/j-yoder /papers/patterns/Security/
- Dmitry Zelenko’s Masters Thesis describing Query Models
  http:// www.uiuc.edu/ph/www/j-yoder /papers/thesis/zelenko.ps/
- Reflective Facilities and Evolutionary approaches - PLoP ’95
  http:// www.uiuc.edu/ph/www/j-yoder /papers/patterns/Evolution/
- Jeff Barcalow’s Masters Thesis describing Scenario Planning
  http:// www.uiuc.edu/ph/www/j-yoder /papers/thesis/barcalow.html