Reports

John Brant **Joseph Yoder**

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{brant, yoder}@cs.uiuc.edu

Abstract

This paper is the beginning of a pattern language for reporting applications. While there are many different aspects of reporting, this paper focuses on the ability to create new reports at runtime. This pattern language consists of patterns that pull data from the database and those that manipulate the data after it has been extracted from the database. This is accomplished by converting both queries and formulas into objects. These object are then assembled into reports through the high-level *Report Objects* pattern.

Introduction

These reporting patterns are based upon research using VisualWorks Smalltalk environment to dynamically generate reports that map to a relational database. This research is part of the *Business Modeling project* funded by Caterpillar. It is a pilot project to demonstrate how an appropriate tool might support financial analysis and business decision making more effectively. This project converts legacy data into a relational database where it is then mapped to and from objects. Reporting the financial aspects of the Business Model required building many reports based upon business logic and SQL queries.

Originally we predefining reports, but this solution was not dynamic enough as the users always came up with some new report we had not anticipated. Although predefined reports can allow for fewer bugs by statically checking things ahead of time, they required programming changes whenever a new report or query was needed. These changes can introduce bugs and take time to get the right people together to understand the needs of the user.

We then researched making the system very flexible and user configurable. The difficulty here is to make a system here that is both easy to understand and modify. If done incorrectly, you've just pushed off the task of programming to the end-users of the system. Furthermore, such as system is more difficult to understand since everything is created at runtime and cannot be statically analyzed.

Our pattern language is broken down into five patterns (see Figure 1). *Report Objects* is the top level pattern for defining reports. It consists of *Query Objects*, *Formula Objects*, and viewing patterns. This paper does not discuss the viewing patterns. *Query Objects* are responsible for pulling data from the database and can be composed into more complex queries by applying the *Composable Query Objects* pattern. *Formula Objects* are responsible for processing data returned from *Query Objects* and are maintained by the *Constraint Observer* pattern.

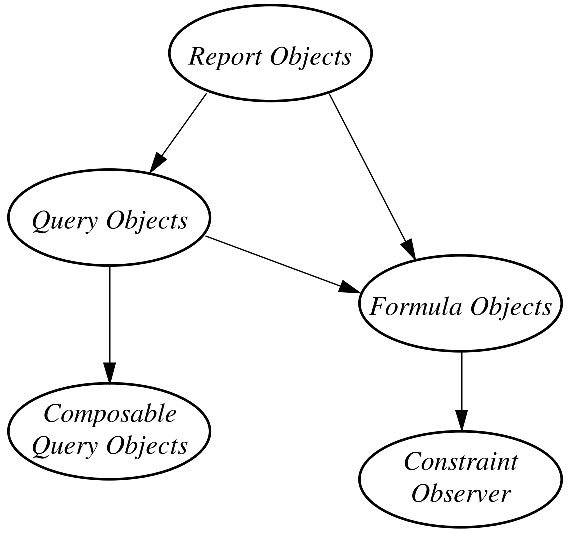

Figure 1: Reporting Patterns Overview

Report Objects

Motivation:

You are creating a reporting application, that given some input from the user, must produce a report. While most reports are known during development, there are many that will need to be customize. For example, consider a financial application. The application would allow users to view the various aspects by selecting different input conditions. You may have designed a report to view sales by marketing areas, but the user might want to also view a report broken down by model. If your reporting framework is not configurable, then you will be required to program the desired changes. Whereas with a configurable application, the user could define these reports at runtime.

Problem:

You need to create a configurable reporting application.

Forces:

Most reports can be predefined with SQL code along with standard views, but these wouldn't contain all of the possibilities that the user might want. Although predefined reports are easier to understand, new reports will require a programmer to modify the system and such modifications are prone to create bugs.

Alternatively, you could make everything configurable so that users could create their own reports such as those seen in many report-writers. But giving the user the ability to define reports on-the-fly can be very complex since you must present the options in an easy to understand format that is less complex than programming. Otherwise you have just pushed the task of programming the system onto the users.

Solution:

Define objects representing the report. These objects can be separated into processing the data and viewing the data. A report is then made by hooking up the processing objects to the viewing objects. In this paper we only discuss the processing objects.

These processing objects can be further split into objects that fetch data from the database and objects that manipulate the data returned. Thus, you can build objects that represent the queries and objects that represent formulas needed in the reports. The *Query Objects* pattern allows you to select what data is extracted from the database, while the *Formula Objects* pattern allows you to operate on values returned by *Query Objects.*

When reporting, you need to define queries and you need to apply functions on the results from queries. As noted in the forces, the solution can range from pre-defining the SQL queries to providing the user with a report-writer to dynamically generate any type of report. This pattern is more of a middle of the road solution which allows you to make objects that represent queries needed for the reports along with objects for constructing complex values by composing these results. This pattern provides the user with something that *Works Out of the Box* [Foote & Yoder 96] while still allowing for customized reports.

Consequences:

This pattern gives you more flexibility in creating reports. Since the data processing is an object, it can be manipulated at runtime. Unlike other implementations that define the processing in methods, this technique gives you more flexibility since objects are more easily manipulated than methods [Foote & Yoder 95].

Although this pattern gives you added flexibility when constructing the reports, it also makes the data processing harder to understand. If the queries were built into methods then it could be statically analyzed, but since they are built at runtime, we must reason about the configurations of the objects at runtime. Determining such runtime properties can be difficult if not impossible.

Related Patterns:

- The *Query Objects* pattern helps build objects that talk to relational databases.
- The *Formula Objects* pattern constructs objects that represent formulas.

Query Objects

Motivation:

You are defining reports for your users. Each report queries the database for the information and displays it to the user. While most queries are known when you are developing the software, there are some that the user will want to create in some user-friendly environment such as a point-and-click interface. For example, you might have a sales query that returns the sales for a specific time interval, but you might want to modify the query to display only Asian sales.

Problem:

You need to create new queries for reports at runtime.

Forces:

If all reports are known in advance, the solution is to create the queries when you build the system. Predefined queries are easy to understand and predict since you just need to analyze the source code. However, they fall short when trying to develop dynamic mappings to the data. Furthermore, extending these queries require program modifications that cannot be performed by the users.

Another option is to create queries at runtime by providing the user with a interface for creating them. This option would require the users to know something about your database in order to build the query. Although users may know what data is stored and what it is used for, they may not know how such information is stored in the database (e.g., they may not grasp the concepts of joining two tables).

Solution:

Create objects that represent queries. Define the operations that will be used to manipulate the queries. For a relational database, you might have operations for selecting which rows should be returned, projecting the fields to be returned, etc. Next, define a method that will return the results of the query. For queries that need to dynamically add new conditions, you can use the *Composable Query Objects* pattern.

For the query operations that take expressions such as selections and projections, you can use the *Formula Objects* pattern to create these expressions. Whenever the expression's value changes, you can update the query which will update the report.

This basic intent of this pattern is to convert methods that contain your queries into objects that are easier to manipulate. This is similar to the *Strategy* pattern, but the main purpose of this pattern is to make objects that are easily manipulated whereas the *Strategy* pattern is more concerned with making them interchangeable.

Example:

For the Asian sales example, you would define an object that represented the sales query which is just a mapping to retrieve the sales records. You would then add basic manipulation operations such as project: that picks which fields are returned; and select: that selects which records are returned. In addition to these operations, you need an operation to return the values from the query such as a value method. This method converts the query into SQL code that is executed by the database and finally returns the results. Using such a protocol, you could modify the sales query object to extract Asian sales simply by evaluating a statement like.

```
salesQuery select: salesQuery continent = ‘Asia’
```

This statement modifies the select where clause so that only records that have the continent field equal to Asia are returned.

Consequences:

The *Query Object* pattern makes it possible to manipulate queries at runtime. Since the queries are objects, they can be changed to add or remove conditions. This gives us the needed flexibility to dynamically create or modifyqueries for unknown reports.

One disadvantage with using this pattern is that dynamic queries can be slower than pre-computed queries. The pre-computed queries can be optimized for the database, whereas the dynamic queries might be translated to a slower query. Furthermore, static queries can be pre-computed in a database views. If this is a major problem, then you might need to write an optimizing translator that will do a better job translating the query objects into database statements.

Related Patterns:

- *Formula Objects* can create expressions for the select conditions.
- *Composable Query Object* pattern can build new queries out of existing queries.
- The *Strategy* pattern [GHJV 95] is similar since methods are being encapsulated into objects.
- The *Crossing Chasms* pattern language [Brown & Whitenack 95] can be used to map between objects and relational databases.

Composable Query Objects

Motivation:

You have decided to use the *Query Objects* pattern to build dynamic queries, but you have many reports with similar queries. Rather than having to define a query for each report, you would like to define one query for the common part and create new queries based on the common query. This would allow you to easily create new queries is some minimalistic fashion. Not only would this help during the creation of queries, it would also help during debugging since you only need to fix the common part once and all of the dependent queries are automatically fixed. Furthermore, it allows for the dependent queries to dynamically update whenever the common query modifies its selection criteria. For example, given the sales query from the previous section, you might want to look at both Asian sales and North American sales during a time interval. Whenever you update the time interval, both reports should also be updated. The only differences between these two reports would be the sales records that are selected.

Problem:

You want to create different queries with simple modifications to common base queries.

Forces:

You could use the previous patterns to build a query object for each sales query, but this would be hard to keep up to date with other selections. From the example above, imagine that the user decides to continue to look and compare the North American sales with the Asian sales, but decided to view the fourth quarter instead of the third quarter. This would require triggers or call-backs to make sure that all open views were updated appropriately.

Another solutions would be to use wrappers to “wrap” new constraints on existing queries. Although these are harder to implement, they do provide a means for keeping values consistent with changing constraints. The wrappers also provide a nice way to build a new query from existing queries. If any of query needs to be modified to fix a bug then its wrappers will also be updated without needing to directly modify them.

Solution:

Build queries out of composable parts. You can define one class that will represent tables in the database and for each query operation define a class that performs the operation. Additionally, define methods for each operation that will create the appropriate object for that operation on the query. For a

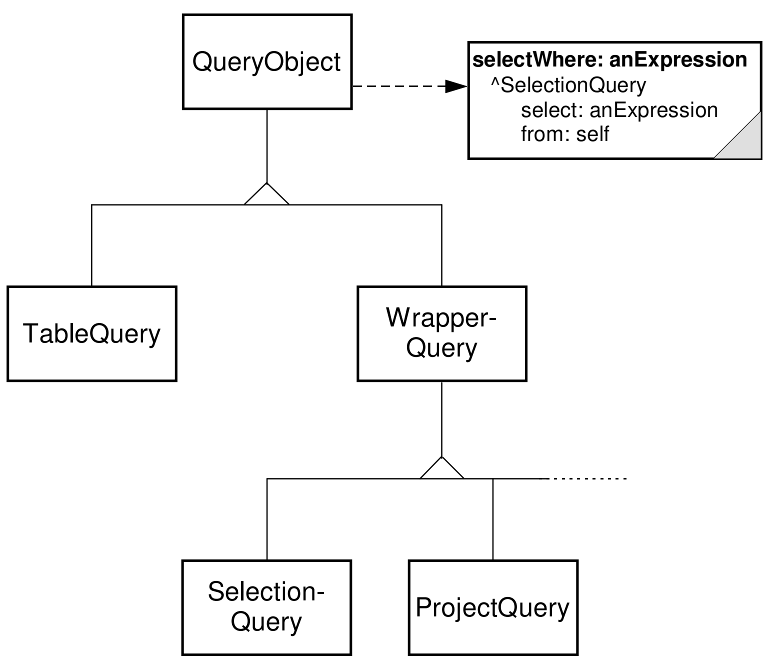

Figure 2: Query Objects Hierarchy

relational database, we would define a TableQuery class to represent tables, and define classes such as SelectionQuery to represent the selection operations, JoinQuery for join operations, etc. Since all of the operations are performed on some query, we can create the abstract superclass, WrapperQuery, to handle wrapping these queries with the operations. Finally, you might define methods that create these operations. For example we can define a selectWhere: method that will create a SelectionQuery on the query (see Figure 2).

To convert a query object into database statements, we use the *Interpreter* pattern from [GHJV 95]. Each type of query object will be a node for the interpreter. The table query objects are the terminal nodes for the interpreter, and the operation nodes are the non-terminals since they wrap other query objects. Instead of making an interpreter to create the complete database statements, you can make separate interpreters that will create the statement parts. For example, you would have an interpreter for the where clause, another for the projected fields, etc. The primary difference between this use and the *Interpreter* pattern is that the query objects can generate their output without any additional input.

Finally, each of the WrapperQueries should be updated whenever its wrapped query changes. This can be accomplished by using the *Observer* pattern from [GHJV 95]. The WrapperQueries will be the observers and their wrapped queries will be the subjects. Whenever a wrapped query changes the WrapperQuery will also change, allowing the reports to update.

Example:

For the sales by continent example, we would have a base query object that retrieves all the sales records from the database for a particular time interval. To create a particular report for a continent we would just need to create a new query to select those records for that continent. For example, we could create the Asian's sales query with

salesQuery selectWhere: salesQuery continent = 'Asia'

Whenever the salesQuery's date interval changes it will notify its dependents which are the continent sales queries. Both of the continent sales query objects will change and cause their reports to update. Figure 3 displays the base salesQuery object. Both the Asian sales query and the North American sales queries are observers to the salesQuery.

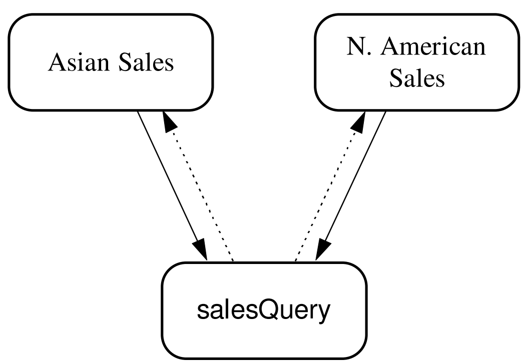

Figure 3: Sales Query Objects

Consequences:

This pattern makes programming easier for you to do "exploratory programming" since you do not necessarily need to understanding all facets of the database when creating a query. For example, you can extend your sales query without understanding all of the details of which tables were used, what are the date fields, etc. You would only need to know that there is a continent field.

While this pattern allows you to dynamically create similar queries and automatically maintain them, it also results in objects that are harder to implement. Since the knowledge of a query is spread out into several objects, it is harder to create the database statements for the query. We must traverse all of the objects to create the database statements, whereas if all of the details of the query were in one object, then we could directly create the database statements.

Another disadvantage of this pattern is that it results in many small objects wrapping other query objects. These wrappers make it harder to debug the system, since it is more difficult to traverse the wrappers to see what the query is performing.

Related Patterns:

- This pattern uses the *Decorator* and *Composite* patterns from [GHJV 95] as it wraps new responsibilities and composes new objects.

Formula Objects

Motivation:

You are creating a reporting application. Some values are directly available, but there are many that are calculated. For example, PROFIT is a calculated value that is based on INCOME and EXPENSES. While some of these formulas might be known when the program is created, we want to allow the user to create or modify formulas at runtime.

Problem:

You need formulas to change at runtime and have them automatically maintained.

Forces:

One option would be to define methods for all of your formulas, but this would require programmer intervention whenever adding or changing formulas. Not only might this introduce new bugs, it also will take some time before its completed because of paperwork, getting the right people together, etc.

Another option would be to have the user create the formulas at runtime, but many formulas do not change. Users shouldn't have to enter these formulas; they should already be defined. Furthermore, users are more likely to have precedence errors in entering their formulas and are less likely to fully test them.

Solution:

Define an object that represents the results of a computed formula. For the result object to be used in other formulas, the result object should be in the same hierarchy as the objects used to create it. Next, define the basic formula operations to more easily create the formula objects. For example, you might define methods such as + and - if you are performing basic arithmetic operations. To make sure that the result object is consistent with its inputs we can use the *Constraint Observer* pattern.

Example:

As an example, consider VisualWorks, which stores many of its values in ValueModels so that they can easily be plugged into its UI framework. To create a result object we could subclass ValueModel to create a ResultHolder[1] that would hold the result of the formula (see Figure 4). We could then define basic operation of +, -, *, and / on ValueModel to return a new ResultHolder whenever they were executed. Now if both INCOME and EXPENSES are ValueModels, then we can create a ValueModel that represents PROFIT simply by executing "INCOME - EXPENSES".

[1] VisualWorks already has a BlockValue class that incorporates both the *Formula Object* pattern and the *Constraint Observer* pattern.

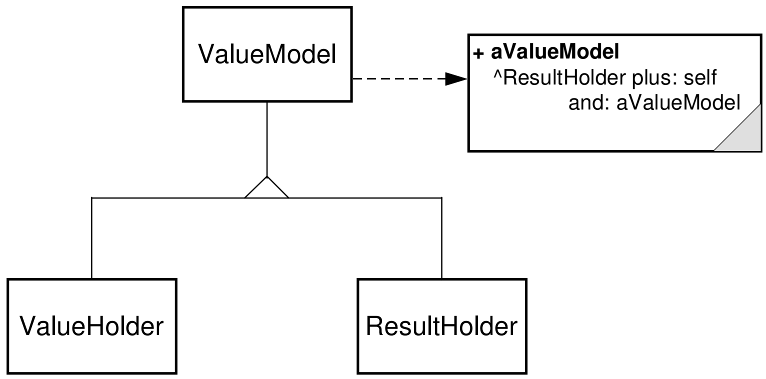

Figure 4: Result Holder Example

Consequences:

This pattern allows people to use formulas without knowing how they are maintained. Only the person implementing the pattern needs to understand that. This reduces much of the complexities that would have been required when creating new formulas. Instead the people creating the formulas can concentrate on making them correct, and not on how to maintain them.

One disadvantage of using *Formula Objects* is that it is easy to create too many objects since an object is created for each sub-expression in the formula. For example, a simple formula for computing margin percentage such as "(INCOME - EXPENSES) / INCOME" would create two objects; one that represents "INCOME - EXPENSES" and another for the margin percentage. The extra objects aren't generally a problem in practice, but could cause problems in applications with many complex formulas. In these applications it may be necessary to "optimize" the formula by making the computed result responsible for the complete formula instead of just expressions in the formula.

Related Patterns:

- *Constraint Observer* pattern can be used to keep the result object consistent with its inputs.
- The *Interpreter* pattern from [GHJV 95] can be used to create result objects from user entered formulas.
- *Composable Query Object* pattern is similar. The primary difference is that *Composable Query Objects* operate on queries while *Formula Objects* operate on the results of queries.
- The *WyCash Report Writer* [Cunningham 91] has a few formulas which can add additional columns to a report. These are similar to instances of *Formula Objects* in that they define objects for "formula columns" that are based on other columns and/or rows. Using such a system, they were able to build their reports without having to define new types of formulas.

Constraint Observer

Motivation:

You have applied the *Formula Object* pattern to create formulas dynamically, but now you need to update these values whenever one of their input values change. For example, you may have created a formula "INCOME - EXPENSES" to calculate PROFIT. Whenever INCOME or EXPENSES change, PROFIT should change to maintain the consistency of the formula.

Problem:

You need to automatically update calculated values of formulas whenever an input value changes.

Forces:

A simple solution would be to create a method for each formula that updates the new values whenever an input changed. This would require a new method for every possible formula, which may not be known in advance.

Constraint systems are another solution that give the needed flexibility to automatically maintain the formulas. However, they can limit how programs are written and can be very difficult to understand, especially when trying to follow the control flow while maintaining the formulas.

Solution:

Create a constraint object that is responsible for the calculation. Use the *Observer* pattern [GHJV 95] to update the constraint whenever an input to the formula changes (i.e., the inputs for the formula will be the subjects and the constraint will be the observer). Whenever the constraint is updated, it evaluates the formula and assigns the result.

Example:

For the PROFIT example, you would have the constraint depend on both the INCOME and the EXPENSES objects (see Figure 5). Whenever the "-" constraint receives the update message from INCOME or EXPENSES, it evaluates and assigns the result into the PROFIT object.

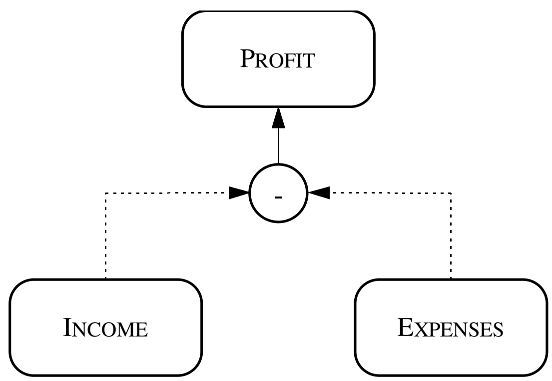

Figure 5: Profit/Loss Example

Consequences:

The main advantage of this pattern is its simplicity. If you were to use another constraint system, then you would be required to maintain some complicated data structures so that the constraints would be optimally solved. And, if you were to use methods, then you would also need some other data structure that listed the methods that need to be executed when an event occurred.

The main disadvantage of this pattern is that it can led to sub-optimal performance. For example, the formula above will change PROFIT whenever INCOME or EXPENSES change. But if every time one value changes the other also changes, then we will be updating PROFIT twice, when one update is all that is required. Although for simple arithmetic formulas, these extra calculations don't have a big impact, for formulas that involve queries these extra calculations might be unacceptable. In these cases a simple flag might be used to delay updating until all objects have changed. If the flag still does not work, you probably need a constraint system that can optimally solve them.

The *Constraint Observer* is a simple pattern for implementing a constraint system, but it will not work for many general constraint problems. For example, constraints that are circular would cause this system to get into an infinite cycle of update messages. This is not a problem though, since the constraints were created with formulas. Thus, they guarantee no circularities.

The constraint object can either be a part of the result or a separate object. Representing them as separate objects is more flexible since the constraint and the result can be varied independent of each other. Furthermore, each result object could have multiple constraints associated with it. But if the constraints were created by formulas, then we don’t need the extra flexibility since every result object would have only one constraint object.

VisualWorks has a special class that implements the *Constraint Observer* pattern for ValueModels. The BlockValue object has both the result and the constraint. The result is stored in the value instance variable, and the action performed by the constraint is stored as a BlockClosure in the block instance variable.

Related Patterns:

- This pattern is similar to the *Constraints* pattern described in [Johnson 92]. The *Constraint* pattern was used by HotDraw to update figure locations whenever a dependent figure changed its location. HotDraw only defined simple equality constraints, but it could simulate more complex formulas by using Locator objects.

Summary

With changing business requirements users need flexible programs that quickly adapt to their business needs. No longer can users wait for programming changes that take weeks to finish. Therefore, it is important to provide the flexibility necessary for the program to adapt to the users needs. The patterns discussed in this paper help provide such support. They accomplish this by transforming queries and formulas into objects where they can be more easily manipulated.

We are grateful to the members of the Professor Johnson's Patterns seminar: Jeff Barcalow, Ian Chai, Brian Foote, Charles Herring, Ralph Johnson, Mark Kendrat, and Susanne Schacht, who reviewed an earlier draft and provided valuable feedback.

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*.

[Foote & Yoder 96] Brian Foote and Joseph Yoder. “Attracting Reuse,” To appear in *PLoP’96 Proceedings*.

[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.