Patterns For Developing Successful Object-Oriented Frameworks

Joseph W. Yoder

August 27, 1997

1 Overview

The work described here extends last years OOPSLA framework workshop paper [Yoder 1996] describing a "Business Modeling" application framework that was developed and deployed at Caterpillar, Inc. Caterpillar, Inc. joined the National Center for Supercomputing Applications at The University of Illinois as an Industrial Partner in December 1989 to support the educational function of the University and to use the University environment to research new and interesting technologies. During the partnership Caterpillar has initiated various projects, including the evaluation of supercomputers for analysis and the investigation of virtual reality as a design tool.

As described in last years workshop paper, the Business Modeling project aims to provide managers with a tool for making decisions about such aspects of the business as: financial decision making, market speculation, exchange rates prediction, engineering process modeling, and manufacturing methodologies.

This tool must be flexible, dynamic, and be able to evolve along with business needs. Therefore, it must be constructed in such a way so as to facilitate change. It must also be able to coexist and dynamically cope with a variety of other applications, systems, and services [Foote & Yoder 1996].

One of the problems that we had to deal with was that there were many different business units within Caterpillar in which the system needed to be deployed. Each of these business units had a different business model and different requirements for viewing and searching their data.

The system that was developed and deployed was a Financial Modeling framework [Yoder 1997] that generated reports, answered questions such as why costs were too high, allowed for error corrections, and included a

security model. The framework allowed for quickly creating applications that examine financial data stored in a database. The applications that are generated produce profit and loss statements, balance sheets, detailed analysis of departments, sales regions and business lines, with the ability to drill down to individual transactions.

The framework was developed using Smalltalk but it can create a complete system for examing financial data without any Smalltalk programming. This is primarily because the business model is stored in a database. The framework interprets the descriptions of the business model along with GUI descriptions for the reports and dynamically builds an application based upon this information.

The primary principle discussed in this paper 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 a higher-level language you can more quickly program, within a limited domain, in fewer lines of code. This work provides a visual-object-oriented 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.

2 Patterns for Developing Frameworks

The framework development process follows the model presented in the Evolving Frameworks paper [Roberts & Johnson 1997]. We started out with three business units that appeared to be completely different. After working on these for a few months, we saw many similarities. This lead us to develop supporting white-box frameworks for reusing code within and across these applications. It required the developer to write many subclasses for specifying the differences between business units, however their were some key functional areas that wer able to be abstracted out and reused. We only fully implemented one system for Caterpillar.

The user-interface frameworks primarily consisted of summary reports, detailed transaction reports, and graph reports. Some basic components were built with some abstract superclasses. Every time you needed a new report for a business unit, you wrote a new subclass for the GUI that also implemented much of the business logic. These white-box frameworks followed what you might think of business objects as our framework had classes for income, cost, inventories and the like. All of the similarities were ab-

stracted out into a common ReportModel superclass which was a subclass of the ApplicationModel class in VisualWorks.

Early in the development stage, we noticed that the business logic needed a lot of queries to get values from a relational. These queries often had to trigger updates of other queries or values whenever some selection criteria changed. For example, maybe the user was looking at the income for product "X" and wanted to change the view to show the income for product "Y". In order to get all of the views to update correctly, every view and every query that had values coming from the database representing income for product "X" had to be updated to represent income for product "Y". We used Smalltalk's dependency mechanism to handle these updates. Needless to say, the maintenance of these updates became a nightmare. Every time we wanted to change queries or write a query based upon the values of other queries, we had to either change or write the appropriate update method.

These were the early "hot-spots" for us. This lead us to develop a Query-Object class hierarchy that allowed to quickly build queries for getting values from the database and also allowed us to use the ValueModel provided by Smalltalk for automatically updating values. These because components for building the business logic and plugging them into views for being displayed.

Queries could be built that were dependent upon a ValueModel. These QueryObjects were automatically updated whenever the ValueModel that they were dependent upon changed. You could also build QueryObjects based upon other queries that automatically updated themselves whenever any of the queries that they were dependent upon changed. This reduced our maintenance nightmare by a large factor as we no longer had to worry about writing or updating dependency code. [Brant & Yoder 1996] talks about these issues in more detail in their Reports patterns.

We were also able to build a black-box framework for graphs and detailed reports early on as each of them only took a collection of values and displayed a report for them. Thus, we could parameterize these reports to "openOn:" a query or collection of values that then dynamically built what was needed for viewing the data. This was done by reusing what was provided by the VisualWorks graphing and database frameworks.

Most reports still required customization and subclassing. A major improvement happened when we separated the business logic from the application models. We ended up with a logically three-tiered system where the domain objects knew how to get all of their values from the database and the GUIs knew how to display the values and update them. We had two class hierarchies that needed to be maintained, one for the GUIs and one

for the domain objects. Whenever a new report needed to be created, the developer would subclass one of the GUI objects and overwrite what was being displayed and also wrote a subclass of a domain object that described the business logic for what was being displayed.

We then noticed that all of the GUIs were very similar and were really just the same view on different data elements. This lead us to build a component library for the GUIs which were primarily pluggable objects. This became the beginnings of a black-box framework as the GUIs were built from descriptions that were decoded into how to build the GUI and what domain elements should be plugged into the view.

As we saw more and more similarities, we continued to either extend our components or make new components. We developed more pluggable objects and they became more and more fine-grained. This was an iterative process where we continued to rethink our design and refactor our code.

Ultimately we saw where we could also describe the business logic in a database and build the domain objects on the fly. This allowed for the business logic to dynamically change without rewriting code. We ended up with a restrictive domain-specific language. It allowed the developer of the system to program at a high-level within the limits of the domain, thus quickly producing an application for analyzing the finances of a business. The set of frameworks had evolved to be completely black-box. We provide ways to describe the GUIs and the business logic and store the values in a database. The values are read in on demand and through the use of the Interpreter and Builder patterns [Gamma et. al 1995] dynamically build an application as needed.

The next step was to create visual builders for describing the GUIs and the business logic. We took a top-down approach for a proof of concept. This approach took the way you viewed reports from the top-level down to the summary, detailed and graph reports and allowed you to step through and describe the reports. We also allowed for the business logic to be described in this manner as it starts from the high level reports and you let describe the values that get plugged into the GUIs down to the queries that ultimately get mapped into SQL code for getting the values from a relational database.

We ended up with two databases. One database is the database that models the real business data. It has the values of the actual transaction of your business that may include for example costs, income, and so forth. This database can evolve with your business needs which might not only take new types of data, but might also have a new structure defining the database. Even though our frameworks maps into a relational database,

since QueryObjects are a layer above the real business database, our framework can easily be changed to map into any type of database for getting your business values.

The second database which holds the values describing the GUIs and the business logic (the meta-data). The structure of this database should never change unless you change the framework. However the values that are kept in this database does change with your business needs. Whenever a new report or a new way to slice your data is needed, the modeler simply describes the new look of the new report along with the business logic needed for presenting the values in the report. This data is used for dynamically building the view onto your business data.

The surprising part of this was that instead of ending up with normal business objects such as Income and Cost, our framework consisted of Formula, Query, and GUI objects that really just supported our domain-specific language.

Ten years ago, people would have thought that our first version of the system separated the business logic from the GUI. After all, we didn't really write any view classes. We only wrote application models. However, when you have a well-developed GUI framework, the GUI work is how you build application models. Now you want to separate application models from the "real" model. And if you want it to be dynamic, thus adaptable to the changing needs of your business, it is important to incorporate reflective techniques into your framework.

3 Conclusion

This workshop paper has briefly described the evolutionary process that was undertaken during the development of the Financial Modeling Framework. Framework development was an interactive process for us as we first had to develop some working applications in order to see the common abstractions. Early on as we were learning the domain we developed white-box frameworks. We were then able to generalize from the concrete examples. As we iterated through the process, the abstractions revealed ways to generalize into components. These components became more black-box and fine-grained which required much less programming.

Ultimately we ended up with a domain-specific language for building these frameworks without writing any Smalltalk code. This then lead to the development of a visual-language for building your application into some-

thing that the domain-experts will be familiar with. The only thing left is to extend the language with debugging tools, profilers, version control, and optimizers.

This work was supported in part by Caterpillar/NCSA at the University of Illinois. The author is grateful for the support provided by the above mentioned entities but in no way holds them responsible for any possible inaccuracies or claims.

4 References

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

[Brant & Yoder 1996] John Brant and Joseph Yoder; Reports Collected papers from the PLoP '96 and EuroPLoP '96 Conference, Technical Report #wucs-97-07, Dept. of Computer Science, Washington University Department of Computer Science, February 1997, URL:
http://www.cs.wustl.edu/~schmidt/PLoP-96/yoder.ps.gz.

[Foote & Yoder 1996] Brian Foote and Joseph Yoder; Architecture, Evolution, and Metamorphosis, Second Conference on Pattern Languages of Programs (PLoP '95) Monticello, Illinois, September 1995 Pattern Languages of Program Design 2 edited by John Vlissides, James O. Coplein, and Norman L. Kerth. Addison-Wesley, 1996.

[Gamma et. al 1995] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[Roberts & Johnson 1997] Don Roberts and Ralph Johnson; Evolving Frameworks: A Pattern-Language for Developing Object-Oriented Frameworks, Third Conference on Pattern Languages of Programs (PLoP '96) Monticello, Illinois, September 1995 Pattern Languages of Program Design 3 edited by John Vlissides, James O. Coplein, and Norman L. Kerth. Addison-Wesley, 1997.

[Yoder 1996] Object-Oriented FrameWorks Tutorial for OOPSLA '96, URL:
http://www-cat.ncsa.uiuc.edu/
~yoder/papers/oopsla96/ooframewks/ooframewks.html.

[Yoder 1997] A FrameWork for Building Financial Models, URL:
http://www-cat.ncsa.uiuc.edu/~yoder/financial_framework/.

5 Biography

Joseph W. Yoder is an object-oriented consultant and is working on his Ph.D. with Professor Ralph Johnson. His focus is currently on object-oriented technology and how it changes the way software is developed. In particular, he is interested in how to use and develop frameworks, which he believes is a key way of reusing designs and code. He is studying and writing design patterns for developing reusable software and domain specific languages.

Joe has worked on the architecture, design, and implementation of various software projects dating back to 1985. These projects have incorporated many technologies and range from stand-alone to client-server applications, multi-tiered, databases, object-oriented, frameworks, human-computer interaction, collaborative environments, and domain-specific visual-languages.

These applications have been in many domains, including Medical Information Systems, Medical Examination Systems, Statistical Analysis, Scenario Planning, Client-Server Relational Database System for keeping track of shared specifications in a multi-user environment, Telecommunications Billing System, and Business & Medical Decision Making.

Joe believes that in order to see abstractions, you must iterate through the development process. Architects and designers are usually not the domain experts so it becomes important to provide quick feedback loops for learning the domain from the experts. Once the basics of a domain have been learned, it is then possible to see the abstractions and build frameworks that consist of a high-level domain-specific language.

For the last two years Joe has been investigating "visual languages for business modeling". He is designing them, using them, and implementing them. His current project has been on using frameworks to develop and implement visual languages for use with "business modeling". This project is aimed at providing support for decision making during the business process.

Joe believes that Frameworks are both a way of coming up with visual languages and a way of implementing them, because if you focus on building something in an Object-Oriented language, then building a framework for it, then making the framework composable, and then making a direct manipulation tool for composing applications using a framework, you will automatically discover a visual language.

Joe's Research Interests Include: Computational Theory, Learning Theory, Human Computer Interaction, Software Engineering, Computer-supported Cooperative Work, Visual Programming (including grammars and parsing),

Expert Systems, Intelligent User Interfaces (providing intelligent automatic semantic feedback), Object-Oriented Programming and Databases, Design of Reusable Software-specifically with the use of Frameworks, Domain Analysis and Engineering, and Pattern Languages of Programming.