

Refinements and Separation of Concerns

Don Batory

Dept. Computer Sciences
University of Texas at Austin
Austin, Texas 78712
1-512-471-9713
batory@cs.utexas.edu

ABSTRACT

Today's notions of encapsulation are very restricted — a module or component contains only source code. What we really need is for modules or component to encapsulate not only source code that will be installed when the component is used, but also encapsulate corresponding changes to documentation, formal properties, and performance properties — i.e., changes to the central concerns of software development. The general abstraction that encompasses this broad notion of encapsulation is called a “refinement”.

Keywords

Refinements, components.

1 INTRODUCTION

The history of planetary astronomy offers an interesting lesson in the progression scientific understanding. Planetary motion was not well-understood in the 1500s. In particular, retrograde motion — where planet traversals across the celestial sphere have loops — was very difficult to explain and predict. A progression of complex models of spheres inside spheres were proposed, but none really worked that well. The core problem, as we eventually learned, was that the Sun, not the Earth, was at the center of the solar system. The mathematics of a heliocentric theory are both simple and elegant. The mathematics of a geocentric theory (which is obtained by coordinate transformations of heliocentric mathematics) shows the difficulty of developing a correct model of planetary motion from purely a geocentric perspective: totally unnecessary and complicating mathematics are introduced simply because the “wrong” perspective (i.e., coordinate system) was chosen. A generic lesson to be learned is that selecting the “right” perspective not only simplifies core problems, it provides a gateway to the understanding of much more complex phenomena.

Software design has strong similarities to theories of planetary motion. As a general rule, we don't know how to design software very well. Software design remains a very difficult problem. It is a massive exercise of information organization: how can one coherently and cleanly organize enormously complex and diverse details so that resulting

computations can be performed correctly and efficiently? What a software architect does is to impose his “perspective” on this mass of details to give an application its “organization” or “modularization”. Often, the contents of a module is chosen by convenience or that it adheres to some conception of the application based on the architect's “perspective” (e.g., functional v.s. object-oriented decomposition). Of course, different architects have unique perspectives, so if the same design problem were given to multiple architects, no two solutions/designs/modularizations would be the same. The resulting designs aren't completely arbitrary, but almost always the solutions are different enough so that modules from one project cannot be easily interchanged with modules from another, equivalent, project.

I argue that we teach software design from a geocentric viewpoint. Some of the critical issues that we want to express — evolvability, maintainability, simplicity, performance, etc. — are not captured in our designs. And consequently, dealing with these omitted issues is hard. To paraphrase an old refrain of Parnas, making conceptually simple changes to a software artifact is often out of proportion to the effort needed to make those changes. This is a classical symptom of geocentric designs. Geocentric designs are largely one-of-a-kind; they tend to be monolithic, hard to change, and hard to understand. Introducing new features, for example, or viewing the design from a different perspective tends to be excruciatingly difficult.

2 REFINEMENTS

I argue that there is a “heliocentric” perspective to software development that can simplify software design and related concerns for families of similar applications (a.k.a. *product-lines*). The unifying concept is the old and largely abandoned notion of *step-wise refinement* — i.e., the ability to create complex programs by progressively introducing implementation details into simpler programs [7]. The reason why practitioners have ignored it (although new efforts are on the horizon to revive it [3, 9]) is the scale of refinements. Classical refinements work on microscopic code fragments using program rewrite rules (e.g., $x + 0 \Rightarrow x$). The problem is that one must apply hundreds to hundreds of thousands of such rewrites in order to transform a compact specification into an admittedly small program.

Step-wise refinement is now being rediscovered under many different names and guises: the new twist is the scale.

Namely, a single refinement that is now the subject of discourse modifies *multiple* classes simultaneously and consistently. Such a refinement encapsulates the implementation (which could be arbitrarily complicated) of a “feature” that can be shared by a family of related programs. The benefit of this approach is that composing a few of these “large” refinements produces complex and sizable programs (e.g., a composition of 25 refinements yields a program of 10K lines to 70K lines of code). Names given to such refinements are (in historical order) layers, protocols, features, collaboration-based designs, subjects, and aspects [1, 6, 5, 4]. The technical details of their associated research programs is, to be certain, quite different and so too are their agendas. But still, there is a common and underlying theme. For example, an aspect language in aspect-oriented programming defines a refinement to be made, an aspect compiler is a function that takes an existing program and aspect specification as input and produces a refined program as output. The output program is a weaving the aspect into the original source code [5]. A layer in GenVoca is similar: a layer refines a given abstract interface by introducing implementation details of the feature represented by that layer. Like aspect compilers, a layer is a function that maps an input program (that does not have that particular feature) to an output program (that does have that feature) [1]. There are many other examples that fit this “scaled” refinement paradigm.

3 SEPARATION OF CONCERNS

How do refinements address the problem of “separation of concerns” or providing alternative perspectives of a software artifact? Why should refinements be more important than, say, OO classes? There are two answers.

First, a refinement is a very abstract concept. It allows us to codify the essence of a “feature” and its impact on a software artifact without requiring us to choose a particular implementation technology. That is, we can encode refinements as COM objects, or Java objects, or C++ templates, or as metaprograms (i.e., programs that generate other programs), or as rule-sets of program transformation systems. Refinements can be composed statically or dynamically. Of course, to implement a refinement requires choosing an implementation technology which imposes limitations, i.e., COM components are binaries that are composed only at application run-time whereas C++ templates are parameterized code fragments that are composed only at application compile-time. However, to understand a software domain one doesn’t necessarily have to understand (or commit to) an implementation technology up-front. This has the following benefit: one can understand a domain of applications in terms of implementation-free refinements; once this is done, one can then choose a technology for refinement implementation as multiple technologies might be appropriate. This flexibility is important: choosing an implementation technology first may force more effort be expended on minimizing the technology’s shortcomings rather than focusing on the general problem of application design. (This is akin to developing geocentric motion equations directly, rather than transforming a simple heliocentric equations into

corresponding equations for a particular coordinate system [2]).

Second, the focus of existing refinement research today is mostly on producing source code by composing refinements. The refinements themselves encapsulate *only* changes that are to be made to application source when that refinement is applied. Refinements are more general than this. When a new feature is added to a system (i.e., when a refinement is applied to a system), not only does its source code change, but so too does its *documentation*, *formal properties*, *performance models*, and so on. These are the “concerns” that architects have w.r.t. software design. Thus, a “complete” implementation of a refinement will encapsulate changes that are made to a software artifact across all “concerns” (source code, documentation, formal properties, etc.) that enable tools to analyze the impact of adding or removing this feature (refinement) from a given system.¹

These ideas are not far-fetched. We have had enormous success in building product-line architectures (i.e., component-based architectures where large families of applications can be synthesized solely through component composition) in this manner. The implementation of our refinements are called “components” (which probably is a bad choice of names). The domains that we and other people have been able to build using this approach include: databases, network protocols, avionics, radio software, extensible languages/compilers, and fire support simulators [2]. Not only can we generate source code for these systems (ranging from 5K lines to 70K lines of code), we can generate customized documentation, analyze formal properties, and generate application-specific performance models, all because we were able to encapsulate different concerns within the implementation of our refinements — i.e., the changes to different aspects of software artifacts that were of interest.

4 RECAP

It is easy to insist that one particular viewpoint of software development is more important than another. Each viewpoint can impose a different modularization scheme or a different way in which to view or analyze application source. Tools for each concern will use their own representations of a software artifact to perform their particular analysis (e.g., compilation, validation checks, etc.). Insisting that one scheme be used in favor of another may make it difficult to address other concerns.

We believe that each concern has its own representation of a software artifact, and the consistency of all necessary representations must be maintained. How modularization fits into this world-view is the challenge, because it seems orthogonal to concerns. We argue that refinements provide us a way to achieve fundamental modularizations and separation of concerns simultaneously. Years of research in different domains has taught us that the fundamental

1. Refinements are largely orthogonal to other refinements and do have constraints on their use. We have found that Perry’s light semantics are well suited for this task [8].

building blocks of application domain is captured by the abstract concept of refinement. A refinement defines the changes that are made to a software artifact when a new detail, feature, aspect, ... is added to a system. These changes are not limited to source code, but to other "concerns" like documentation, performance, and so on. By encapsulating within a refinement the changes to be made to all concerns of interest *and that these changes are consistent across concerns*, we have achieved a very powerful model of software construction that permits multiple viewpoints in a simple and coherent manner, while at the same time allowing us to be able to synthesize huge families of related applications.

For examples of a "large-scale" refinement-based approach to product-line development, we invite readers to visit our web page: <http://www.cs.utexas.edu/users/schwartz/>.

Acknowledgements. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

5 REFERENCES

- [1] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.
- [2] Don Batory, Product-Line Architectures, *Smalltalk und Java in Industrie und Ausbildung*, Erfurt, Germany, October 1998.
- [3] I. Baxter, "Design Maintenance Systems", *CACM*, April 1992, 73-89.
- [4] S. Cohen and L. Northrop, "Object-Oriented Technology and Domain Analysis", *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.
- [5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.
- [6] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-428.
- [7] H. Partsch and R. Steinbruggen, "Program Transformation Systems", *Computing Surveys*, March 1983, 199-236.
- [8] D.E. Perry, "The Logic of Propagation in The Inscope Environment", *ACM SIGSOFT 1989*.
- [9] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", Microsoft Corporation, September 1995.