Using Executable Domain Models to Implement Legacy Software Re-Engineering

Position Paper

Jean-Marc DeBaud

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
debaud@cc.gatech.edu

ABSTRACT

In this position paper, we advocate a domain-centric approach to the evolution of legacy systems. The migration of legacy systems is a difficult endeavor because traditional methods have two principal deficiencies. First, they fail to capture the context of a system, i.e., its domain. Second, the legacy system's comprehension results are not directly usable for the system evolution. We propose the construction of executable domain models to alleviate both problems. The construction of an executable domain model entails a process of domain analysis that leads to a domain model, as well as the transition of the former to an executable state. The domain model provides domain expectations that drive legacy system understanding. The executable domain model provides a medium in which the result of the legacy system comprehension can be recorded. In fact, the executable domain model is instantiated using the system requirements derived during program comprehension. The artifact thus created takes the role of the re-engineered program. Our work uses the technique of object-oriented frameworks (OOF) as the executable domain model representation.

1. INTRODUCTION

The process of software re-engineering is multi-phased. In the first phase, the re-engineering of a software artifact entails the comprehension of both *what* the artifact does, i.e., its context, and *how* it accomplishes its purpose, i.e., its structure and control flow. This phase traditionally corresponds to reverse engineering. In the second phase, the artifact is evolved using both the information gained in the first phase and its new specifications. Because of the difficulties of each of these phases, program re-engineering is a complex endeavor.

Research in the field is moving somewhat away from the question of how an artifact accomplishes its purpose. Using lexical, syntactic, and semantic rules for legal program constructs, tools such as Reasoning Systems' Software Refinery [9] can discover an artifact's structure and control flow. The purpose of an artifact is a much more difficult problem to solve.

Programs have a purpose. They exist because of some computational needs. But a computation has value only if it models or approximates some aspects of the real world. Insofar as the model is accurate, the program will succeed in performing as expected. And, to the extend that the model is comprehended by the reverse engineer, the process of understanding the program will be eased. Hence, to understand what a program does, one must understand the context in which it evolves; that is, the part of the world it is modeling or its application domain.

So to understand a program, one must understand the domain in which it revolves. To understand that domain, one must carefully analyse it. Research in computer science offers a technique, domain analysis, to do that. Once one has such a domain model, one can use it to drive program understanding.

To complete the re-engineering process, a software artifact must be evolved once it is fully comprehended. The form and representation used to record the comprehension of an artifact are critical to its utilization in the evolution step. Should the new evolution specifications concern the artifact's application domain, then it may be necessary for the programmer to use the domain model to implement them or to acquire some domain knowledge. Hence, the evolution step also benefits from the existence of a domain model.

In a previous paper [4] we have argued that application domain modeling provides key concepts to facilitates program context comprehension. In this position paper we argue that an executable application domain model provides a key technique to implement legacy software re-engineering.

2. THE METHOD AND RESULTS

The gist of our re-engineering method consists of the construction of an executable, domain-specific reuse infrastructure (or domain model) and its use to drive, record and evolve software artifacts. The method is presented in details in [5]. The main steps of the method (see Figure 1) are as follows: First, an application domain must be chosen. Second, a domain analysis is performed upon that realm and a domain model is created. Third, that model is expressed in executable form. The particular technology we use is object-oriented frameworks [6]. At that stage, what is indeed a domain-specific reuse infrastructure is in place. Fourth, the application domain model is used to guide artifact comprehension, i.e., the reverse engineering. By a process of instantiating the object-oriented framework, the results of the artifact comprehension are recorded. This process in fact amounts to specializing the reuse infrastructure according to the recovered artifact specifications. At the end of the artifact understanding process, its functionality is replicated by the framework. Now, as the fifth and last step, the evolution of the artifact can begin. This is done by augmenting and/or modifying the previous set of instantiations.

To apply this method, we have developed an executable domain model for Report-Writing. This is a stable, well understood domain that has been successfully modeled by database management system vendors in the form of report writing tools. We tested this approach on the Installation Materiel Condition Status Reporting System [3], a standard U.S. Army management information system. It consisted of approximately 10000 lines of COBOL code broken into 15 programs, most of which were writing reports.

While the process of domain analysis and framework construction is difficult and time consuming, we found our efforts rewarded in many respects. First, we experienced a substantial improvement in the time it took us to comprehend existing programs. We estimate, conservatively, to have speeded the understanding step by a factor of two. Second, recording the artifact specifications generated from the comprehension process was vastly simplified by simply having to parametrize the framework. Third,

artifact evolution was also greatly simplified because that process meant evolving the artifact specification as opposed to the source-code. Our experience shows that complex artifact evolution became a matter of minutes for someone familiar with the domain.

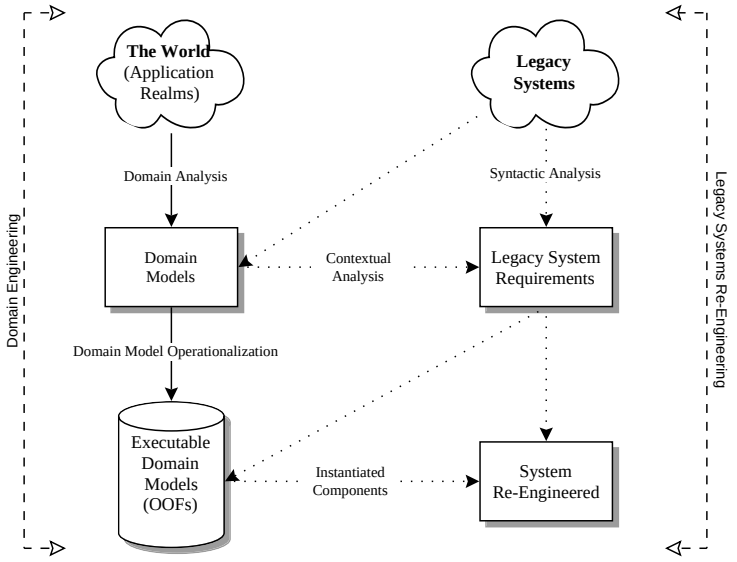

Figure 1: Using Domain Models for Software re-Engineering

Object-oriented frameworks provide a clear and normative structure to guide the reverse engineering effort through feature expectations and purpose patterns. To achieve this, domain analysis is crucial. Frameworks are also easily extended with new domain features (i.e., concrete classes), so long as no dramatic conceptual changes are made to the domain. Frameworks, like every model representation techniques are prone to difficulties when used to model fluid domains. Domain analysis must produce a model that is robust to changes; a non trivial task. Frameworks are based on the object model, making their overall understandability to users somewhat easier than other representations.

Having first developed the report writing framework for forward engineering purposes, we were impressed by the capabilities of the technique to record the process of reverse engineering via framework instantiation. It was an unexpected benefit. We were also impressed by the power of domain models to build expectations and their related prescription abilities.

3. DOMAIN MODELING AND REVERSE ENGINEERING

We now say a few words about the relation between domain modeling and reverse engineering. A domain is a problem area. Typically, many applications programs exist to solve the problem in a single domain. Arango and Prieto-Diaz [1] give the following prerequisites for the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain.

According to Neighbors [7] [8], domain analysis "is an attempt to identify the objects, operators, and relationships between what experts perceive to be important about the domain." As such, it bears a close resemblance to traditional systems analysis, but at the level of a collection of problems rather than a single one. Domain engineering/modeling/analysis is an emerging research area in software engineering. It is primarily concerned with understanding domains to support initial software development and reuse, but its artifacts and approaches prove useful in support of reverse engineering as well.

In order for domain analysis to be useful for software development, reuse, or reverse engineering, the results of the analysis must be captured and expressed, preferably, in a systematic fashion, hence the need for a representation method [2]. Among the aspects that might be included in such a representation are domain objects and their definitions, including both real world objects and concepts; solution strategies/plans/architectures; and a description of the boundary and other limits to the domain. An unresolved issue, of importance both to software developers and reverse engineers, is the exact form of the representation and the extent of its formality.

What role might a domain description play in reverse engineering a program? In general, a domain description can give the reverse engineer a set of expected constructs to look for in the code. These might be computer representations of real world objects or algorithms or overall architectural schemes. Because a domain is broader than any single problem in it, there may be expectations engendered by the domain representation that are not found in a specific program. Because a program is not always accurate or up-to-date, there may be things missing or incorrectly expressed in the program, despite contraindications in the domain representation. And, because a program is often used for more than one purpose, it may include components that do not appear at all in the domain representation.

Nevertheless, a domain representation can establish expectations to be confirmed in a program. Furthermore, the objects in the domain representation are related to each other and organized in prototypical ways that may likewise be recognized in the program. Hence, a domain representation can act as a schema for controlling the reverse engineering process and a template for organizing its results.

ACKNOWLEDGEMENT

The author gratefully acknowledge the original support of the Army Research Laboratory through contract DAKF 11-91-D-0004-0019 and helpful comments from Spencer Rugaber.

REFERENCES

[1] Arango, Guillermo and Prieto-Diaz, Ruben. Domain Analysis Concepts and Research Directions, in *Domain Analysis and Software Systems Modeling*, ed. Ruben Prieto-Diaz and Guillermo Arango ,IEEE Computer Society Press, 1991.

[2] Arango, Guillermo. Domain Analysis Methods. In *Software Reusability*. (Eds.) W. Schaeffer, R. Prieto-Diaz, and M. Matsumoto. Ellis Horwood, New York, 1993, pp. 17-49.

[3] *Automated Data Systems Manual, Installation Material Condition Status Reporting System (IMCSRS), Functional User's Manual*, Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[4] DeBaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. Domain Analysis and Reverse Engineering, *Proceedings of the Conference on Software Maintenance*, pp. 326-335, Victoria, British Columbia, September 1994.

[5] DeBaud, Jean-Marc, and Rugaber, Spencer. A Software Re-Engineering Method using Domain Models, To appear in *Proceedings of the Conference on Software Maintenance*, Nice, France, October 1995.

[6] Johnson, Ralph E. and Foote, Brian. Designing Reusable Classes. *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, pp 22-35

[7] Neighbors, James. “Software Contruction from Components”, PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.

[8] Neighbors, James. DRACO: A Method for Engineering Reusable Software Systems. 1989 ACM, Inc. Addison-Wesley Publishing Co., Reading MA.

[9] Reasoning Systems, Inc., Palo Alto, CA. REFINE User’s Guide, 1990. For REFINE (TM) version 3.0