A Generative Approach To Componentware

Marcelo Sant'Anna   Julio Cesar Sampaio do Prado Leite   Antonio Francisco do Prado

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente, 225.
Rio de Janeiro 22453-900
e-mail: {santanna, julio}@inf.puc-rio.br

Departamento de Computação
Universidade Federal de São Carlos
Av. Washington Luiz, 235
04499-610 São Paulo, Brazil
prado@dc.ufscar.br

Abstract

Componentware practice faces several problems to handle the evolutionary aspects of software. We focus on four major problems in the area and propose the use of domain-oriented generic software generators as one possible solution. Motivated by the Draco-PUC project, this proposal defines an agenda for improving the Draco-PUC component model and for making it interoperable with CORBA.

Keywords: *Component-based software construction, Software Generators, Software Evolution, Draco Paradigm, Software Architectures and Transformation Systems.*

1 Introduction

Component-based software construction has been proposed for some time [23], but just in the last five years it has become available to a larger amount of people around the world. This unprecedent growth of the reuse culture can be observed by the large sales of popular rapid-application-development (RAD) [9] tools, as well as by the crescent availability of components [34]. Added to the scene, the Internet/Intranet marketing has provided a fast growth in the interconnection of computers, inside and outside companies, creating a large demand for distribute interoperable applications [7], raising the complexity of software systems [31].

Learning from experience, software engineers are concerned with the difficulty of software evolution and its impacts on the costs od software engineering projects [21]. In this proposal we point out how software evolution problems are manifasted on current and near-future componentware and we also present possible paths towards handling these problems.

2 Componentware Today

The main actors in popular componentware scene today are OCX controls, Java classes and some OO toolkits. Also, Design Patterns [15], OO frameworks and template-based libraries, such as the Standard Template Library (STL) [24] are starting to play some minor roles in the field. Client/Server computing is extensivily sold as the natural scenario for these actors to perform their roles [10], where CORBA [33] and CORBA-like standards seem to be of increasing importance.

Studying the problem of software evolution on today's componentware, we understand that, in addition to the intrinsic problems of software evolution, there are specific factors which restrain the delivery of better results to the software development process.

As a basic premise to our research, we believe that these problems are mostly due to:

- The high-impedance among components
- The inflationary aspects of implementation-level encapsulation and aggregation
- The very narrow window of opportunity for systems optimization
- The lack of scalability for component libraries

2.2 High Software Impedance among Components

When an application is being built from components it is necessary to tie together several kinds of software components to fulfill the feature requirements. Components can either be easilly combined or may need extra glue in order to work together. The later case is extremely frequent in modern distributed applications and sometimes a rather complex plumbery is needed to make things work in an interoperable way. The harder it is to put components together, the higher is the impedance among them. The higher the impedance, the larger the number of glue-components we have to use. Several researchers have pointed to this specific kind of problem [26] [29] [1].

2.3 Inflationary Aspects of Implementation-level Encapsulation and Aggregation

As software spreads out to almost all activities in our daily lives, the intrinsic complexity of software systems tends to grow faster [31]. If we have plug-compatible components that can be connected in a tinkertoy-like manner to fulfill this crescent demand, it is natural that software-Frankensteins (poorly-composed software) come to life. We believe this is the result of unsing the concepts of aggregation an d encapsulation athe implementation level only. An immediate consequence is that, once a component is added to an application it will always be part of its implementation. As soon as aggregation provides a horizontal growth to our perception of the application, we use abstraction to hide details in a vertical dimension. The negative aspect of current practice is the application of this approach at the implementation level, when software systems end up carrying forever a great amount of useless lines of code, generating larger and more inefficient systems. In this case we have kind of a snowball effect applied to software systems. As a consequence maintanability becomes more problematic.

2.3 Narrow Opportunities for Optimization

The lack of opportunities for making optimizations and maintenance as software systems grow in size is characteristic of current practice component-based software construction tools. Anyone attempting to maintain a large system, using any of currently popular RAD tools, know how tough this job is. Being focused on the implementation level, these tools are not able to apply domain-specific optimizations. If one desires to improve the design, maintenance will take place at the code level, severely impacting the software evolution process. Neighbors and Baxter provide very good insights about this fact [25] [26] [5]. A notable evidence for this is the proliferation of resource-hungry unoptimized software (sometimes called *fatware*).

2.4 Library Scaling Problem

As noted by researchers such as Ted Biggerstaff [8] and Don Batory [3], current component-based software practice for component libraries present an inherited difficulty to scale up. More precisely, following Biggerstaff [8], “the library scaling problem resides in the difficulty in scaling reuse libraries in both component sizes and feature variations”. Anyone wanting to support a large set of features in a component library, must also handle the managerial aspects of having to deal with a large library, when it does not become unpractical to build such a large library. Since libraries can either be domain-specific or generic, there are two orthogonal dimensions of growth, where portability among different platforms and diversity of features are driving factors.

3 Searching for a Better Componentware Practice

We have, so far, pointed out current problems of componentware practice without giving proper recognition of its importance to software reuse. We believe that current practice is slowly preparing practitioners for more sophisticated software engineering approaches. Our research focus exactly on this window of opportunity.

In order to tackle the afore mentioned problems, our work is focused primarily on the use *of domain-oriented generic software generators* which, we believe, can support the evolutionary aspects of software.

Several researchers have *proposed generic software generators* (GSG), but none of them, except Neighbors in [28], has firmly worked out an example of a domain-oriented GSG. In fact, it was Neighbors who forged, for the software engineering community, the now wide-spread term *Domain Analysis* [25]. Good work on the problems of constructing software generators is reported by Ornburn in [30]. Feather [12], Partsch [32], Fickas [13] and Baxter [5] have managed to compare several different approaches, aiding the field of GSGs. A large european experiment on the area, Prospectra, is reported in [17]. As far as we know, only Kestrel Institute [18], through Reasoning Systems [22], really succeeded to transfer its technology to industry in order to produce a commercial working GSG. Current work in the area can be found in [19], [4], [14], and [20]. Some insights on the way that software generators can aid software engineering in the following years can be found in [6], [26] and [7].

3.1 Our Understanding of Generative

A generative approach to software assumes that it is possible to describe generic (abstract) architecture for software so they can be further, automatically or semi-automatically, composed in order to produce working (concrete) software through generative tools. This view understands that software can assume different stages on a gradient of density, from more-abstract to more-concrete software. This view is grounded on computer science concepts of Abstraction and Refinement [2] [35]. We understand that software generators can just produce concrete software if there are sets of available abstractions, with associated interpretations, that can be used according to pre-defined laws of compositions.

A *generic software generator* (GSG) is one which can be used on several scenarios, being customizable to fulfill the generative needs for the production of several different kinds of software, rather than being limited to a very specific class of software. As such, we define a *domain-oriented GSG* as a GSG in which the features of a generic architecture are segmented and encapsulated through the several domains it encompass. A domain model must be clearly defined because, in this kind of tool, domains play a major role. This is the kind of generative approach we are currently pursuing on our research.

4 A Generative Architecture: Draco Machine Revisited

Given the previous sketched thoughts, we believe, a generative tool should minimally address the following requirements:

- A component model where the protocol of interfaces is formally specified. In this way, the component builder's view on how it can be used is explicitly defined. Also, as a consequence of proceeding like this, adapters can be automatically produced when we want to tie components.
- Abstract specifications as the primary source of description for software systems, so it is possible to reason about and choose among multiple choices of concrete implementations during the design process.
- Use of a domain-specific scheme for the encapsulation of system knowledge so that common concepts can be reused, at the requirements ans specification levels. As a consequence, there is a possibility to provide pre and post requirements traceability [16].
- Generative production of concrete componentry so that specific parts of it can be constructed on-the-fly, by demand of instantiated applications, rather than using a statically exhaustive set of components libraries.

Our main motivation for this research is our work at the Draco-PUC project [1]]. In this project, we are putting forward the ideas of the Draco paradigm [25], by building a system that makes possible experimentation with domain-oriented software production. Until now, our research has been deeply biased towards the transformational mechanism which gives support for Draco-PUC machine. This proposal aims to develop a Draco-PUC concept of components. In order to attain this goal, we believe the following tasks should be tackled:

- Development of a component model that address our understanding of the problem with current componentware practice.
- Building laguages and mechanisms in Draco-PUC which enable the use of the proposed component model.
- Making Draco-PUC and CORBA component models interoperable at the implementation level.

Currently, we are studying the requirements for Draco-PUC component model and we are also conducting experiments with CORBA so we can better understand its workings.

5 Conclusion

This work is investigating how domain-oriented GSGs can help us in developing component-based software, in a way to overcome current failures with the practice of componentware. Studying popular RAD tools, we have pinpointed four main probleem factors: the high-impedance among components, the inflationary aspects of implementation-level encapsulation and aggregation, the narrow window of opportunity for systems optimization and the lack of scalability for component libraries. Motivated by the Draco-PUC project and by Neighbor's vision of components, we are researching paths to put into practice a domain-oriiented GSG that can handle these problems of evolutionary componentware.

Following our agenda of tasks, we are currently studying requirements for a component model, which will be followed by the construction of languages and mechanisms for the Draco-PUC machine in order to put this component model in practice. A point of great interest to us is optimization, so we can develop a model where final applications can carry efficient essential code only. To attain this objective, we believe we can stand firmly on the transformational technology we have already developed in the Draco-PUC project. Further in our research path, at the implementation level, we also expect to make Draco-PUC and CORBA interoperable, so that Draco-PUC components can be tied together with CORBA objects, as well as allowing CORBA components to be incorporated into Draco-PUC domains.

References

[1] P. S. C. Alencar, D. D. Cowan, C. J. P. Lucena, and T. Nelson. Towards a Formal Link Between Viewpoints in Analysis and Implementation. In *#rd. OOPSLA Workshop on Subjectivity*, San Jose, California, October 1996.

[2] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25, 198.

[3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Libraries. In *ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering*, Los Angeles, California, December 1993. ACM.

[4] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software (Issue on Systematic Reuse)*, 11(5), September 1994.

[5] I. D. Baxter. *Transformational Maintenance by Reuse of Design Histories.* PhD thesis, University of California at Irvine, 1990.

[6] I. D. Baxter. Generators as Key to Effective Software Reuse. In M. Sitaraman, editor, *4th International Conference on*

Software Reusability, page 218, Orlando, Florida, April 1996. IEEE Press.

[7] M. Betz. Interoperable objects. *Dr. Dobb's Journal*, 19(11):cover story, October 1994.

[8] T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In W. B. Frakes, editor, *3rd International Conference on Software Reusability*, pages 102-109, Rio de Janeiro, Brazil, November 1994. IEEE Press.

[10] T. E. Carone. Client/Server development. *Dr. Dobb's Journal*, 21(11):cover story, November 1996.

[11] J. C. S. do Prado Leite, M. Sant'Anna, and F. G. de Freitas. Draco-PUC: a Technology Assembly for Domain Oriented Software Development. In W. B. Frakes, editor, *3rd International Conference on Software Reusability*, pages 102-109, Rio de Janeiro, Brazil, November 1994. IEEE Press.

[12] M. S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. In *IFIP WG2.1 Working Conference on Program Specification and Transformation*, Bad Toelz, Germany, April 1986.

[13] S. F. Fickas. Automating the Transformational Development of Software. *IEEE Transactions on Software Engineering*, SE-11(11):1268-1277, November 1985.

[14] J. Floch. Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator. In *17th International Conference on Software Engineering*, pages 21-219, Seattle, Washington, April 1995. IEEE Press.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.*Addison-Wesley, 1994.

[16] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of The Requirements Traceability Problem. In *1st International Conference on Requirements Engineering*, pages 94-101, Colorado Springs, 1994. IEEE Press.

[17] B. Hoffmann and B. Krieg-Bruckner. *Program Development by Specification and Transformation*. Springer-Verlag, 1993.

[18] R. K. Jullig. Applying Formal Software Synthesis. *IEEE Software (Issue on Software Synthesis)*, May 1993.

[20] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kopov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A Software Engineering Experiment in Software Component Generation. In *18th International Conference on Software Engineering*, pages 542-552, Berlin, Germany, March, 1996. IEEE Press.

[21] M. M. Lehman. Laws of Software Evolution Revisited. In *EWSPT'96*, Nancy, October 1996.

[22] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an Enabling Technology to Reengineer Legacy Systems. *Communications of the ACM*, 37(5), May 1994.

[23] D. McInroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138-155. NATO Science Comitee Report, 1968.

[24] D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming With the Standard Template Library.* Addison-Wesley, Reading, MA, 1996.

[25] J. M. Neighbors. The DracoApproach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, SE-10(5):564-574, September 1984.

[26] J. M. Neighbors. An Assessment of Reuse Technology After Ten Years. In W. B. Frakes, editor, *3rd International Conference on Software Reusability*, pages 102-109, Rio de Janeiro, Brazil, November 1994. IEEE Press.

[27] J. M. Neighbors. The Benefits of Generators for Reuse. In M. Sitaraman, editor, *4th International Conference on Software Reusability*, page 218, Orlando, Florida, April 1996. IEEE Press.

[28] J. M. Neighbors, G. Arango, and J. Leite. *Draco 1.3 User's Manual*. University of California at Irvine, September 1984.

[29] G. S. Novak. Creation of Views for Reuse of Software with Different Data Representations. *IEEE Transactions on Software Engineering*, 21(12):993-1005, December 1995.

[30] S. B. Ornburn and R. J. LeBlanc. Building, Modifying and Using Component Generators. In *15th International Conference on Software Engineering*, pages 391-402, Baltimore, Maryland, April 1993. IEEE Press.

[31] D. L. Parnas. Fighting Complexity. *IEEE Engineering of Complex Computer Systems Newsletter*, 2(2), October 1995.

[32] H. Partsch and R. Steinbruggen. Program Transformation Systems. *Computing Surveys*, 15(3):199-236, September 1983.

[33] J. Siegel. *Corba Fundamentals and Programming*. Johm Wiley & Sons, 1996.

[34] J. Udeli. Componentware. *Byte Magazine*, 19(5):cover story, May 1994.

[35] M. Ward. *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1989