2nd WORKSHOP ON DOMAIN-SPECIFIC VISUAL LANGUAGES

An OOPSLA 2002 Workshop, 4th of **November, 2002, Seattle, WA, USA**

Program Committee

Pierre America
Philip T. Cox
Krzysztof Czarnecki
Jeff Gray
Steven Kelly
Kalle Lyytinen
David Oglesby
Jyrki Okkonen
Matti Rossi
Juha-Pekka Tolvanen
Sharon White

INTRODUCTION TO THE 2nd WORKSHOP ON DOMAIN-SPECIFIC VISUAL LANGUAGES

Workshop web site: http://www.cis.uab.edu/info/OOPSLA-DSVL2/

Juha-Pekka Tolvanen
MetaCase Consulting
Ylistonmentie 31
FIN-40500 Jyvaskyla, Finland
jpt@metacase.com

Jeff Gray
University of Alabama at Birmingham
Computer & Information Sciences
115A Campbell Hall, 1300 University Blvd.
Birmingham, AL 35294-1170, USA
gray@cis.uab.edu

Matti Rossi
Helsinki School of Economics
FIN-00100 Helsinki, Finland
mrossi@hkkk.fi

ABSTRACT

Current modeling languages are based on the concepts taken from programming languages, leading to a poor mapping to organizations' own domains and duplication of effort in problem solving, design and coding. Domain-specific languages allow faster development of applications, based on models of the product rather than on models of the code. A domain-specific modeling language applies concepts and rules found in the domain. Together with generators and components it can automate a large portion of software production. In these proceedings we report on recent advancements in this area.

1. INTRODUCTION

An upward shift in abstraction leads to a corresponding increase in productivity. In the past this has occurred when programming languages have evolved towards a higher level of abstraction. Today, domain-specific visual languages (DSVL) provide a viable solution for continuing to raise the level of abstraction beyond coding.

Domain-specific visual languages raise the level of abstraction, while at the same time narrowing down the design space, often to a single range of products for a single company. In a DSVL, the models are made up of elements representing things that are part of the domain world, not the code world [5]. The language follows the domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts. The models are simultaneously the design, implementation and documentation of the system, which can be generated directly from them [1].

This is unlike current visual modeling languages that are based on the code world using the semantically well-defined concepts of programming languages (like UML, SA/SD). Here, developers have to leap straight from requirements into implementation concepts, and map back and forth between domain concepts, UML concepts, and program code. This requires a lot of time and resources and easily leads to errors.

With a DSVL, the problem is solved only once by modeling the solution using familiar domain concepts. The final products are automatically generated from these high-level specifications with domain-specific code generators [4, 5]. There is no longer any need to make error-prone mappings from domain concepts to design concepts and on to programming language concepts. Industrial experiences of this approach show major improvements in productivity, time-to-market responsiveness and training time [2, 6].

2. MAKING AND USING DOMAIN-SPECIFIC VISUAL LANGUAGES

Three things are necessary to achieve full automatic code generation from domain modeling: firstly a modeling tool supporting a domain-specific visual language, secondly a code generator, and lastly a domain-specific component library. Figure 1 shows these three elements at two levels, definition and use.

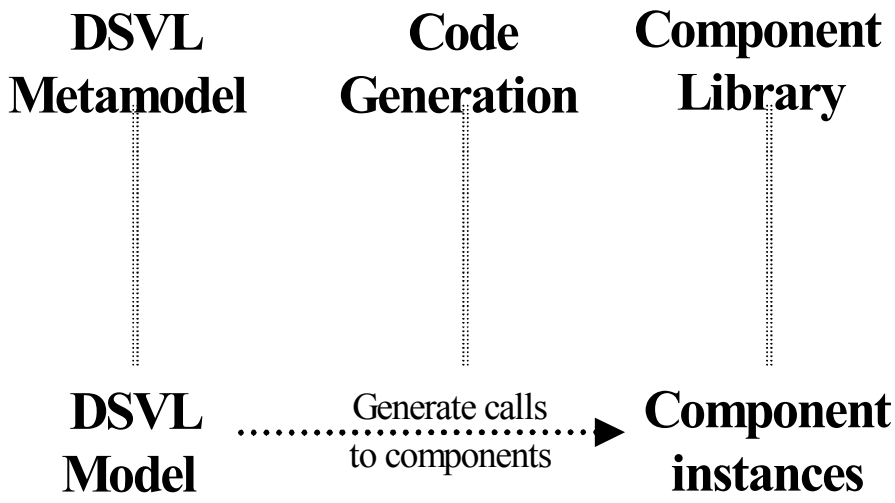

Figure 1. Framework for domain-specific visual languages

The top level is made once by the organization for a given domain. This forms the start-up cost of the DSVL approach.

Normally one or two experts will make the DSVL metamodel and code generation, normally with a metaCASE tool [4, 5]. The DSVL metamodel is the implementation of the domain-specific visual language, and includes the concepts and rules directly from the domain. The components will often have been made by developers in earlier projects in the domain, with some being added or modified specifically for the DSVL project.

The bottom level's process of making a DSVL model and generating its code is performed many times, once for each product, by normal developers. Development time can often be further reduced by reusing parts of the DSVL model which are common to several products. The code generation and component instances require no effort by the developer. No time need be spent on separate design and documentation of the product implementation, because the DSVL model itself already fulfils these roles. Together, these savings form the primary payback of the DSVL approach.

The DSVL model is built up of instances of the domain concepts from the DSVL metamodel, in accordance with the rules. The code generation walks through the model and transforms the concept structures into code. In some cases the code will be fully self-contained; more often significant parts of the code will be calls to components. Since the code is generated, syntax and logic errors do not normally occur, and the resultant improvement in quality forms a significant secondary payback of the DSVL approach.

3. ABOUT THE ARTICLES IN THESE PROCEEDINGS

The papers in this compilation present twelve different views to DSVL research and practice. The papers are divided into four sections, each comprising one workgroup in the actual workshop. In the first section we have four papers that explore the theory and practice of developing DSVL's. This section begins with Bottoni et. al's article on representations of

transitions in DSVL's. The second Chapter by Brunette presents a visual framework for developing reactive applications. In the third article Guizzardi presents a model for ontology development. The section is concluded with Bettin's analysis of productivity measuring between a DSVL and UML.

The second section presents four cases of developed DSVL's. The first example is a visual language for courseware authoring called CAPE, by Howard. Heberling, Maier and Tensi present an architecture modeling case of a large bank in chapter six. In chapter seven Kozaczynski and Thario develop a simple for modeling interactions between users and web applications.

In the third section three views to tools for DSVL modeling are presented. First Kienle et. al. tackle the practical problems of DSVL adoption, then in chapter nine Schloegel, Oglesby & Engstrom discuss the areas, where next generation metamodeling tools could improve the current state-of-the-art. In the third tool article Linvald & Østerbye tailor UML for use in ERP development.

In the fourth section DSVL evolution and introduction to organizations is tackled. Sprinkle & al. develop tools for handling the evolution of metamodels of DSVL's. The important aspect of reusability in domain specific languages is presented by Korhonen and last, but not least, Schmidt & al. present the SIMtelligence designer, which is a tool for specifying SIM toolkit applications. These papers give an excellent snapshot into the current state-of-the-art in DSVL research.

REFERENCES

[1] Tolvanen, J-P, Kelly, S., Gray, J., Lyytinen, K., Proceedings of OOPSLA workshop on Domain-Specific Visual Languages, Tampa Bay, Florida, USA, University of Jyväskylä, Technical Reports, TR-26, Finland, 2001.

[2] Kelly, S., Tolvanen, J.-P., (2000) Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, International workshop on Model Engineering, ECOOP 2000, (ed. J. Bezivin, J. Ernst)

[3] Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press, March, 1996.

[4] Lédeczi, A., et al., "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001.

[5] Pohjonen, R., and Kelly, S., "Domain-Specific Modeling," *Dr. Dobbs Journal*, August 2002.

[6] Weiss, D., Lai, C. T. R., Software Product-line Engineering, Addison Wesley Longman, 1999