

Design Wizards for Software Product Lines

Guillermo Jiménez-Pérez

Centro de Investigación en Informática

Instituto Tecnológico y de Estudios Superiores de Monterrey

E. Garza Sada, 2501 Sur, Monterrey, N.L., México 64849

gjimenez@campus.mty.itesm.mx

Abstract

Large software applications have complex architectures and constraints restricting how components can be composed. The infrastructure to implement product lines for these applications require the support of software tools. To be effective in constructing software families, tools need structural and semantic domain knowledge. Our work addresses the aspects concerned with the development of tools that incorporate both types of knowledge to assist developers in rapidly producing members of a product line. These tools allow developers to construct applications from high-level specifications thus relieving them from the burden of dealing themselves with architectural details.

1 Introduction

A *system family* or *product line* is a set of software applications sharing common aspects and with an anticipated variability. A *product-line architecture* captures the variability and commonality aspects for a system family. Effectively constructing product line architectures and their realization into software products require special methods and tools. Development methods whose aim is the production of a single software system are not enough for producing system families [Cza99]. Methods for constructing product lines need to apply *domain engineering* approaches [JGJ97]. Different methods for conducting domain engineering and their implementation into product lines have been outlined [Wei99, Cza99]. However, too much work remains to be done on detailing such methods and their supporting tools [Gri99].

One unavoidable step in product line engineering is the production of a reference or product-line architecture prescribing components, defining their interrelationships, and imposing constraints in their composability:

- **Components.** Defining what a software component is and is not depends on how we want to take advantage of using it [Szy97]. Here we limit our perception of components as being the constituent parts of the software architecture. In a reference architecture, components are assumed to be reused to construct different application systems [Bat99].
- **Interrelationships.** A software architecture describes how components are connected to one another. The set of connections defines how components collaborate and depend on the others. How components are related defines the architectural style.
- **Constraints.** A reference architecture will be realized into multiple software products. Different software products will share a subset of architectural components. Which components can be absent in a particular application and how present components should be interrelated needs to be defined, thus meaningful compositions can be constructed. Constraints are semantic rules restricting how components can be combined to construct applications [Bat97].

2 Design wizards for product lines

A *design wizard* is a software tool incorporating information of a reference architecture, implemented components, interrelations, and constraints governing valid compositions. Design wizards are generator tools for the automatic production of system families. The elements in a design wizard are a component base and a design rule base. The *design rule base* contains constraint declarations that should be met, and valid interrelationships that a component composition may have. The *component base* is a container of parameterized component implementations.

Design wizards take high-level application specifications, verify specifications against rules and, if everything is correct,

generate the corresponding applications. For application generation, the software generator performs two tasks: adapt components to fit in a composition, and generates weaving code that glues components into applications.

This simple idea of design wizards requires investigating:

- **Architectural description mechanisms and notations** for describing domain commonality and variability, and serve also to communicate requirements with domain experts.
- **Component implementation and parameterization mechanisms.** If we want our approach to be broadly useful among programming languages and development environments, mechanisms should be applied consistently for different development environments and different application domains.
- **Careful design of specification interfaces.** Since much domain and structural knowledge is implemented in the wizard, the user interface needs to be flexible.
- **A domain engineering method.** Different approaches to domain engineering have to be applied for producing a design wizard

We are in the final stage of implementing three different design wizards for three disparate domains, using different development environments. One is the production of computer numerical control systems for different machine tools, which are characterized by real-time constraints; this first domain is being implemented in C++. Other is the development of a family of simulators for intelligently controlled vehicles, whose implementation is being done in Java. The third is a general ledger information system, which is being developed in object-oriented Pascal using a commercial tool. We are using faceted classification [KCH+90] for domain analysis, and the GenVoca model [Bat99] to build the architectures. Our resulting models are hierarchical reference architectures whose layers are then implemented by parameterized and interchangeable components.

Design wizards are an architecture-driven, component-based and generator centric tools encapsulating application configuration knowledge for rapidly generating members in a product line. In the process of constructing design wizards, we have come to realize that we need two different modeling approaches: one for the domain (GenVoca) and one for the wizard tool itself (UML).

In our experiments, we have needed to extend Java and object-oriented Pascal to support component parameterization. In each case, we implemented simple preprocessors for parameter substitution. Even though our experiments have been oriented to augment the native development environments (i.e., the wizard producing Java applications is implemented in Java, the wizard producing C++ applications is implemented in C++, etc.) there is no reason why a cross-generator design wizard could not work (i.e., a design wizard producing code in a programming language different from that in which it is implemented).

3 References

- [Bat97] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators". In *IEEE Transactions on Software Engineering (Special Issue on Software Reuse)*, February 1997, pages 67-82.
- [Bat99] D. Batory and Y. Smaragdakis. "Building Product-Lines with Mixin-Layers", *ECOOP 99 Workshop on Product-Line Architectures*.
- [Cza99] K. Czarnecki, and U. W. Eisenecker. "Components and Generative Programming"; ESEC/FSE'99, Invited talk.
- [Gri99] M. L. Griss. "Domain Engineering and Reuse"; *IEEE Computer*, Roundtable on Software Development Trends. May 1999.
- [JGJ97] I. Jacobson, M. L. Griss, and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley, 1997.
- [KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study" (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.
- [Szy97] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1997.
- [Wei99] D. M. Weiss and C. T. Lai. *Software Product-Line Engineering*, Addison Wesley, July 1999.