Abstract

We would like to understand the interests of our target audience. Please register at **www.softmetaware.com/whitepapers.html** to provide us with some information about yourself, and to obtain access to the full content of all SoftMetaWare white papers.

Complexity & Dependency Management

Creating an Environment for
Software Asset Evolution and Software Mass Customization

Author: Jorn Bettin

Version 0.2

June 2004

SoftMetaWare

Revision History

Version	Author	Description
0.1- 0.2	Jorn Bettin	Initial version, June 2004

1 Introduction

This paper addresses the question of how to successfully create durable and scalable software architectures that enable the underlying design intent of the system to survive over a period of many years, such that no accidental dependencies are introduced as part of further software development and maintenance. Some may question whether this goal is achievable. In real-world software projects the concern about software architecture usually increases with team size, but in the end, when it comes to building a system, software is cobbled together without effective control over the dependencies that are created between various pieces. In order to prevent long-term design degradation, and in order to efficiently execute software development in-the-large, the introduction of dependencies between components needs to be *actively managed*, relying on tools for the mechanical enforcement of standards and rules.

Of course model-driven generation can be used to enforce specific patterns and to prevent arbitrary dependencies. That still leaves the work of those who produce prototypes and reference implementations from which generator configurations and templates are derived. If a single person produces the complete reference implementation, and if that person consciously and carefully thinks about every dependency in the design, then all is fine. In reality, consciously managing the impact of every dependency is only possible by adhering to a set of guiding principles for encapsulation and abstraction, even in a single-developer project. The larger the software development effort, the greater the need for conscious dependency management.

The principles described in this document are a natural extension to the six principles for design by contract.

- Whereas the design-by-contract principles focus on precision in the specification of component behavior,
- the *principles for fully externalized interface definitions* described in this document focus on precision in the delimitation of types that can be used in the communication between components.

The latter principles have a direct impact on the package/module/component structure of software code, and therefore their correct application can easily be verified (mechanically) in the code base. If correct usage is confirmed, then a (potential) component user can understand what each component is doing by looking at the component specification, without having to look inside.

The concept of fully externalized interfaces adds value to all non-trivial software development projects, and is particularly well suited for use in a model-driven software development process. A major influence on the material developed in this paper is the concept of *uniformity* that underpins the KobrA software product line approach [ABKLMPWZ 2002], in which every behavior-rich software entity is treated uniformly, regardless of its granularity or location in the "containment tree" that describes a software system. In other words software entities - which in KobrA are called "Komponents" - have the property of a *fractal*.

Taken together, the two sets of principles mentioned above form the foundation for Industrialized Software Asset Development, serving the following goals:

- Promotion of reuse and pluggability of software assets[1]
- Provision of patterns for the design of Open Source software assets that help maximize the (re)use potential of Open Source software assets
- Providing support for the mass-customization[2] of software products, i.e. the management of variability in a software product family
- Provision of a set of simple rules that limit the creation of dependencies between software assets, so that Model-Driven Software Development tools can be used to enforce software architecture specified in the form of models.

In describing standards for software component development, we work from the inside out: starting with a single component and its specification, then looking at its use, at customization/extension, and lastly taking one step back and raising the level of abstraction by one level.

2 Component Specifications

2.1 The Problematic Side of Design Patterns

Design patterns - and patterns in general it seems - tend to encourage describing the solution in more precise terms than the problem, i.e. by leading from context to the solution, they focus inwards, on the "how" rather than the "what". There are exceptions, but software pattern authors generally tend to prefer precise descriptions of a solution over precise descriptions of the context to which the solution applies. In a model and domain-driven approach however, the need for precision starts at the level of abstraction of the problem space. Ironically, the principles for externalized interface definitions can of course be described as a design pattern.

If we want to build durable software, we need to use component specification techniques that force us to focus on understanding the "what" first. What services do clients want from my component? And precisely what is the "language" that my clients understand or want to talk in?

A classical design pattern to control and limit dependencies between components/subsystems is the façade pattern [GoF 1994], and that's about as far as most projects worry about dependencies. In fact, the façade pattern is a good example of a pattern that is obviously very useful, but is not specific enough to achieve active dependency management without further qualification. The usage example in the GoF book completely ignores the dependencies created by the types used in the signatures of the operations exposed in the façade. The particular types used in the operations of the example seem to reside inside the subsystem that is supposed to be encapsulated by the façade. The façade pattern says nothing about

[1] Model-Driven Software Development defines a *software asset* as "anything from models, components, frameworks, generators, to languages and techniques".

[2] *Mass customization* meets the requirements of increasingly heterogeneous markets by "producing goods and services to match individual customer's needs with near mass production efficiency".