Foreword for Practical Software Factories in .NET

This evolution of software technologies over the past five decades has involved the creation of languages and platforms that help developers program more in terms of their design intent, such as architectural concepts and abstractions, and shield them from the complexities of the underlying computing substrate, such as CPU, memory, and network devices. After years of progress, many projects today use third-generation programming languages, such as Java, C++, and C#, and middleware run-time platforms, such as service-oriented architectures and web services. Despite these improvements, however, the level of abstraction at which software is developed still remains low relative to the concepts and concerns of the application domains themselves. As a result, too much time and effort is spent manually rediscovering and reinventing solutions to common domain requirements, which has led to the situation where the majority of software projects are - late, over budget, and defect ridden.

These problems occur for various reasons. For example, most application and platform code is handcrafted using third-generation languages. This manual approach incurs excessive time and effort due to complexity stemming from the semantic gap between the design intent and the expression of this intent in third-generation languages, which convey domain semantics and design intent poorly. This semantic gap is particularly noticeable and problematic for integration-related activities, such as system deployment, configuration, and quality assurance, that software developers perform when assembling applications using off-the-shelf components. Third-generation languages also force developers to focus on numerous tactical imperative programming details that distract them from strategic concerns, such as satisfying user needs and meeting quality requirements.

A related set of problems stem from the growth of platform complexity, which has evolved faster than the ability of third-generation languages to mask this complexity. For example, popular middleware platforms, such as J2EE, .NET, and CORBA, contain thousands of classes and methods with many intricate dependencies and subtle side-effects that require considerable effort and experience to program, tune, and maintain properly. Moreover, since these platforms often evolve rapidly – and new platforms appear regularly – developers expend considerable effort manually porting application code to different platforms or newer versions of the same platform. Due to these types of problems, platform technologies have become so complex that developers spend years mastering – and wrestling with – platform APIs and usage patterns, and are often familiar with only a small subset of the platforms they use regularly.

Without significant enhancements in our languages, platforms, and development processes, therefore, we will be unable to meet increasing customer demands for large quantities of quality software. Achieving the necessary levels of productivity and quality requires a movement away from today's guild-based software paradigm, where applications are largely handcrafted as "one-off" custom artifacts using third-generation languages and general-purpose middleware platforms. We instead need to shift closer to a manufacturing paradigm, where applications are assembled from configurable software components using domain-specific tools and automated quality assurance processes.

Various efforts in the past several decades have attempted to address the problems described above. An effort begun in the 1980's was *Computer Aided Software Engineering*

(CASE). One goal of CASE was to enable developers to write programs more quickly and correctly by using general-purpose graphical notations, such as state machines, structure diagrams, and dataflow diagrams. In theory, these graphical notations are more effective at expressing design intent than using conventional third-generation programming languages. Another goal was to synthesize implementation artifacts from graphical representations to reduce the effort of manually coding, debugging, and porting programs.

Although CASE tools attracted considerable attention in the research community and trade literature, they weren't adopted widely in practice for various reasons, including:

- Early CASE tools tried to generate entire applications, including the business logic and the software substrate, which led to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with code from other sources.
- It was hard to achieve round-trip engineering that is moves model representations back and forth seamlessly with synthesized code.
- Due to the simplicity of the notations for representing behavior, CASE tools have largely been applicable to a few domains, such as telecom call processing, that map nicely onto state machines.
- Due to the lack of powerful and common middleware, CASE tools targeted proprietary execution platforms, which made it hard to integrate the code they generated with other software languages and run-time environments.

This book focuses on a more practical and effective approach to address problems resulting from the complexity of platforms – and the inability of third-generation languages to alleviate this complexity and express domain concepts effectively. This approach is called *Software Factories* and is based on four core concepts that are well established in today's leading software organizations and projects: *architectural frameworks*, *context-based guidance*, *domain-specific languages*, and *product line architectures*. Software Factories combine and extend related technologies associated with these four core concepts, including patterns, models, object-oriented frameworks, and tools, to help industrialize the development of software. In particular, Software Factories enable the rapid assembly and configuration of separately developed, self describing, location-independent components to produce families of similar but distinct software systems.

At the heart of the Software Factory paradigm are product line architectures, which consist of object-oriented frameworks whose designs capture recurring patterns, structures, connectors, and control flow in application domains, along with the points of variation explicitly allowed among these entities. Product line architectures are typically designed using common/variability analysis, which captures key characteristics of software product lines, including

- *Scope*, which defines the domains and context of a product line,
- *Commonalities*, which describe the attributes that recur across all members of a family of products,
- *Variabilities*, which describe the attributes unique to different members of a family of products, and

- *Extension points*, which can be used to add new features to product variants and accommodate features outside the scope of the product line.

Although product line architectures are a powerful technology, they have historically been hard to use and even harder to develop and evolve using conventional third-generation languages and run-time platforms. A key contribution of this book is therefore its focus on developing Software Factories with a high degree on automation using *model-driven development* and *context-based guidance*. Model-driven development helps raise the level of abstraction and narrow the gap between application and solution domains by combining

- *Metamodeling* and *model interpreters* to create *domain-specific languages* that help automate repetitive tasks in product line architectures that must be accomplished for each product instance, such as generating code to glue components together or synthesizing deployment artifacts for middleware platforms, and
- *Commonality/variability analysis* and *object-oriented extensibility capabilities* to create *domain-specific frameworks* that factor out common usage patterns in a domain into reusable middleware platforms, which help reduce the complexity of designing domain-specific languages by simplifying the code generated from models.

Automated context-based guidance provides product developers with suggestions on what activities to perform in a particular context, such as changing the solution structure, providing custom activities in the context they are applicable, and displaying context specific help and guidance to developers. This guidance can be created by domain experts, systems engineers, and software architects to convey best practices and semantic constraints to developers using tools that help automate these activities and integrate them into products.

Although the concepts underlying Software Factories have been published before, this book breaks new ground by showing detailed examples of how model-driven development tools and context-based guidance can be used in software projects today. The book presents a case study using the Microsoft .NET Framework, C#, and other .Net related tools to specify, analyze, optimize, synthesize, validate, and deploy Software Factories, that can be customized for the needs of a wide range of software systems. The case study covered in the book includes desktop applications based on the Smart Client Baseline Architecture (SCBAT) from the Microsoft Patterns and Practices Group, local services, and third-party services that are accessed via the Internet.

The book covers the entire process of specification, design, implementation, deployment and use of a Software Factory. It also shows how Software Factory schemas and templates can be used to combine separate software technologies into a coherent application development paradigm. In addition, the book shows how to develop domain-specific languages and guidance packages that incorporate best practices to help eliminate common errors when developing systems in particular application domains. The entire case study can be downloaded and used as a comprehensive reference. While the Software Factory paradigm can be used with any language and platform, these working examples are invaluable to reinforce key concepts and principles when implementing your own Software Factories.

It’s been my experience developing and applying advanced software technologies over the past two decades that mastering the concepts and technologies described in this book is hard without effective guidance from experts who can guide you step-by-step. We are therefore fortunate that Gunther and Christoph have found time in their busy lives to write a book on practical Software Factories in .NET. If you want thorough coverage of the technologies that will shape the next generation of software applications read this book. I've learned much from it and I'm sure you will too.

Douglas C. Schmidt

Vanderbilt University