

Software Factories

Assembling Applications with Patterns, Models, Frameworks and Tools

Jack Greenfield

Visual Studio Enterprise Frameworks & Tools
Microsoft Corporation
One Microsoft Way
Redmond, WA 98053
jackgr@microsoft.com

Keith Short

Visual Studio Enterprise Frameworks & Tools
Microsoft Corporation
One Microsoft Way
Redmond, WA 98053
keithsh@microsoft.com

ABSTRACT

The confluence of component based development, model driven development and software product lines forms an approach to application development based on the concept of software factories. This approach promises greater gains in productivity and predictability than those produced by incremental improvements to the current paradigm of object orientation, which have not kept pace with innovation in platform technology. Software factories promise to make application assembly more cost effective through systematic reuse, enabling the formation of supply chains and opening the door to mass customization.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques], D.2.11 [Software Architectures]: languages, patterns, domain-specific architectures, D.2.13 [Reusable Software] domain engineering.

General Terms: Design, Languages.

Keywords: Design Patterns, Domain-Specific Languages, Model-Driven Development, Software Product Lines, Software Factories.

1. INTRODUCTION

1.1 Industrializing Software Development

The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. However, growing pressure to reduce cost and time to market, and to improve software quality, may catalyze a transition to more automated methods. We look at how the software industry might be industrialized, and we describe technologies that might be used to support this vision. We suggest that the current software development paradigm, based on object orientation, may have reached the point of exhaustion, and we propose a model for its successor.

Some have suggested that software development cannot be industrialized because of its creative character. Others have suggested that significant progress can still be made on the foundation of object orientation, especially in light of the growing agility of development methods. We are quite sympathetic with

these points of view, and have seen that small teams of skilled developers can develop complex applications on time and on budget, with high levels of quality and customer satisfaction. However, we have also seen that such results are the exception, not the rule. Recognizing that software development is an inherently people-oriented discipline that cannot be reduced to purely mechanical and deterministic processes, however, we propose a people-oriented approach to industrialization, one that uses vocabularies that are closer to the problem domain, and that leaves more of the mechanical and deterministic aspects to development tools.

1.2 Lessons from Other Industries

Over the last ten years, the software industry has met the demands of an increasingly automated society by honing the skills of individual developers, just as artisans met the demands of an increasingly industrialized society in the early stages of the industrial revolution by honing the skills of individual craftsmen. Up to a point, this is an effective way to meet demand. Beyond that point, however, the means of production are quickly overwhelmed, since the capacity of an industry based on craftsmanship is limited by its methods and tools, and the by size of the skilled labor pool. A quick look at the state of software projects suggests that we are already struggling to meet demand using the current means of production.

According to the Standish Group [26], businesses in the United States spend more than \$250 billion annually on software development, with the cost of the average project ranging from \$430,000 to \$2,300,000, depending on the company size. Only 16% of these projects are completed on schedule and on budget. Another 31% are canceled, primarily due to quality problems, creating losses of about \$81 billion annually. Another 53% cost more than planned, exceeding their budgets by an average of 189%, creating losses of about \$59 billion annually. Projects that reach completion deliver an average of only 42% of the originally planned features.

These results are likely to be exacerbated by continuing platform technology innovation, which has outstripped the methods and tools used to develop software over the last ten years. In the business application market, for example, we can now integrate heterogeneous systems in different businesses located anywhere on the planet, for example, but we hand-stitch the applications deployed on this platform technology one at a time, treating each one as an individual, implementing coarse grained, domain specific concepts like stock trades and insurance claims using fine

grained, generic concepts like loops, strings and integers. Similar observations can be made in markets for mobile devices, media, smart appliances, embedded systems and medical informatics.

When faced with similar challenges many years ago, other industries moved from craftsmanship to industrialization by learning to customize and assemble standard components to produce similar but distinct products, by standardizing, integrating and automating their production processes, by developing extensible tools that could be configured to automate repetitive tasks, by developing product lines to realize economies of scale and scope, and by forming supply chains to distribute cost and risk across networks of highly specialized and interdependent suppliers. These changes enabled the cost effective production of a wider variety of products to satisfy a broader range of customer demands.

What will industrialization mean in the software industry? No one will know until it happens, of course, but we can make educated guesses by looking at how the software industry has evolved. We can also gain insights by looking at what industrialization has meant in other industries, and by comparing them with ours to see how our experience might be similar or different.

1.3 Economies of Scale and Scope

Others have drawn and debated such analogies between the software industry and industries that produce physical goods [4, 12, 13]. However, some of the discussion has involved an apples-to-oranges comparison between mass production in industries that produce physical goods, on the one hand, and the development of designs and one-off implementations in the software industry, on the other. There are two keys to clearing up the confusion.

- The first is to distinguish between economies of scale and scope. While both refer to the realization of cost reductions by producing multiple products jointly, rather than independently, they arise in different situations. Economies of scale arise in the production of multiple implementations of a single design, while economies of scope arise in the production of multiple designs and their initial implementations. Economies of scale arise in the production of software, as in the production of physical goods, when multiple copies of an initial implementation are produced mechanically from prototypes developed by engineers. Economies of scope arise when the same styles, patterns and processes are used to develop multiple related designs, and again when the same languages, libraries and tools are used to develop the initial implementations of those designs [14]. When used to drive mass production, these initial implementations are called prototypes. Accurate comparisons between the software industry and industries that produce physical goods should compare economies of scale in the mass production of copies in one industry with the same measure in the other, or they should compare economies of scope in the development of prototypes in one industry with the same measure in the other. They should not compare economies of scale in the mass production of copies in one industry with the development of prototypes in the other.
- The second is to distinguish between mass markets and custom markets. In mass markets, where the same product can be sold many times, economies of scale can be realized in software production, as in the production of physical

goods, by copying prototypes mechanically. In custom markets, each product is unique. Instead of being copied, the prototypes are delivered to customers. Here, software production is more like the construction of bridges and skyscrapers. In custom markets, economies of scope can be realized in software production, as in commercial construction, through systematic reuse, by using the same assets to develop multiple unique but similar products.

Of course, while the realization of economies of scale in the mechanical replication of software is well understood, the realization of economies of scope in the development of multiple unique but similar software products is not well understood. We therefore focus on the latter as an avenue for further progress toward the industrialization of software development.

1.4 Software Factories

Economies of scope can be realized in the context of a product family, whose members vary, while sharing many common features. A product family may contain either end products, such as portfolio management applications, or components, such as account management frameworks used by portfolio management and customer relationship management applications.

According to Parnas, a product family provides a context in which many problems common to family members can be solved collectively [23]. Building on software product line concepts, software factories exploit this context to provide family wide solutions, while managing variation among the family members [10]. Instead of waiting for serendipitous opportunities for ad hoc reuse to arise under arbitrary circumstances, a software factory systematically captures knowledge of how to produce the members of a specific product family, makes it available in the form of assets, like patterns, frameworks, models and tools, and then systematically applies those assets to automate the development of the family members, reducing cost and time to market, and improving product quality over one-off development.

Of course, there is a cost to developing a software factory in order to realize these benefits. In other words, software factories exhibit the classical cost-benefit trade-off seen in product lines [28]. While the benefit side of the equation cannot be increased through the production of many copies in custom markets, it can be increased through the production of many related but unique products or product variants, as documented by many case studies [10]. In addition, the cost side of the equation can be reduced by making product lines less expensive to build. In our view, the key to industrialization is enabling the cost effective construction and operation of software factories.

1.5 A Vision of Software Factories

Before looking at how software factories work, we offer the following vision of the future, when software factories are widely used to produce applications. Of course, in order to paint this vision, we necessarily gloss over many issues. This does not mean that we have ignored those issues. Indeed, we have written a book identifying and examining as many as we can, and suggesting resolution strategies, where possible [24]. In other words, while this is a vision, and therefore necessarily incomplete, we think it is close enough to reality to start guiding the way we think now. It implies significant changes, not only in methods and tools, but also in the economics of software development, as expertise comes from field organizations with problem domain knowledge, instead of from platform vendors.

1.5.1 *Development by Assembly*

Application developers will build about 30% of each application. The remaining 70% will be supplied by ready-built vertical and horizontal components. Most development will be component assembly, involving customization, adaptation, and extension. Instead of residing in large amounts of new code written in house from scratch, new functionality will be distributed across many ready-built and built-to-order components provided under contract by many suppliers, each of whom will write small amounts of new code from scratch. Standard components will be commoditized, and custom component suppliers will become ubiquitous, specializing in a wide variety of domains. Software that would be prohibitively expensive to build by current methods will become readily available.

1.5.2 *Software Supply Chains*

To feed the demand for components created by software factories, supply chains will emerge, creating standard product types with standard specification formats that help consumers and suppliers negotiate requirements, standard architectures and implementation technologies that let third parties assemble independently developed components, standard packaging formats that make components easy to consume, standard tools that can be reconfigured for product specific feature variations, and standard development practices. One of the keys to making this work will be the use of standard architectures that reduce component mismatch and simplify the management of supplier relationships by describing the contexts in which components operate. Architectural alignment has been a challenge to supply chain pioneers in the automotive and telecommunications industries.

1.5.3 *Relationship Management*

Requirements capture, analysis and negotiation will become critical elements of customer relationship management. Service level agreements documenting the expectations of consumers and suppliers will govern transactions. Following product delivery and acceptance, repairs and assistance will be provided on a warranty basis. In most cases, consumers will lease components from suppliers, allowing them to receive patches and upgrades systematically. Dynamic patch and upgrade mechanisms will become ubiquitous and much less intrusive. Tools that manage the configurations of deployed products will become critical parts of the platform. Data generated from customer and supplier interactions will be used to improve service levels, to optimize production and delivery, and to plan future product offerings.

1.5.4 *Domain Specific Assets*

At every step, including final assembly, developers will use tools configured for the purpose at hand. Tools for banking and health care application development, for example, will be readily available. These tools will use powerful abstractions and appropriate best practices encoded as languages, patterns and frameworks for specific domains. Application developers will no longer hand craft large amounts of code in general purpose languages. Instead, they will build variants of existing products, customized to satisfy unique requirements, writing small amounts of code in domain-specific languages to complete frameworks. Imagine tools that look like user interface builders for assembling web services and business logic. These languages, frameworks, patterns and tools will be inexpensive to build, enabling organizations with domain expertise to encapsulate their

knowledge as reusable assets. Of course, we are not suggesting that no one will write code by hand any more. When this vision is realized, product line developers will build the languages, frameworks and tools used by application developers, in much the same way that operating system developers build device drivers and other system software components used by application developers today. The mix of product line and product developers will be much more evenly balanced, however, than the mix of operating system and application developers today.

1.5.5 *Mass Customization*

Some industries, such as the web based PC business, produce product variants on demand cheaply and quickly for individual customers today. While we will not see mass customization in software products for some time, the broad adoption of software factories will eventually make it possible. Where this occurs in other industries today, business processes are highly automated and highly integrated. A value chain that integrates processes like customer relationship management, demand management, product definition, product design, product assembly and supply chain management is a fundamental prerequisite for mass customization. When software suppliers achieve the level of value chain integration found in other industries, mass customization will dramatically change the economics of software development. Picture a buyer ordering a customized financial services application from a web site, the same way that they order a customized desktop machine or server today, with the same level of confidence, and similar delivery time.

1.5.6 *Organizational Change*

Clearly, everyone with a stake in application development will be affected by the broad adoption of software factories. Software developers of all kinds will change their focus and think more about assembly than about writing new code. New development methodologies will arise, more in tune with the principles of manufacturing. Development organizations will develop the skills, incentives and processes required to exploit these changes. Packaged software vendors will restructure their products into componentized product families developed and maintained using software product lines. Business users will exploit the higher levels of services offered by their IT organizations.

1.6 **Developing Software Factories**

The current innovation curve could be described as the era of object orientation. Many commentators have observed that it has reached a plateau, and that new technologies are needed to create the next quantum leap forward in application development. According to Kuhn, new paradigms are catalyzed by the exhaustion of their predecessors, correcting mistakes and applying existing knowledge in new ways [22]. Building on domain engineering, software product line technology is catalyzing new developments in two object oriented technologies, model-driven development and component-based development, providing a medium for correcting mistakes and applying existing knowledge in new ways. These developments represent critical innovations on three axes, along which we can plot the transition to the next innovation curve, as illustrated in Figure 1.

The figure sets the course for the remainder of this paper. In the next three sections, we will look at each dimension in turn, and call out the maturity of techniques that support each dimension. In

the last section, we'll give a concrete example of software factories in action.

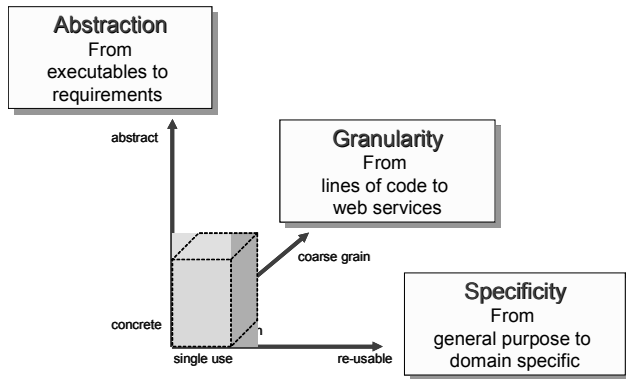


Figure 1: Three Axes of Critical Innovation

2. ABSTRACTION

Abstraction hides characteristics of a subject that are not relevant for some purposes, while emphasizing others, and defines a contract that separates the concerns of various stakeholders [7]. The abstraction axis ranges from abstract models that specify product features, to source code and other concrete artifacts used to produce executable implementations. Like refactoring and adaptive design, raising the level of abstraction reduces the complexity and brittleness of software through encapsulation. By hiding implementations, it keeps complexity from growing as features are composed, and prevents changes from propagating to clients. The higher we move along the axis, the more powerful the abstractions, but also the narrower their scope of application. Powerful abstractions that encapsulate large amounts of low level code tend to address highly specialized domains.

Since the start of the current innovation curve, there has been a trend toward increasing abstraction in platform technology, from distributed computing to message oriented middleware, to distributed object and component middleware, and recently to asynchronous, loosely coupled web services. A similar trend in application development technology is reflected by component architectures, which encapsulate deployment concerns, and by patterns, which describe configurations of existing technologies that represent known solutions to recurring problems in specific domains. While these innovations have advanced the state of the art by giving developers new vocabularies for solving problems [17], they have stopped short of giving them formal languages to express those vocabularies. We are now seeing the extension of the trend in this direction, with the advent of byte code based languages and specialized languages based on XML, such as languages for process specification and execution. In fact, this kind of progression has characterized the evolution of the industry, as noted by Smith and Stotts.

The history of programming is an exercise in hierarchical abstraction. In each generation, language designers produce explicit constructs for conceptual lessons learned in the previous generation, and then architects use [the new constructs] to build more complex and powerful abstractions. [25]

2.1 Model-Driven Development

One of the key themes of object orientation was Object Oriented Analysis and Design (OOA&D) [3]. OOA&D proposed a way to decompose functional requirements and map them onto object oriented implementation technologies. One problem with this approach was that it incorrectly assumed that the structure of the solution would match the structure of the problem [11]. Another problem with OOA&D, was that it promoted a methodical, top down approach to development. These problems have been addressed through a series of adaptations, however, and the basic principles of OOA&D remain the foundation of modern application development practices.

Since it combined contributions from several OOA&D methods, the Unified Modeling Language (UML) became a rallying point for a model-based approach to application development. But despite a large number of books about the UML and a large number of tools claiming to support the UML, the language has had little real impact on application development. Its primary use has been to produce visual representations of classes and the relationships between them. Constrained by weak extensibility mechanisms, the UML is tightly bound to the programming language concepts in vogue at the time of its creation. Its semantic ambiguity and poor organization have prompted a redesign, with contenders for a major revision vying to define what appears to be an even larger and more complex language that is just as ambiguous and as poorly organized as its predecessor. Even with the proposed revisions, the UML falls far short of providing the highly focused, domain-specific modeling capabilities required by the next wave of application development technology. Its primary contribution was the idea of an extensible modeling language with an underlying metamodel.

While the documentation of abstract concepts has some value, the real opportunity for innovation lies not in visualizing the information captured by models, but in processing that information to automate development tasks. This is the thrust of model-driven development (MDD). Despite its ties to OOA&D, MDD embodies ideas and technologies that predate object orientation. It seeks to capture developer intent, usually expressed informally, if at all, in prose or ad hoc diagrams, as formal specifications that describe application features in abstract terms using modeling languages, and to automate the implementation of that intent, usually by compiling the specifications to produce executables. This is valuable because the features are difficult to see in low level implementation artifacts, such as source code, and difficult to develop, maintain and enhance consistently, due to the number of independent elements that must be synchronized. The promise of MDD is that platform specific compilers will produce implementations for multiple platforms from a single specification, letting users retain investments in models describing application features as the platform technology changes beneath them. Of course, this is the same promise offered by byte code languages, such as C# and Java, and by generations of earlier languages that raised the level of abstraction above their predecessors.

With some notable exceptions, early attempts at MDD, represented by the CASE tools of the eighties, failed miserably. Most of these tools did not take advantage of platform-specific features, and produced naïve, inefficient, least common denominator code. The upfront costs of adopting the modeling techniques they required were prohibitively large. Added to this was the risk inherent in spending a substantial portion of the

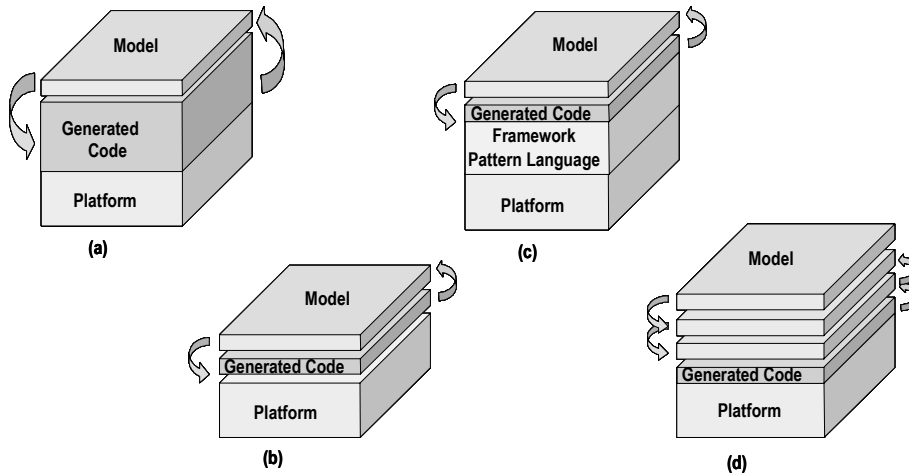


Figure 2: Models and Application Code

project's budget building models with the promise of code appearing only in later stages. This required enormous confidence in the tools and in the longevity of their vendors. Also, concepts like round-trip engineering, where a model could be synchronized with changes made independently to the code it had generated from them, were overwhelmingly complex. Another criticism was that many CASE tools imposed a methodical top-down process. This was the ultimate kiss of death, since rapid iteration of partial solutions has proven to be critically important to the success of application development, as so clearly demonstrated by agile development methods [2]. These problems can be overcome using language technologies that allow higher fidelity in specifying application features, and using more transformation techniques that give developers much greater control over generated code.

2.2 Critical Innovations in Abstraction

We have seen that CASE tools began to exploit the benefits of higher levels of abstraction, but were unable to manage the inherent complexities. Typically they selected a useful set of abstractions, but were naïve about the mapping to the executable platform, and tended to fill the gap between abstractions and executable platform by generating layers of application code. Even good products that successfully managed to keep generated code synchronized with the models, tended to overwhelm the developer with the resulting complexity, and the models fell into disuse. See Figure 2(a). A good example are the generic class modeling tools that have their own vocabulary and type system, and which require complex mapping and code generation to the languages they attempt to model. Usually after one trip through the generation process, the task of synchronization and re-generation becomes too troublesome and the model is abandoned.

Remember that the idea is to simplify the use of complex concepts by hiding unnecessary details, while providing a holistic view that exposes the relationships and dependencies among those concepts, where otherwise such these might be difficult to detect. The key idea is to keep the model – the set of useful simplifying abstractions – as close as possible to the underlying framework in which the concepts lie. For example, consider a graphical class modeling. This tool surfaces these class definition concepts with high fidelity to their original type system and vocabulary,

allowing the developer to manipulate parts of his program visually as he would in a source code editor, but the tool adds value to the developers job because it surfaces relationships and dependencies between classes that are not easy to see in the source code alone. See Figure 2(b).

2.3 Domain Specific Languages

If the set of simplifying abstractions requires too much of a bridge to implementation, the answer is to raise the level of abstraction of the underlying execution platform. There are two ways this can be done. The first is by providing a software framework that specifically addresses a well-defined, narrow problem domain,

and using the abstractions in a model to define how the variability points in the framework must be filled as in Figure 2(c). The model has become focused and specific to this domain – it can be expressed in a *domain specific language*, or DSL, which explicitly describes concepts the new framework offers. In contrast, attempts to capture details of the specific domain in a general purpose modeling language such as UML yields a lower fidelity description. Of course, DSLs may have either graphical or textual concrete syntaxes, whatever suits the developer best.

Let's take the example of user interface form design in Microsoft Windows. When Windows was first introduced, only highly skilled programmers were capable of building working Windows applications. When Microsoft Visual Basic was introduced, the Form and Control abstractions allowed huge numbers of less skilled developers to easily produce working Windows applications. Forms and Controls were highly effective abstractions for the domain of graphical user interface manipulation. Today, in the Microsoft .Net world, these same abstractions are implemented by the .Net framework that underlies the Form designer. Few developers bother to look at the small pieces of code generated to fill the variability points in the framework. We call this approach, where we generate minimal code to fill variability points in a domain specific framework from a domain specific model, *framework completion*, to contrast it with brute force code generation.

Alternatively, we can use a set of related patterns, called a pattern language, to implement the abstractions, instead of a framework. The pattern language for business application development with J2EE published by Alur, Crupi and Malks [1], for example, defines a collection of related patterns, and shows how and when they can be combined to solve many different kinds of problems. We can tool this kind of pattern language, giving the developer a way to apply the patterns, and then evaluating them automatically when applied. Of course, since frameworks embody patterns [20], we may be able to use a pattern language and a framework together, using the patterns to guide the assembly of components supplied by the framework. Note that we can now automate framework based development using either DSLs or pattern languages. The difference is essentially a trade-off between

Domain Specific Languages	Business	Information	Application	Technology
Conceptual	<ul style="list-style-type: none"> ▪ Use cases and scenarios ▪ Business Goals and Objectives 	<ul style="list-style-type: none"> ▪ Business Entities and Relationships 	<ul style="list-style-type: none"> ▪ Business Processes ▪ Service factoring 	<ul style="list-style-type: none"> ▪ Service distribution ▪ "Abilities" strategy
Logical	<ul style="list-style-type: none"> ▪ Workflow models ▪ Role Definitions 	<ul style="list-style-type: none"> ▪ Message Schemas and document specifications 	<ul style="list-style-type: none"> ▪ Service Interactions ▪ Service definitions ▪ Object models 	<ul style="list-style-type: none"> ▪ Logical Server types ▪ Service Mappings
Implementation	<ul style="list-style-type: none"> ▪ Process specification 	<ul style="list-style-type: none"> ▪ DB schemas ▪ Data access strategy 	<ul style="list-style-type: none"> ▪ Detailed design ▪ Technology dependent design 	<ul style="list-style-type: none"> ▪ Physical Servers ▪ Software Installed ▪ Network layout

Figure 3: A Layered Grid for Categorizing Models

complexity and control. Since the pattern language lets the implementations of the abstractions show through, the developer has more control over their implementation, but must also assume more responsibility. By contrast, a DSL hides the implementations of the abstractions, but gives the developer less control over their implementations.

If it's not possible to build a software framework that can provide a natural platform for implementing a useful DSL, it may be possible to define another layer of simplifying abstractions into which the first set may be mapped. This second set of abstractions may prove easier to implement than the first, leading to the notion of *progressive transformations* between models. The abstractions are transformed into executables by a series of steps as in Figure 2(d). When models stack one upon another in this way, it becomes useful to categorize and summarize the models in an orderly fashion, and to study carefully the relationships between them.

2.4 Relationships Between Metamodels

One common way to do this is to use a grid as in Figure 3. The columns of the grid represent concerns (data, activity, location, people, etc.), while the rows represent levels of abstraction. Each cell represents a *viewpoint* from which we can specify software. Typically, for a given family of applications, a path through the grid represents a sequence of modeling deliverables and transformations to properly address functional and non-functional requirements in the course of building and deploying an application.

This modeling grid is not in itself an innovation. What is novel is applying the grid to a product family, defining DSLs for each cell, and mappings between and within the cells that support fully or partially automatic transformations. As we have seen, we must use DSLs, not general purpose modeling languages designed for documentation, in order to provide this kind of automation. A grid like the one in Figure 3 can be generalized as a graph of

viewpoints for a product family, and tools can be developed inexpensively to support the editing and transformation of the associated DSLs. We call this graph a *software schema*, because it describes the set of specifications that must be developed to produce a software product. A software schema for a product family, the processes for capturing and using the information it describes, and the tools used to automate that process collectively form a *software template*. A software template can be loaded into an extensible tool, such as an Interactive Development Environment (IDE), to produce a specific type of product, in much the same way as a document template can be loaded into an extensible document editor, such as Word or Excel, to produce a specific type of document. An IDE configured with a software

template for a product family becomes a factory for producing members of the family. This is what we call a *software factory*. Later, we will see how software factories can be integrated to form automated supply chains.

Given appropriate DSLs and transformations, we can drive from requirements to executables using framework completion and progressive refinement, keeping related models synchronized. For example, we can produce a logical data model, and then an optimized database schema from a business entity model and usage statistics from a business process model. We can also leverage constraints that relate neighboring cells. For example, information known about the deployment environment (e.g., the available protocols and system services) can be used to constrain designs of service interactions (e.g., by limiting them to the protocols and system services available within the deployment environment). This helps to ensure that the implementations of the service interactions will deploy correctly to the designated deployment environment.

2.5 Transformations

Transformations can be characterized as either horizontal, vertical or oblique [14]. Vertical transformations are mostly refinement transformations that map models based on a more abstract DSL to models based on a more concrete one, or to code based on a general purpose programming language. For example, a transformation from a model that describes a business process to models that describe the collaborating web services that implement the business process is a vertical transformation.

Horizontal transformations may be either refactoring [16] or delocalizing transformations [14]. Refactoring transformations reorganize a specification to improve its design without changing its meaning. Refactoring can be applied to both tree and graph based languages.

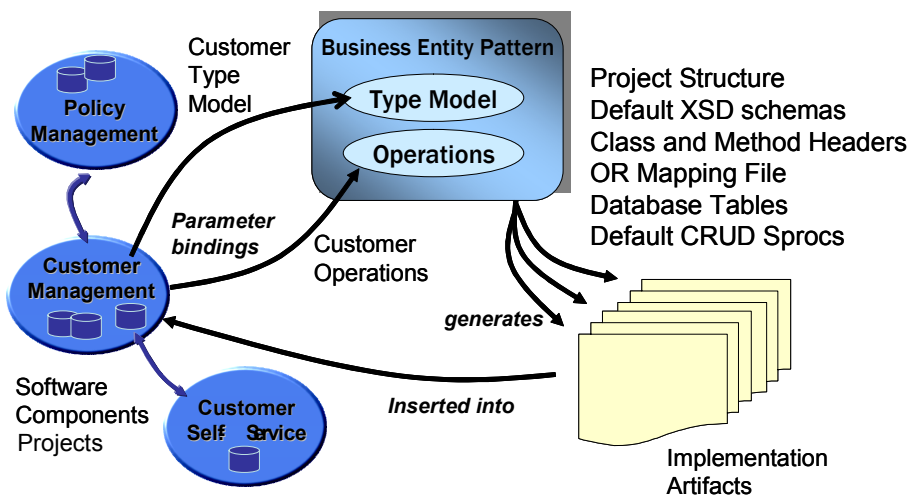


Figure 4: Applying a Business Entity Pattern

Delocalizing transformations can be used to optimize an implementation, or to compose parts of an implementation that are specified independently. Often, the implementation of an operational requirement, such as security or logging, must be distributed across many functional modules, making maintenance difficult. The implementation of the operational requirement is therefore tangled with the implementations of the functional requirements. Delocalizing transformations allow the operational requirement to be defined once and automatically woven into the functional modules during compilation. The operational requirement is called an *aspect*, and the process of weaving it into the functional modules is called *aspect weaving* [21]. Like refactoring, aspect weaving can be applied to both tree and graph based languages. At any layer of the grid, aspects can be modeled separately, and woven into the functional modules automatically when mapping to lower level models, by an aspect weaver. Separating aspect models from the functional modules makes maintenance easier, since it lets the developer focus on one problem at a time.

2.6 Using Patterns in Transformations

Transformations are inherently parameterized, and operate by binding objects in source models as parameter values, and creating or modifying objects in target models. We can think of a transformation as mapping a pattern of objects in a source model to a pattern of objects in a target model. It encodes best practices for improving or implementing models. Using patterns to describe mappings has led to new approaches to model representation, transformation and constraint. Of course, not all patterns can be applied automatically, because in many cases, the mapping between the models cannot be fully defined in advance. In these cases, the patterns must be applied manually by a developer, often with support from a development tool. Once they have been applied, however, they can generally be evaluated automatically, as described earlier.

Figure 4 shows a Business Entity pattern, a template for an abstraction that lets an application designer think in terms of data bearing persistent objects. This is a common abstraction for components used in business applications. It can be implemented

by writing a large amount of data access code, a moderate amount of object-relational mapping code, or a small amount of component persistence code, using a data access framework like Java Database Connectivity (JDBC), an object-relational mapping framework like TopLink, or a component persistence framework like Enterprise JavaBeans, respectively.

The pattern has been applied to the Customer Type Model in the Customer Management component. Its parameters are bound at the time of application. Its Type Model parameter is bound to the Customer Type Model, and its Operations parameter is bound to operations on the Customer Management component. After its parameters have been bound, the pattern can be evaluated to generate implementation artifacts that manage the persistence of Customer entities.

2.7 Some Examples of DSLs

If we revisit the grid from Figure 3, and zoom in to the bottom right hand corner, as in Figure 5, we can now look at the viewpoints and the relationships between them in more detail. In the figure, rectangles represent viewpoints, dashed lines represent refinement transformations and solid lines represent constraints. We now know that each viewpoint contains more than just DSLs. It also contains:

- refactoring patterns that can improve models based on the viewpoint,
- aspect definitions that are applicable to models based on the viewpoint,
- development processes used to produce models based on the viewpoint,
- definitions of constraints supplied by models based on neighboring viewpoint,
- frameworks that support the implementation of models based on the viewpoint,
- mappings that support transformations within and between models based on the viewpoint or neighboring viewpoint.

The figure illustrates the following:

- The Business Entity DSL defines the business entity abstraction, which describes efficient, message driven, loosely coupled data services that map onto an object-relational framework. Examples of business entities include Customer and Order.
- The Business Process DSL defines the business activity, role and dependency abstractions, and a taxonomy of process patterns that can be used to compose them, forming business process specifications. An example of a business process is Enter Negotiated Order, which might use three process patterns: one for a User Interface Process to build a shopping cart, one for a sequential process to submit the order and perform

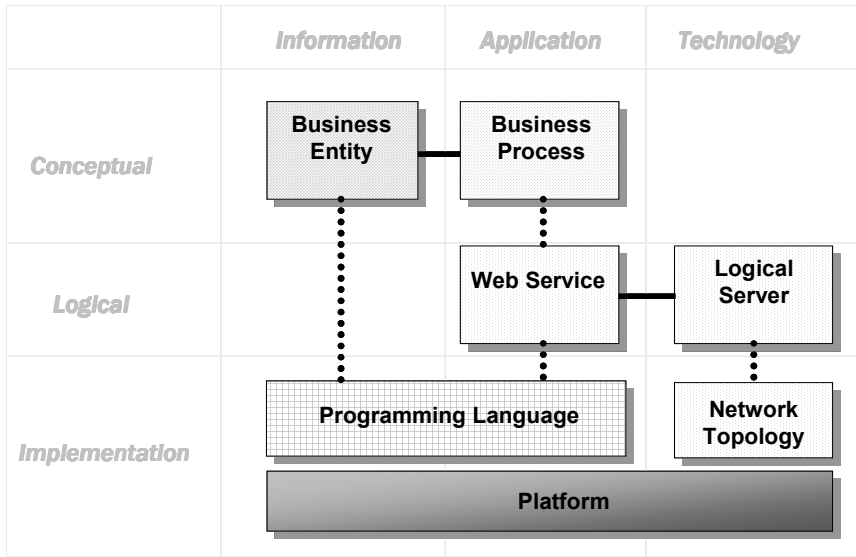


Figure 5: A Simple Software Schema

credit checks, and one for a rule-driven process to calculate the discount.

- These two DSLs map onto a Web Service DSL that describes collaborating web services in a service-oriented application architecture. The Web Service DSL is used to describe how the business entities and processes are implemented as web services, how the messages they exchange are defined and what protocols are used to support their interactions, using abstractions that hide the underlying details of the web service implementations.
- A DSL for describing virtual datacenter configurations in terms of logical servers that will be deployment targets for the web services described using the Web Service DSL, along with the software they have installed, and their configuration settings.
- Information from one model can be used to develop another. Examples are the interactions between business entities and processes, and between web services and logical servers. This last one is particularly interesting because it can be used to design for deployment. Feeding knowledge of the deployment infrastructure into web service designs constrains those designs to prevent deployment problems. Similarly, working this in reverse, if a design is to be deployed on a given logical server type, then we can validate that the server on which it will be deployed is of the correct type, that it has the right software installed, and that it is configured correctly.
- Mappings drive transformations between models at design time. For example, we use a transformation to map the model of web services to an implementation on the target platform, probably in the form of classes that complete a framework, such as ASP.NET.

3. GRANULARITY

Granularity is a measure of the size of the software constructs used as the vehicles of abstraction. Since the beginning of the era of object orientation, granularity has slowly increased, from fine grained language classes that represented abstractions like hash

tables and exceptions, to medium grained components that represented user interface controls, to coarse grained components that represented business entities and activity, and now to even more coarse grained services that typically represent large subsystems, such as billing or credit authorization. Increasing granularity can improve the reusability of abstractions because a coarser grained construct encapsulates more functionality than a finer grained one, has fewer external dependencies, and forms a more independent unit of specification, design, implementation, deployment, testing, execution, management, maintenance and enhancement. However, standardized mechanisms for describing component behavior beyond simple interface descriptors, such as WSDL for web services, have been slow to materialize. These mechanisms are necessary to realize the reuse potential of coarse grained pre-built components or built-to-order components and services for application assembly, because the services required and offered by these large

constructs, and their valid sequences of interaction, are often too complex to understand by experimentation.

3.1 Component Based Development

Component-based development (CBD) arose in the late nineties [15, 6]. CBD is an attempt to facilitate the independent deployment of coarse grained constructs using encapsulation to minimize the dependencies exposed by the objects they contained. Although they appeared at the height of interest in the UML, the principles of CBD are weakly supported by the UML. Perhaps the most serious problem is that the UML defines too many different and incompatible ways to describe abstractions, without defining enough semantics to make any of them usable for actual development. While it does a passable job of supporting language class specification, it does even worse in its attempts at describing the packaging of those classes into components. Its handling of component composition is also weak.

Designers want to express relationships between larger units of design (such as *business components* [HS00], web services, subsystems, etc.), since this can lead to greater reuse of pre-built and commercial off-the-shelf components, and to greater component outsourcing by formalizing contracts that describe component behavior. Larger units of design must compose smaller units without loss of rigor, and interactions between them must allow the same level of analysis as interactions between smaller units. For these reasons, CBD techniques focus on component composition and decomposition, and the challenges of partitioning functionality among interacting components. CBD surfaces two important ideas.

First, building on the established concept of an interface, CBD asserted that there should be a hard distinction between the specification of a component and its implementation. There are several mechanisms in the UML that attempt to address this issue. In addition to creating confusion by offering multiple ways to solve the same problem, these mechanisms are not used consistently within the UML. A component specification should be the subject of a model that describes the behavior of the

component, including its operations, parameters, and pre- and post-conditions, in terms of a model of relationships between specification types. No hint of any underlying implementation should surface through this abstract specification, and yet the specification should be rigorous enough to permit tool based composition, analysis and meaningful search. Such a specification serves as a contract to consumers of the component [18, 6]. With this discipline in place, the stage is set for a proper treatment of reuse, and for contracted-out component provisioning in software supply chains.

Second, CBD asserted that the structure and behavior of an application could be formalized and analyzed in terms of collaborations among components using only the component specifications, independent of any potential implementation. A collaboration describes roles played by component specifications in a sequence of interactions. By grouping collaborations, a complete picture of the design of a set of interacting components can be produced without requiring any information about their underlying component implementations. Collaborations provide a way to discover how functionality should be assigned to a specification, and thereby prescribe the behavior required of the implementation. Moreover, collaborations can be parameterized with variability points and systematically reused as design templates. Implementations of component-based designs can be generated in part and sometimes in whole by progressively refining compositions of collaborations using model-driven development techniques.

3.2 Critical Innovations in Granularity

In many ways the CBD development techniques from the mid-nineties were ahead of their time. While most of the techniques for component specification and composition were mature, the underlying execution platform technology was not mature enough to support applications based on large-scale built-to-order or ready-built components. In 1997, competing platform technologies included the OMG's CORBA, J2EE and COM technology from Microsoft.

Although each of these platform technologies succeeded in other ways, neither was able to support a component marketplace, or architectures where distributed, large granularity components could interact in a secure, efficient manner at scales necessary to support the new application types demanded by business.

The emerging web service technology succeeds where they failed. Protocols and platform extensions based on XML and SOAP, are becoming available for sophisticated web-delivered applications. They offer key infrastructure features, such as transactions, security, asynchronous messaging, routing and reliable, guaranteed-delivery messaging. With broad industry agreement through bodies such as the W3C [27] and WS-I [29], interoperability between proprietary component implementation platforms is being designed in from the beginning.

Web service technology uses the Web Service Description Language (WSDL) to define web services. A WSDL file is an XML file that defines a web service by specifying how it is invoked and what it returns, without describing its implementation. It can be advertised in a catalog and used by tools to generate adapters or client side code. As such, a WSDL file is the web service equivalent of a component specification, and can be generated by tools using CBD techniques for component definition, composition and interaction.

However, the WSDL specification is missing one critical concept that must be present to allow large scale composition of applications from web services. It is not enough to understand how to invoke a web service component. For realistic applications, you also need to know how to perform a sequence of interactions. A web service component specification must be explicit about the expected message order, and what happens when unexpected conditions arise. The protocol information that governs its interaction must be made explicit in a *contract* that should form part of the component specification.

Given an adequate definition of a contract, collaborations between web service components can be specified and composed. In many cases it will be possible to assemble applications using a process engine, such as Microsoft's BizTalk Server to declaratively define and manage the sequence of messages interchanged by the application components. This kind of assembly is called orchestration.

4. SPECIFICITY

4.1 Specificity Concerns Scope of Reuse

The third dimension is *specificity*. Specificity defines the scope of an abstraction. To paraphrase Jackson, the value of an abstraction increases with its specificity to some problem domain [19]. More specific abstractions can be used in fewer products (i.e., by members of a smaller product family), but contribute more to their development. More general abstractions can be used in more products (i.e., by members of a larger product family), but contribute less to their development. Higher levels of specificity allow more systematic reuse.

Of the three dimensions, specificity is perhaps the most important to software factories. Historically, the software industry has stayed at relatively low levels of specificity, compared with more mature industries, where specialized products and services are ubiquitous. The economics of software development reflect this tendency, rewarding generic products that can be used in many applications, but which contribute little to their development. The fact that applications are built primarily by hand shows that reusable components are not available for the vast majority of the features they require. Achieving the levels of reuse required to create significant economies of scope requires much higher levels of specificity, and a migration from generic products, such as tools and libraries for general purpose programming languages like Java and C#, to specialized products, such as tools and frameworks for vertical domains like Banking and Insurance. In order for this migration to occur, companies with domain knowledge must become producers of reusable components that support the development of applications in those domains. This might mean that barriers to component development will be lowered, so that domain knowledge holders can develop components in house, or it might mean that a component outsourcing industry will implement components from specifications supplied by the domain knowledge holders. Both solutions can be seen in other industries.

For example, in the consumer electronics industry, there are companies that build their own branded components, and there are companies that build components branded by other companies on a contract basis. Of course, this will require the development of new composition mechanisms, since software component assembly requires much higher levels of adaptation than the assembly of physical components, the standardization of

specification formats, since software specification is currently quite informal, and much better customer and supplier relationship management, since software development involves much higher levels of iteration.

4.2 Software Product Lines

While CBD focuses on the rigorous partitioning and composition of components, and MDD provides a rigorous framework for abstraction and transformation, both methods are inherently focused on building one product at a time. What they lack is the realization that most applications are members of families [23, 14, 10]. Software product lines are the critical innovation on the specificity axis that capitalizes on the separation of commonality and variability in product families. Figure 6 describes the main steps and deliverables in product line development. Recall that a software product line produces a family of software products that are deliberately designed to take advantage of common features and known forms of variation. Product line developers build *production assets* used by the product developers to produce family members. These include *implementation assets*, such as architecture and components, used to implement the family members, and *process assets*, such as a process, which describes the use of the implementation assets, and tools, which automate parts of the process. A key step in developing the production assets is to produce one or more *domain models* that describe the common features of problems in the domains addressed by the product line, and the ways in which they can vary. These models become detailed descriptions of the problem domains. They collectively define the scope of the product line, and can be used to qualify prospective family members.

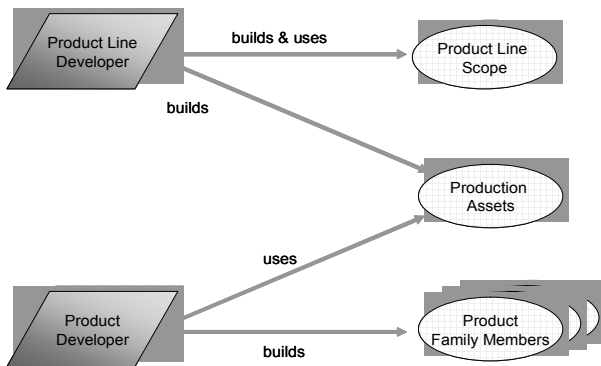


Figure 6: Overview of a Software Product Line (adapted from [28])

5. AN EXAMPLE OF SOFTWARE FACTORIES

5.1 Model-driven Product Lines

We can now define a *software factory* as a model-driven product line – a product line automated by metadata captured by models using domain specific modeling languages. We said earlier that software factories generalize the modeling grid for a product family, defining a graph of *viewpoints* called a *software schema*, which describes the information required to produce a family member. Since each viewpoint is supported by DSLs, we develop

DSL based tools for editing the models, and for translating them either into executables, or into specifications at lower levels of abstraction, such as general purpose programming language source code files, or models based on more concrete DSLs. We then define the process assets for the software factory in terms of this process. Finally, we collect these assets into a *software template* that loads into an Interactive Development Environment (IDE), such as Microsoft’s Visual Studio .NET. When configured in this way, the IDE becomes a software factory for the product family, as illustrated in Figure 7. In other words, for a software factory, the product line developers build production assets for a specific IDE, and deliver them as plug-ins to the IDE.

Using the software factory, the product developers can rapidly assemble family members. Recall that in a product line, the requirements model is used to specify a family member by identifying the feature variations that define it uniquely. In a software factory, selecting feature variations automatically or semi-automatically configures the production assets for the selected family member, including the project structure, imported subsystems and components, available viewpoints and patterns, and constraints. For example, imagine that when we select the content personalization feature for an online commerce application, the following things happen:

- A folder for the personalization subsystem is added to the project we’re using to build the application.
- The personalization subsystem is imported into the project.
- The viewpoint used to configure personalization is added to the schema for the application, causing the personalization configuration tool to appear on the menu.
- The Front Controller pattern is applied automatically in the transformation between the user interaction model and the web front end design model, and is made available in the designer where we model the web front end design, instead of the Page Controller pattern, so that the application will vector to different pages for different users, instead of showing the same content to all users.
- We are not allowed to create a class that derives from PageController in the folder for the personalization subsystem

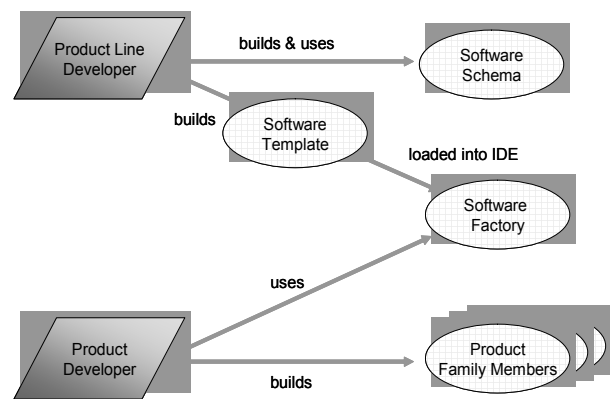


Figure 7: Overview of a Software Factory

Having configured the software factory appropriately, the product developers use it to build the family member. They build models

for the viewpoints defined by the software schema, starting with models near the top of the graph, and then working their way down, producing executables using framework completion and progressive refinement. At times, they may work bottom up, instead of top down, generating test harnesses for various pieces of the product, and testing the pieces as they work. When the software schema is completely populated, the process is complete.

5.2 Software Factory Example

Figure 5 illustrates a software schema for a family of software products, in this case business applications that can be specified as interacting business processes and business entities, and deployed as collaborating web services.

Imagine a bank that needs to build business applications for rapidly changing financial instruments. In Step 3 of Figure 8, the product line developers at the bank, armed with Software Factory B, build production assets for the schema shown in Figure 5. These assets comprise a software template that can be loaded into another instance of the same IDE. Configured in this way, the IDE is now Software Factory C. In Step 4 of Figure 8, the product developers at the bank now use this software factory to build business applications for rapidly changing financial instruments.

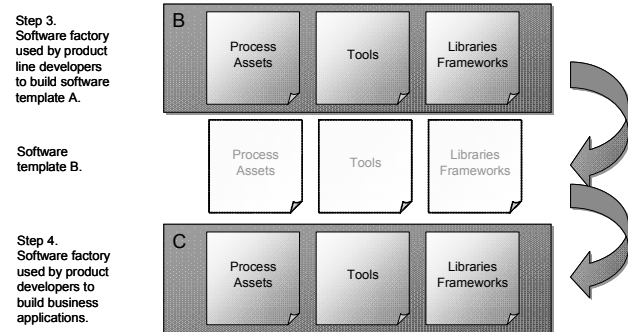


Figure 8: Building and Using a Software Factory

Of course, instead of fairly generic production assets for business applications, Software Factory B could have been used to build more specialized production assets for financial services applications, such as implementation assets like risk calculators and forecasting engines, and process assets like special project types and wizards for financial application development. Configured with these assets, the IDE would become a software factory for financial applications. It would be used again by the product developers at the bank to build business applications for rapidly changing financial instruments. This time, however, the DSLs, tools and frameworks at their disposal would be much more specific to the financial services domain. They would therefore be more productive using this software factory than they were using the previous one, since more of each application would be supplied by the frameworks and generated from the DSLs by the tools, leaving less work to be done by hand.

At this point, we should ask where the product line developers got Software Factory B, and why the examples above started out with Step 3. As it turns out, software factories can be used to produce other software factories. In Step 1 of Figure 9, an IDE is used to build a languages, frameworks and tools for building software factories. These assets comprise a software template that can be

loaded into another instance of the same IDE. Configured in this way, the IDE is now Software Factory A in Step 2. This software factory can be used to build software factories. We can think of the IDE as Software Factory A, although in practice, it is just an extensible IDE. This process, called *bootstrapping*, is standard practice in compiler development.

How do supply chains fit into the picture? From what we have said in the preceding paragraphs, we might conclude that software factories do not allow multiple suppliers to collaborate in the development of the product. On the contrary, they promote the formation of supply chains in two ways:

At any level, the schema can be partitioned vertically among multiple suppliers. The components provided by the suppliers are then assembled at the level above. Imagine, for example, that the risk calculators and forecasting engines used by the bank come from different suppliers, and are assembled by the developers at the bank.

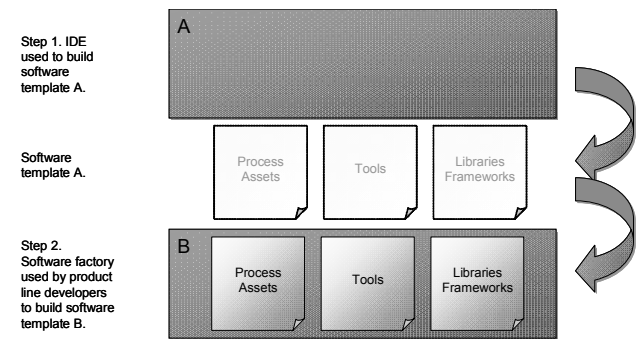


Figure 9: Building a Software Factory with a Software Factory

Within any vertical partition, the schema can also be partitioned horizontally among multiple suppliers. Imagine, for example, that instead of working for the bank, the product line developers who created Software Factory B work for an Independent Software Vendor. Instead of building software factories for in house developers, they build them for downstream consumers. They might then, in turn, use less specialized software factories, such as the original Software Factory B that produced components for building business applications.

Of course, the two types of partition can appear anywhere in the schema, and can be combined in arbitrary ways. They can also disappear and then reappear, as dictated by prevailing business conditions. In a mature industry, we would see many levels of suppliers contributing to the completed application, and constant churn in the supply chain, as suppliers enter the market, consolidate or leave the market at various points.

6. CONCLUSION

Software Factories are the convergence of key ideas in software product lines, component-based development and model-driven development. The innovation lies in integrating these ideas into a cohesive framework that supports new tools and new practices. By combining model-driven and component-based techniques with product line principles, Software Factories usher in a new application development model, where highly extensible

development tools are quickly and cheaply configured to create software factories for specific domains.

Realization of this vision is the goal of software factories. It will require us to rethink tools and methods, languages and frameworks. Of course, some parts of this vision may never be realized. However, it is commonly said that we frequently over-estimate what can be achieved in five years, and under-estimate what can be achieved in ten. That said, there is already substantial momentum toward the realization of the vision. It is our conviction that key elements will be realized. The evidence of their realization has already started to appear.

7. ACKNOWLEDGEMENTS

Special thanks to Krzysztof Czarnecki and Paul Clements, who reviewed earlier versions of this material and provided valuable comments.

Adapted from Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools, by Jack Greenfield and Keith Short, Copyright © 2003 by Jack Greenfield, Keith Short. All rights reserved. Reproduced here by permission of Wiley Publishing, Inc.

8. REFERENCES

- [1] D. Alur, J. Crupi and D. Malks. Core J2EE Patterns, Best Practices and Design Strategies. Sun Microsystems Press, 2001.
- [2] K. Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.
- [3] G. Booch. Object Oriented Analysis and Design With Applications. Second Edition. Addison-Wesley, 1994.
- [4] F. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. Computer Magazine, 1987.
- [5] C. Christensen. The Innovator's Dilemma, Harvard Business School Press, 1997.
- [6] J. Cheesman and J. Daniels. UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2000.
- [7] J. Cleaveland. Program Generators with XML and Java. Prentice Hall PTR, 2001.
- [8] A. Cockburn. Writing Effective Use Cases, Addison Wesley, 2000.
- [9] S. Cook. The UML Family: Profiles, Prefaces and Packages. Proceedings of UML2000, edited by A. Evans, S. Kent and B. Selic. 2000, Springer-Verlag LNCS.
- [10] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001.
- [11] J. Coplien. Multi-paradigm design. In Proceedings of the GCSE '99 (co-hosted with the STJA 99).
- [12] B. Cox. Planning the Software Industrial Revolution. IEEE Software Magazine, November 1990.
- [13] B. Cox. No Silver Bullet Revisted. American Programmer Journal, November 1995
- [14] K. Czarnecki and U. Eisenecker, Genererative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
- [15] D. D'Souza and A. Wills. Objects, Components And Frameworks With UML. Addison-Wesley, 1998.
- [16] M. Fowler. Refactoring: Improving The Design Of Existing Code. Addison-Wesley, 1999.
- [17] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. 1995, Addison-Wesley.
- [18] P. Herzum and O. Sims. Business Component Factory, A Comprehensive Overview of Component Based Development for the Enterprise. March 2000, John Wiley and Sons.
- [19] M. Jackson. Problem Frames: Analyzing and Structuring Software Development Problems. Addison Wesley, 2000.
- [20] R. Johnson. Documenting Frameworks Using Patterns. ACM SIGPLAN Notices, volume 27, number 10.
- [21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In Proceedings of the European Conference on Object-Oriented Programming, 2001.
- [22] T. Kuhn. The Structure Of Scientific Revolutions. The University Of Chicago Press, 1970.
- [23] D. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering, March 1976.
- [24] J. Greenfield and K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. John Wiley and Sons. 2004.
- [25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture, Volume 1. A System Of Patterns. John Wiley and Sons, 1996.
- [26] J. Smith and D. Stotts. Elemental Design Patterns - A Link Between Architecture and Object Semantics. Proceedings of OOPSLA 2002.
- [27] Standish Group, Chaos – A Recipe for Success, 1999, available online at http://www.standishgroup.com/sample_research/
- [28] <http://www.w3.org/>.
- [29] D.M. Weiss, C.T. Robert Lai. Software Product-Line Engineering, Addison-Wesley , 1999.
- [30] <http://www.ws-i.org/>.