Learn the discipline, pursue the art, and contribute ideas at www.ArchitectureJournal.net

Resources you can build on.

THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

Journal 9

Software Factories

Bare Naked Languages or What Not to Model

Domain-Specific Modeling

Measuring Success with Software Factories

A Foundation for the Pillars of Software Factories

A GSI's Perspective of Software Factories

The Perspective-Based Architecture Method

Microsoft®

Contents

THE ARCHITECTURE JOURNAL™
Input for Better Outcomes

Journal 9

Founder
Arvindra Sehmi
Microsoft Corporation

Editor-in-Chief
Simon Guest
Microsoft Corporation

Microsoft Editorial Board
Gianpaolo Carraro
John deVadoss
Neil Hutson
Eugenio Pace
Mauro Regio
Javed Sikander
Philip Teale
Jon Tobey

Publisher
Marty Collins
Microsoft Corporation

Online Editors
Beat Schwegler
Kevin Sangwell

Design, Print, and Distribution
Fawcette Technical Publications
Terrence O'Donnell, Managing Editor
Michael Hollister, VP of Art and Production
Gerry Guzman, Circulation Manager
Kathleen Sweeney Cygnarowicz, Production Manager

The information contained in *The Architecture Journal* ("*Journal*") is for information purposes only. The material in the *Journal* does not constitute the opinion of Microsoft or Microsoft's advice and you should not rely on any material in this *Journal* without seeking independent advice. Microsoft does not make any warranty or representation as to the accuracy or fitness for purpose of any material in this *Journal* and in no event does Microsoft accept liability of any description, including liability for negligence (except for personal injury or death), for any damages or losses (including, without limitation, loss of business, revenue, profits, or consequential loss) whatsoever resulting from use of this *Journal*. The *Journal* may contain technical inaccuracies and typographical errors. The *Journal* may be updated from time to time and may at times be out of date. Microsoft accepts no responsibility for keeping the information in this *Journal* up to date or liability for any failure to do so. This *Journal* contains material submitted and created by third parties. To the maximum extent permitted by applicable law, Microsoft excludes all liability for any illegality arising from or error, omission or inaccuracy in this *Journal* and Microsoft takes no responsibility for such third party material.

Foreword

Dear Architect,

I would like to be the first to welcome you to Issue 9 of *The Architecture Journal*, the theme of which is "software factories."

Henry Ford was one of the pioneers of building a *factory* that first used a production line in the early 20th century. Much of his work was notable because it led to higher production rates for workers and more inexpensive products. Stepping forward almost a hundred years, and for many of the same reasons, the term and application have now become part of daily vocabulary in our industry.

One of the books that first popularized the thinking in this area is *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* by Jack Greenfield, et al. (Wiley 2004). Building on the ideas outlined in this book, we are incredibly fortunate to have coauthors Jack Greenfield and Steve Cook lead off this issue of *The Architecture Journal*.

In his article, "Bare Naked Languages," Jack explains how to fit domain-specific languages (DSLs) into the software factories methodology and covers some common pitfalls to avoid when using this approach. As part of Jack's article, we are also introducing a new feature in this issue. Many readers have been asking for more information about the careers of well-known and respected software architects. As well as having Jack write our lead article, we had the chance to sit down with him and get the full details on his career, which are outlined in a new, "Profile of an Architect" section. I hope you enjoy his answers, and we look forward to suggestions about who you would like to see profiled in upcoming issues.

Steve Cook's article digs into the details of using DSLs to simplify larger problems into smaller problems, including the application in a software factory platform. Marcel de Vries continues the software factories theme, providing a look at the reporting and warehouse capabilities necessary to determine which aspects of product development need improvement.

Returning contributor to *The Architecture Journal*, Tom Fuller, looks at a foundation for the pillars of software factories to promote reusability and other strategic processes. Steve Eadie from EDS follows Tom's article with a view of the perspective of implementing software factories from a Global System Integrator (GSI). Steve offers some thoughts about using software factories through the lens of GSI, even with a development team that's located all around the world.

Finally, Lewis Curtis and George Cerbone outline their thoughts around a method for perspective-based architecture. This way of analyzing requirements is a technique that architects can use to "ask the right questions" for projects in which they're engaged.

In the spirit of Henry Ford's innovation, I hope this issue of *The Architecture Journal* helps you build software factories and production lines that enable you to offer software in any shape, size, and even painted in any color—well, as long as it's black of course!

Simon

Simon Guest

Bare Naked Languages or What Not to Model

by Jack Greenfield

Summary

Domain-specific language (DSL) technology was introduced at Microsoft as part of the software factories methodology. While DSLs are useful as stand-alone languages, placing them in the context of a software factory makes them more powerful and helps avoid some common pitfalls in applying the technology. This article explains how DSLs fit into the software factories methodology and how using the methodology can help DSL developers avoid some common pitfalls.

If you have been involved in the Visual Studio community over the last two years, you're aware of the excitement around domain-specific languages (DSLs) in the Visual Studio community (see Resources). DSLs are visual, powerful, and easy to use. More important, there are compelling examples of DSLs already in use or under development, including DSLs for defining work item types, object relational mappings, and service contracts.

Unfortunately, in all the excitement, we sometimes see people misapplying the technology. The problem is not that the motivation for using DSLs is poorly understood. On the contrary, the community seems to clearly understand the benefits they offer over general-purpose modeling languages like unified modeling language (UML), which is gratifying, since we have devoted a lot of article, podcast, panel, and keynote space to that topic over the last two years. Here is a quick recap.

A general-purpose modeling language is designed to support the development of models that serve primarily as documentation. Such a language can describe any domain, but only imprecisely. The imprecision is a consequence of using generic abstractions, such as class, activity, use case, or state. The use of such generic abstractions is what makes a general-purpose modeling language broadly applicable. In the case of UML the generic abstractions are defined using informal, natural language rather than formal or semiformal semantic, definition techniques, such as translation, execution, or denotation. The combination of generic abstractions and informal semantics prevent such a language from describing any particular domain precisely.

A DSL, by contrast, is designed to describe precisely a specific domain, such as a task, platform, or process. Instead of generic abstractions, it uses concepts taken directly from the target domain. The precision offered by such a language is further improved by defining its semantics using either translation or execution. This level of formality is possible because the language designer knows how models based on the language will be used for computation. Examples of the kinds of computations performed with DSL-based models include: generating code or other artifacts; creating relationships among model elements and code or other artifacts; validating relationships among model elements and code or other artifacts; configuring a run-time component, such as a Web server; and analyzing models and related code or other artifacts (for example, computing the impact of a given change).

At least that's the theory. The problem is that in practice people don't always seem to know when to use or not to use a DSL, what to model, or how to develop a set of related models that can be stitched together to support a development process. These problems can be avoided by building software factories, instead of bare naked languages, which are DSLs that are not defined in the context of software factories. A software factory usually supplies a variety of reusable assets, including DSLs, to support software development. More important, it helps developers build useful assets by placing them in the context of a product architecture and a development process for a specific type of deliverable.

When Not to Use DSLs

Let us look more closely at the most common pitfalls in applying DSL technology. We see people using DSLs to solve problems that are not well suited to modeling, such as problems that can be solved more easily with other tools, or problems in poorly understood or rapidly-evolving domains, where reusable patterns and practices have not yet emerged or stabilized. For example, we have seen DSLs for entity relationship mapping and business application development become obsolete while they were still being developed because the platform technology underneath them was still evolving.

We also see people who are confused about what to model. This confusion is usually manifested by scoping problems. The two most common scoping problems are using a single DSL to capture information about too many different kinds of things, or to capture too many different kinds of information about the same thing, and factoring concepts poorly among multiple DSLs. For example, we have seen one DSL used to describe the decomposition of systems into collaborating service-based components, and the messages exchanged by the components, and the payloads carried by the messages, and the structure of the component implementations, and the configuration of the components for deployment.

Industry experience with patterns and aspects has clearly demonstrated that tangling many different concerns in a single piece of code creates problems. This principle applies to models as well as to code. For example, I might want to change a message payload definition for a component, while someone else changes the component implementation structure using the DSL described previously. We will face a complex merge problem when we check in our changes because two different concerns have been tangled in a single model. The problem is that the model is poorly factored. A poorly factored model is like poorly factored code. It fails to separate concerns, making change difficult and error prone. We can improve poorly factored models the same way we improve poorly factored code, by refactoring to patterns that cleanly separate pieces of code or concepts that represent different concerns. A good clue that two concepts in a language represent different concerns is that they change at different rates or for different reasons.

As for stitching models together to support a development process, we see a lot of interest in the Visual Studio community, but not many working examples based on DSLs. Object-oriented analysis and design (OOA&D) popularized the idea of using a set of interrelated models to span gaps in the development process, especially the gap between requirements and design (see Resources). Unfortunately, it never quite lived up to the promise, since the models involved were merely documentation based on general-purpose modeling languages that only loosely described the requirements or the design.

Profile of an Architect–Jack Greenfield

Who are you, and where do you work?

I am Jack Greenfield. I work as an architect in the Enterprise Frameworks and Tools (EFT) group in the Visual Studio Team System organization, which is part of the server and tools business at Microsoft.

Can you tell us a little about your career at Microsoft?

When I joined EFT, the group was already building the set of tools that we now call the Distributed System Designers in Visual Studio 2005 under the leadership of Anthony Bloesch and Kate Hughes. Keith Short saw the tools as a ”down payment” on a vision of model-driven development (MDD), where organizations would use models as source artifacts for automating application development, not just as documentation.

EFT was a natural fit for the vision of MDD presented in my book, which was partially completed when I joined Microsoft. My vision was similar to Keith’s, in seeking to leverage MDD, but it brought a new twist into the picture by placing MDD in the context of software product line engineering, which emphasizes the development of reusable assets supporting a custom process for building a specific type of deliverable. Keith and I worked together to create a unified vision and to describe it in the book. The result was the methodology we now call software factories.

While we were writing the book we saw an opportunity to use the modeling framework used by the Distributed System Designers as a basis for domain-specific languages (DSLs), which constitute an important component of the methodology. We put together a business case for productizing the framework and for building tools to support it as the first step toward implementing software factories.

We took the proposal to Eric Rudder, supported by a demo, and asked for resources to build a team. We were able to show that we had a good foundation to build on. Eric agreed and provided the funding. We were fortunate to be able to hire an all-star team led by Steve Cook and Stuart Kent to do the work. The result is what we now call the DSL Tools, which were just released in the Visual Studio SDK. Steve and Stuart also joined us in writing the book, contributing chapters on language design and implementation.

During that time, I also led an effort to apply DSL technology to the tools being developed for the Microsoft Business Framework, working with a team led by Steve Anonsen, Lars Hammer, and Christian Heide-Damm. We used some of the ideas from the book about model references and model management on that project. I also did a lot of speaking, writing, and customer interviews to evangelize software factories. Several people helped us promote, refine, and implement the methodology. Michael Lehman, Mauro Regio, and Erik Gunvaldson, in particular, were instrumental. Mauro and I are writing a new book called *Software Factories Applied* that shows how to build factories using the prototypes he developed for HL7 and UMM as examples.

We are now building additional products to support the software factories vision. One of the most important steps enabling our current work was Mike Kropp becoming the product unit manager for EFT and integrating it with the patterns and practices group, another all-star cast led by Wojtek Kozaczynski and Tom Hollander. The merger of the two groups has put the wood behind the arrow to build the vision.

What kind of advice would you give to someone who wants to become an architect?

Good question. In general, you do not apply for a job as an architect; it just kind of happens to you, and I think it happens because of the way you think, the way you talk, and the kind of work you do. For me, an architect is someone who goes beyond the fine details of programming and thinks about the big picture.

I like a description that was offered by someone I met recently from the Netherlands: an architect mediates communication among people who think about end-user concerns, people who think about technology, and people who think about the business case. They see how the thing

(Profile continued on page 4)

Also, the relationships between the models were defined only informally, with the goal of helping humans perform the transformations between them by hand. Perhaps the most serious problem, however, is that the gap between requirements and design must be spanned in different ways for different types of applications. A lot of important detail was lost as people tried to apply a one-size-fits-all approach using generic models to complex problems.

Despite these shortcomings, the vision remains powerful, and we see many people attempting to realize it in a variety of ways. Most of them do it manually, some have developed DSL integrations, and some have tried model-driven architecture (MDA) from the Object Management Group (OMG). Of course, MDA never quite lived up to the promise, either, as has been explained in several podcasts and articles. I will recap briefly the reasons here to save you the effort of looking them up. You can then refer to them if you like for more detailed discussion (see Resources).

MDA emphasizes platform independence. In practice, platform independence is not an absolute quality of a model. It's a quality that a model can have to varying degrees. For example, an entity relationship model may not depend on the features of a specific version

(Profile continued from page 3)

can come together in much the same way that an architect in the construction trade sees how a building comes together to serve the needs of its users in an aesthetically pleasing way within the owner's budget and schedule using available building technologies.

At some point someone will say about you, "he or she is our architect," and that is how you will know you are functioning in that role. Over the last few years a lot of good material has been written about architecture and the role of the architect, and an industry consensus regarding best practices is emerging. In addition to development and organizational skills, an architect needs to know how to find and apply relevant research in product designs.

How do you keep up to date?

First, I spend a lot of time with folks inside Microsoft. I have a lot of interactions with other teams that help me see the bigger picture and where my work fits within it. I think being familiar with your company's products is one of the most important obligations of an architect. Of course, Microsoft is huge, and it is hard to get your head around even a small chunk of it, but the more you can learn the better.

From there, I tend to follow movements in industry and academia, such as the design patterns and agile development movements. I read a lot of books and articles by other folks who push the envelope, such as Martin Fowler and Rohan McAdam. On the academic side, I follow the Software Engineering Institute (SEI), especially the work being done by Linda Northrup, Len Bass, Paul Clements, and others on the practice of architecture, which contributed to the design of software factories. I also stay in touch with colleagues like Krzysztof Czarnecki at Waterloo and Don Batory at Austin. I also serve on conference program committees and peer review boards for journals, which give me opportunities to see a lot of good papers. I also do my share of speaking and writing to stay engaged in the community.

Name the most important person you have ever met and why?

Apart from sitting on Lyndon Johnson's knee as a five year old, the most important person I have ever met has to be Bill Gates. I have presented to Bill on a couple of occasions, sitting right across the table from him. On every occasion he has played the same kind of role, which was to ask a lot of questions and then to point us at related work taking place in other parts of Microsoft. I think this really demonstrates the point about knowing your company. Bill is always concerned about leveraging as much as possible of what we are doing around the company.

Another person I would put on my list is Steve Jobs. He is a charismatic and visionary leader. He creates a tremendous amount of excitement in people. It gave me the feeling of doing something world changing.

What is the one thing that you regret most in your career?

I have done more than my share of things that I would do differently in hindsight, but the one thing I regret most is not a single thing, but rather a trend or pattern that if I could I would correct from the start. In the past, I have often thought, "I can do it!" and have just plowed ahead. Instead, I should have stepped back and spent more time working with others as part of a team.

It is hard to overemphasize the value of being part of a team. No one has all the cards as an individual. There have been many times in my career when I have held nine of the ten cards I needed to win, but could not get the tenth card without a team. Unless you work in a way that brings out and synthesizes the contributions of other people, you will not have all the cards when you need them. Thinking you can do it alone is an affliction that plagues many technically gifted people. Look around some time and notice how many B students who work well with teams end up being more successful than A students who know most of the answers. It should be clear from the number of people I have named in this interview that the success of my work depends on the work of many other people and vice versa, and there are many people who have made key contributions to the software factories vision. Some of them are named in the acknowledgements in the first book, and a whole new set will be named in the second book.

What does Jack Greenfield's future look like?

I want to see software factories change the industry. This goal relates to the comment I just made about teams. To have industry-wide impact, software factories will have to be owned and developed by many people. Object orientation emerged in the late 70s and early 80s, but in the course of being scaled out to the masses in the 90s a lot was lost along the way. A lot of what we call object-oriented code today is not object oriented—it just happens to be written in an object-oriented language. I think the same thing is happening with service orientation.

The key to fixing these kinds of problems is changing software development from a craft to an engineering disciof a database management system from a specific vendor, but it still assumes the existence of a relational platform with certain capabilities. I have yet to see a model that is truly independent of all assumptions regarding the capabilities of the underlying computational platform.

MDA assumes that the world consists entirely of models. Of course, we know that it contains many other types of artifacts manipulated by many techniques and technologies other than modeling. A model-driven development methodology should embrace those other techniques and technologies and should explain how models integrate with those other types of artifacts.

MDA relies on a single general-purpose modeling language: UML. I have already explained the motivation for using DSLs rather than a general-purpose modeling language like UML. A model-driven development (MDD) methodology should use modeling languages that are powerful enough to support application development in the real world.

MDA assumes that only the same three kinds of models are needed regardless of what is being modeled. One is called a computation-independent model, one is called a platform-independent model, and one is called a platform-specific model. This assumption is really another manifestation of a generic approach. It does not make sense to define a specific set of models for a specific type of software if the modeling language is not capable of describing that type of software any differently than it would describe some other type of software. In practice, many different kinds of models are needed to describe any given piece of software, and the kinds of models needed are different for different types of software.

MDA focuses entirely on transformation. From the platform-independent model, we push a magic button that generates the platform-specific model. From there, we push another one to generate the code. Computer-aided software engineering (CASE) demonstrated in the late 1980s and early 1990s that there are no magic buttons. MDA is trying to go one better than CASE by relying on two of them. In practice, transformation is usually the last computation to be supported effectively between two models or between a model and code or other artifacts, simply because it requires so much knowledge about the two domains. Also, the engineering challenges of managing changes to one or both of the artifacts are daunting. Long before transformation can be supported, other forms of model-based computation are possible.

Best Practices for MDD

At this point, you may be wondering if there are some best practices you could follow to avoid these problems. Not only are there best practices, there is a methodology that embodies a set of integrated best practices for MDD. I am talking about the methodology we call software factories. If you have been involved in the Visual Studio community over the last two years, you are also aware of the excitement around software factories. If you're not familiar with software factories, their home page on the MSDN site is a good starting point for finding more information (see Resources). You may also have noticed that DSLs and software factories are often mentioned in the same breath, and you may be wondering how they are related. Let us look at the answer to that question.

pline. A lot of people have said it cannot be done, and for some time our industry has been in a phase of focusing on craftsmanship at the expense of developing and applying better engineering practices. Unfortunately, craftsmanship does not scale, and there is a tidal wave coming, driven by changes in the global economy, which will require approaches that do scale.

I think the key to scaling up and scaling out is to leverage experience, which is the goal of software factories. In the simplest terms a factory is experience codified and packaged in such a way that others can apply it. As much as I appreciate "following design smells," if you are still smelling your way through the fifth or sixth Web application something is wrong. You may smell your way through a part of it, through some new requirement or some new algorithm, but by and large most of the work we do is similar to work that has already been done, and we need to get much better at leveraging that experience. We are getting there slowly but surely. If I can help bring about transition, I will have achieved my professional goals.

Jack Greenfield's Résumé

Education

B.S. Physics, 1981; George Mason University; GPA, 3.15/4.00

Graduate Course Work

1988, George Mason University

Continuing Education

1991, Oregon Graduate Institute

Experience

Microsoft Corporation – Redmond, WA

10/2002 – present: architect, enterprise tools

Rational Software Corporation – Redmond, WA

11/2000 – 7/2002: chief architect, practitioner desktop group

Inline Software Corporation – Sterling, VA

5/1997 – 11/2000: chief technical officer

Objective Enterprise LLC – Reston, VA

4/1995 – 4/1997: principal

Personal Library Software Inc. – Rockville, MD

6/1994 – 4/1995: manager, core technology
3/1994 – 6/1994: senior architect

NeXT Computer Inc. – Redwood City, CA

12/1992 – 3/1994: senior software engineer
3/1989 – 12/1992: software engineer

Micro Computer Systems Inc. – Silver Spring, MD

10/1986 – 3/1989: senior architect

RDP Inc. – Manassas, VA

3/1986 – 10/1986: senior systems analyst

TransAmerican Computer Systems Inc. – Fairfax, VA

6/1985 – 3/1986: senior software engineer

Business Management Systems Inc. – Fairfax, VA

10/1981 – 6/1985: systems programmer/ analyst

Patents

Dynamic object communication protocol, 1994
Method for providing stand–in object, 1994
Method and apparatus for mapping objects to multiple tables of a database, 1994
Method for associating data-bearing objects with user interface objects, 1994
Systems and methods that synchronize data with representations of the data, 2004
Partition-based undo of partitioned object graph, 2005

Books

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools (Wiley 2004)
Software Factories Applied (Wiley 2007)

In a nutshell, *software factories* is a methodology for creating domain-specific development processes and custom development environments to support them. A lot has been written about them, so I am not going to take on the whole topic here. Instead, we will focus only on how they relate to DSLs and how they address the problems described previously.

Let us start our discussion of software factories by looking at the root cause behind the pitfalls described previously. All of the problems boil down to the desire for a universal solution technology that can be applied to all problems. In the case of DSL technology, this desire manifests itself in the assumption that a DSL is the best solution, no matter what the problem. This tendency, which I call "the hammer syndrome," is not unique to DSLs, of course. It afflicts any technique or technology that has proven useful for solving a frequently encountered class of problems. What causes the hammer syndrome? Michael Jackson offers this diagnosis (see Resources):

Because we don't talk about problems, we don't analyze or classify them, and so we slip into the childish belief that there can be Universal Development Methods, suitable for solving all problems.

Despite the marketing hype, we know that there cannot be a universal modeling language that describes every domain precisely enough to support computation. By the same reasoning, we should also know that a DSL cannot be the best solution to every problem. Given the power and elegance of DSL technology, however, it's easy to start thinking that every problem looks like a nail.

If we could describe the classes of problems that DSLs are good at solving, then we should be able to codify some guidance for using them appropriately. But why stop there? Why not develop a systematic way to analyze and classify problems? If we could describe frequently encountered classes of problems, then we should be able to work on developing techniques or technologies that are good at solving the problems in each class. This approach is the basic thinking behind patterns and frameworks. It is also the basic thinking behind software factories, which are based on the same underlying principles and assumptions as patterns and frameworks.

A factory describes one or more classes of problems that are encountered frequently when building a specific type of deliverable, such as a smart client, a mobile client, a Web client, a Web service, a Web portal, or a connected system. These examples are horizontal, meaning that they are based on specific technologies and architectural styles. A software factory can also target vertical deliverables, such as online auctions, commerce sites, banking portals, or even retail banking portals. As we shall see, the description of the frequently encountered classes of problems supplied by a factory identifies one or more domains that may be candidates for DSLs.

Why target a specific type of deliverable? Why not use a generic problem analysis and classification mechanism like an enterprise architecture framework (EAF)? There are several well-known EAFs, such as the Zachmann Framework, the Open Group architecture framework (TOGAF), and the pattern frame from Microsoft patterns and practices (see Resources). As you might have already guessed from these examples, EAFs are often expressed as grids with rows representing different levels of abstraction and columns representing different aspects of the development process.

The motivation for using a factory instead of an EAF is the same as the motivation for using a DSL instead of a general-purpose modeling language. Like a general-purpose modeling language, an EAF is merely documentation that imprecisely describes almost any deliverable. By contrast, a factory contains a schema that describes a specific type of deliverable precisely enough to support many useful forms of computation.

Clearly, different classes of problems are frequently encountered when building different types of deliverables. A mobile client presents different problems than a smart client, and a Web service presents different problems than a Web portal. Such contrasts do not mean that we should think in terms of building two different DSLs, one for each type of deliverable. We need to think more broadly than a single language when dealing with problems as large as building an entire application, to avoid the kinds of scoping problems described previously.

We should therefore think in terms of building two different factories. Those two factories may indeed contain two different DSLs. More likely, however, each factory will contain multiple DSLs, since multiple DSLs will be needed to support the development of the entire deliverable. How many of those DSLs will appear in both factories? The answer would require a bit more analysis than space allows here, but we can safely assume that there will be some DSLs in common, and some DSLs unique to each factory. In other words, the differences between the two factories will probably go beyond two different DSLs.

There will be many different DSLs and they will be integrated in different ways. The two factories will have different schemas. Also, because a factory schema is not obliged to be rectangular, we can define as many classes of problems as we need to accurately analyze a given type of deliverable. Relationships between different classes of problems are also easier to express, since they do not rely on adjacency between cells arranged in rows and columns.

To develop a factory schema we analyze the deliverables of a given type to identify and classify the problems we frequently encounter when building them. We start by breaking down the problem of building a typical deliverable into smaller problems, and we continue until the leaves of the tree are bite-sized problems dealing with just one part of the deliverable, such as data contracts, service agents, workflows, or forms. In practice, we often end up with multiple, bite-sized problems working with different aspects of the same part, such as its structure, its behavior, or its security policy. By decomposing the problem, we separate concerns and identify discrete problems that can be worked relatively independently.

Generalizing Problems

The next step is to generalize from specific problems encountered when building several typical deliverables to classes of problems that we can reasonably expect to encounter when building any deliverable of the same type. The specifics will vary from one deliverable to the next, of course, but the classes of problems will remain the same. The specific services accessed will vary from one smart client to the next, for example, but we can reasonably expect to encounter service access problems when building any smart client.

Look more closely at the smart client example to see the kinds of problems we can define. A smart client has a composite user interface that displays multiple views. Each view reflects the state of some underlying model. The state of the view and the state of the model are synchronized by a presenter. Changes in one view may affect another view through events. Presenters collaborate by publishing and subscribing to events, enabling scenarios like master detail, where the contents of a detail view, such as a form, track the selection in a master view, such as a tree or list view.

Models, views, presenters, and events are grouped into modules that can be loaded on demand. Smart clients also often interact with databases through data access layers or with remote services through service agents. These components can cache the results of database or service interactions for efficiency or let an application continue working offline when a database or service goes offline, synchronizing changes when it comes back online. Security is another key concern, as a smart client may interact with multiple databases or remote services using different authorization or authentication mechanisms.

As the example suggests, a good way to find frequently encountered classes of problems for any deliverable type is to follow the architecture. We start with the largest components of the architecture and break them down into smaller ones. We then identify the problems we face in designing, implementing, testing, configuring, or changing each component. The example also illustrates that different parts of a deliverable present different kinds of problems in different phases of the life cycle, and at different levels of abstraction. Building service interfaces presents different kinds of problems than building service proxies, and testing the data access classes presents different kinds of problems than designing the business logic.

To put this discussion into the vocabulary of DSLs, we can think of each class of problems as a problem domain. Building an entire deliverable is a large problem domain containing many concerns. We therefore decompose it into smaller problem domains until the leaves of the tree are bite-sized problem domains that can be worked relatively independently. For a smart client, these independent problem domains might include:

- User interface design, consisting of view development, model development, presenter development, and view integration
- Service design, consisting of contract design and service interface construction
- Business logic construction, consisting of business logic workflow and business logic classes
- Data access construction, consisting of service proxy construction and data access classes

Some of these domains may be good candidates for DSLs.

This conclusion is important. We can now answer one of the questions posed earlier: How do we know what to model? The first step in answering that question is to identify the type of deliverable we are trying to build. A fundamental tenet of DSL technology is that a modeling language must focus on a specific domain to provide precision. We cannot focus on a specific domain outside the context of a specific type of deliverable. The second step is to decompose the target deliverable type into bite-sized problem domains that can be worked relatively independently.

A factory schema captures the results of this analysis in a computable model that identifies those problem domains. (The factory schema is a model based on a DSL supplied by the factory-authoring environment.) If we do a good job of defining the factory schema, then DSLs developed for those problem domains will be scoped in a way that effectively separates concerns.

Now we will look more closely at the factory schema to see exactly how those problem domains are described, how DSLs map onto them, and how they are organized in the factory schema in a way that makes the DSLs easy to stitch together to support a development process.

When to Use DSLs

Of course, DSLs are not the only choice of technology available, as noted earlier. DSL-based designers are just one type of asset that can be supplied by a factory. Other asset types include Visual Studio templates, text generation templates (known as T4 templates in the DSL Tools), recipes defined using the Guidance Automation Toolkit (GAT), code snippets, documentation describing patterns and practices, class libraries, and frameworks (see Resources).

We have already seen that different parts of a deliverable may present different kinds of problems. Building them may therefore require different techniques or technologies. For example, we might use .NET framework classes and code snippets to build data access classes, but we might use a wizard to expand a template that creates a starting point for a service agent, and then complete the coding by hand using guidance provided by documentation. Note that in this example, we are not using any DSLs. It's quite legitimate to build a factory that provides only passive assets like patterns and practices.

We are now ready to look at another question posed earlier: How do we know when to use or not to use a DSL? I will rephrase the question based on this discussion. How do we know that we should use a DSL, instead of some other type of asset, to solve the problems in a given domain?

To answer that question, we need to look more closely at the factory schema. We have already said that the factory schema is a model designed to support computation, and that it describes one or more problem domains representing the classes of problems that are encountered frequently when building a specific type of deliverable. We have also talked about decomposing large problem domains into smaller ones that can be worked relatively independently. We are now ready to put these ideas together to explain how a factory schema works, and how it helps us determine when to use or not to use a DSL.

A factory schema is a tree. Each node in the tree is called a viewpoint. A viewpoint corresponds to a problem domain. The viewpoint at the root of the tree corresponds to building an entire deliverable. The viewpoints below the root are derived by decomposition. A viewpoint describes its corresponding problem domain by describing the problems we may encounter and telling us how to solve them. It describes the solutions in terms of activities we may perform, explaining how to perform each activity, and how to recognize when the activity is required. The activities are described in terms of the work products they manipulate. Work products are the artifacts we build to produce a deliverable. Finally, a viewpoint describes a set of assets supplied by the factory to help us solve the problems we may encounter in the domain. The assets are designed to support the activities, often by fully or partially automating them.

Take the contract design viewpoint as an example. The work products are data contracts and message contracts. The activities include creating or destroying a data contract; adding, removing, or modifying data elements in a data contract; creating or destroying a message contract; adding, removing, or modifying methods in a message contract; and connecting data contracts to method arguments in message contracts. The assets supplied by the factory for the contract design viewpoint include a DSL-based Contract Designer, including a set of T4 templates for generating the contract implementations; a set of documents explaining how to build a data contract and how to build a message contract; a checklist for validating contract designs; a set of common message exchange patterns; and a set of patterns describing the message contracts and data contracts used by each message-exchange pattern.

Clearly, this viewpoint was a good candidate for a DSL. We can now codify some guidance regarding when to use or not to use a DSL: A DSL should be used instead of some other type of asset when it is the best mechanism available for supporting the activities of a specific viewpoint.

Employ a Maturity Curve

This answer begs another question, of course. How do we know when a DSL is the best mechanism available?

The best way I know to answer that question is with a maturity curve. We start by making modest investments in simple assets for a given viewpoint and then gradually increase our level of investment over time, building increasingly sophisticated assets as we gain a deeper understanding of the problem domain targeted by the viewpoint and greater confidence that we have scoped and defined the viewpoint correctly. Then we place it correctly in the context of enclosing and neighboring viewpoints as part of a factory schema.

At the low end of the maturity curve are simple assets like guidelines and other unstructured documents that help the reader know what to do and how to do it. After using and refining the documents, we may decide to formalize them into patterns. Over time, we may decide to implement the patterns as templates that can be instantiated rapidly using wizards or recipes developed with the GAT. At some point, we may decide to replace the templates with class libraries delivered as example code or as assemblies.

The class libraries, in turn, may become frameworks, servers, or other platform components, as we connect the classes to form mechanisms that embody the underlying patterns. At this point, we have reached the high end of the maturity curve, where DSLs are most effective. We can use DSLs to wrap a framework, a server, or other platform components. They are particularly effective when the metadata captured by the designer can be consumed directly by the framework, a server, or other platform components, or used to generate configuration and completion code. DSLs are very good at solving these kinds of problems:

- Providing a graphical editor for an existing XML document format
- Rapidly generating a tree view and form-based user interface
- Capturing information used to drive the generation of code or other artifacts
- Expressing a solution in a form that is easier for people to understand
- Describing abstractions that cannot be discovered easily from the code
- Describing domains that are easy to represent diagrammatically, such as flows, conceptual maps, or component wirings

In general, they are good at handling domains containing moderate amounts of variability. On the one hand, using a DSL to configure a run time that has a few simple variables with fixed parameter values would be overkill. A form containing a few drop-down choices, check boxes, or radio buttons would be a better choice. On the other hand, a DSL may not be powerful enough to support the construction of complex procedural logic. A conventional, textual programming language would be a better choice.

It is also possible to use a DSL to implement a pattern language without the assistance of a framework, a server, or other platform components. One situation in which this approach makes sense is generating from models based on the DSL to models based on less abstract DSLs. For example, we might generate models describing message contracts from models describing business collaborations. Another situation in which this approach makes sense is generating from models based on a DSL to code that effectively implements a framework, a server, or other platform components. For example, we might generate a device-independent platform used by high-level code by generating services that run on a target device from a model describing the device characteristics.

Supporting a Development Process

Of course, it sometimes makes sense to build a DSL without climbing the maturity curve described previously. Building simple DSLs can be quite inexpensive, and a small team may be able to rapidly refine their understanding of a target domain by iterating over the definition and implementation of a DSL. However, I would not recommend attempting to build a complex designer or attempting to deliver even a simple designer into a large user base that requires significant training, such as a field organization, without first making sure you understand the target domain and its relationships to enclosing and neighboring domains.

Some factories may consist of nothing more than documents describing patterns and practices organized around the factory schema. Other factories may have a DSL for every viewpoint. In practice, we tend to see a mix of asset types in the typical factory, with a large number of viewpoints served by code-oriented assets, such as patterns, templates and libraries, and a modest number of viewpoints, usually the most stable and well understood, served by powerful DSL-based designers.

We have addressed two of the three questions introduced earlier. Can we answer the third? How do we stitch DSLs together to support a development process? As it turns out, we have already covered most of the ground required to answer that question. Given a good factory schema that decomposes the problem of building a specific type of deliverable into relatively independent viewpoints, the next step is to discover relationships between the viewpoints.

We start by looking at how the work products in one viewpoint relate to the work products in other viewpoints. Since we are working with DSL-based viewpoints, the work products are defined in terms of model elements on diagrams. There are many ways in which they can relate. Here are just a few:

- Work products in one viewpoint may provide details about work products in another. For example, a class diagram may provide details about the implementation of an application described in a system diagram.
- Work products in one viewpoint may use work products in another. For example, a business collaboration diagram may use message types defined in a contract design diagram.

- Work products in one viewpoint may be wholly or partially derived from work products in another. For example, a business collaboration diagram may be partially derived from a use case on a use diagram.
- Two viewpoints may provide different information about the same work products. For example, a class diagram describes the structure of a group of classes, while a sequence diagram describes their behavior in terms of interactions.
- Work products in one viewpoint may describe the relationship between work products in others. For example, a deployment diagram contains elements that describe the relationship between an element in a logical, data-center diagram and one or more elements in a system diagram.

Once we have relationships between viewpoints based on work products, the next step is to look at how activities in the related viewpoints affect each other. For example, take the relationship between the logical data-center diagram, the deployment diagram, and the system diagram. In the system design viewpoint supported by the system diagram, the primary activity is configuring systems, applications, and endpoints. In the logical, data-center design viewpoint supported by the logical, data-center diagram the primary activity is describing logical host types and their configurations and settings. In the deployment diagram the primary activity is mapping systems onto logical host types.

The relationships between the viewpoints support an activity that combines these activities to form a workflow called design for deployment, where a solutions architect configures systems, then maps the system configurations onto logical host types captured by an infrastructure architect in a trial deployment, and then validates the trial deployment to see if the system configurations will deploy into the data center correctly.

We can now see how DSLs fit into the larger picture of software factories. We introduced DSLs at Microsoft to support the Software Factory methodology, and the technologies are highly complementary. A factory schema breaks a large problem domain into smaller ones. The viewpoints at the leaves of the schema target relatively independent components or aspects of the target deliverable type. Some of the viewpoints may be candidates for DSLs from day one. Others may start with assets further down the maturity curve and gradually evolve toward DSLs over time as experience is gained. Some may remain volatile and challenging, relying on relatively unsophisticated assets, such as documentation, for the life of the factory.

DSLs that target the factory viewpoints will be well scoped, if the factory is well designed, and will have well-defined relationships to other DSLs that make it easy to stitch them together to support development processes. At some point, we may offer technology that generates DSL integration code from the factory schema, and run-time technology that supports the generated integrations.

The next time someone asks you to build an isolated DSL for a broad domain like banking applications, ask them what aspect of banking they want you to target. When you get a blank stare in response, you will know that what they really need is a factory that breaks the domain into bite-sized nuggets, some of which may be supported by DSLs and some of which may be supported by other types of assets, not a bare naked language. •

Resources

MSDN
Enterprise Solution Patterns Using Microsoft .NET
Microsoft Patterns & Practices
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/esp.asp

MSDN – Microsoft Visual Studio Developer Center
Domain-Specific Language Tools
http://msdn.microsoft.com/vstudio/DSLTools/

Guidance Automation Extensions and Guidance Automation Toolkit
http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/default.aspx

Jack Greenfield's Blog
http://blogs.msdn.com/jackgr/archive/2006/01/22/502645.aspx

Software Factories
http://msdn.microsoft.com/vstudio/teamsystem/workshop/sf/default.aspx

Object-Oriented Analysis and Design with Applications, 2nd Edition, Grady Booch (Addison-Wesley Professional 1993)

OMG
OMG Model-Driven Architecture
www.omg.org/mda/

The Open Group
Architecture Forum
www.opengroup.org/togaf/

Problem Frames: Analyzing and Structuring Software Development Problems, Michael Jackson (Addison-Wesley Professional 2000)

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools
Jack Greenfield, et al. (Wiley 2004)

Wikipedia
Enterprise Architecture
http://en.wikipedia.org/wiki/Enterprise_Architecture

The Zachman Institute for Framework Advancement
www.zifa.com

About the Author

Jack Greenfield is an architect for enterprise frameworks and tools at Microsoft. He was previously chief architect, practitioner desktop group, at Rational Software Corporation, and founder and CTO of InLine Software Corporation. At NeXT Computer he developed the enterprise objects framework, now a part of Web objects form Apple Computer. A well known speaker and writer, he is a coauthor of the best-selling and award-winning book *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, with Keith Short, Steve Cook, and Stuart Kent. Jack has also contributed to UML, J2EE, and related OMG and JSP specifications. He holds a B.S. in Physics from George Mason University.

Domain-Specific Modeling

by Steve Cook

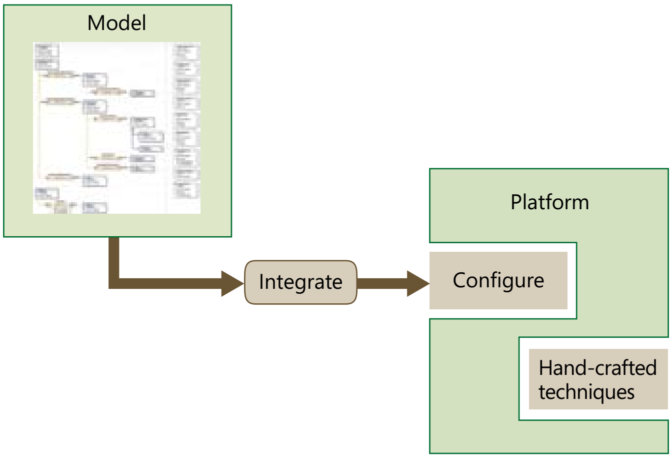
Summary

Domain-specific languages (DSLs) are special-purpose languages designed to solve a particular range of problems. DSLs are nothing new. Common examples are: HTML, designed for representing the layout of Web pages; SQL, designed for querying and updating databases; and regular expressions, designed for finding and extracting particular patterns in textual strings. The essence of a DSL is that it makes a large problem smaller. Without HTML, the problem of rendering Web pages with more or less equivalent appearance on millions of screens connected to different kinds of PCs would be insurmountable. Without SQL, the problem of allowing multiple concurrent users to establish, query, and combine large lists of data would be a massive programming task. Without regular expressions, searching for substrings within text would involve writing a complicated program. There is a pattern here: to turn large problems into small problems, identify a language that efficiently expresses that class of problems, and apply the pattern to connect expressions in the language into the final environment in which the problem is to be solved. Given that we are confronted daily with large problems to solve, let's look at how we can harness this idea in practice.

The DSL pattern has three primary components (see Figure 1). First, there is the model that the DSL user creates to represent the problem. This model might be a textual expression, as in the cases cited earlier in the article summary, or it might be an annotated diagram. Second, there is a platform that will be used to execute the solution to the problem at hand. In the case of HTML, this platform will be a Web browser; in the case of SQL, a database; and in the case of a regular expression, a text editor or programming environment. Third, there is a method to integrate the language expression into the platform to configure it to the problem at hand.

There are two primary means of integration: interpretation and generation. With *interpretation*, part of the platform itself is dedicated to recognizing expressions in the DSL and executing their intent. With *generation*, a separate procedure is used to convert the DSL expression into something that the platform recognizes natively. You can also see the use of hand-crafted techniques (see Figure 1). A particular model will inevitably only represent one aspect of the problem to be solved, and other techniques must be used to solve the rest of it.

Figure 1 The DSL pattern



In recent decades model-driven techniques have been proposed widely as a means to increase the efficiency of software development. Under names such as structured, object-oriented, or analysis and design, the idea is to draw a diagram that represents an aspect of the system under construction, and to use that diagram directly to help generate or implement that system. Such was the vision behind Computer Aided Software Engineering (CASE) tools, promoted by many vendors during the 1980s, and more recently model-driven architecture, promoted by the Object Management Group.

On closer inspection, we can see that model-driven development is exactly a case of the DSL pattern. The model is an expression in the domain-specific language, this time a modeling language; the platform is the execution platform for the system under construction; and the integration step is a code generator that transforms the model into code that executes on the platform.

Although it is definitely an application of the DSL pattern, CASE was not a great success. If we try to analyze why, we can identify two primary reasons. First, the models were not a particularly pleasant or convenient expression of the problem at hand. Working programmers would not necessarily recognize that the alternative expression of the problem manifested in the diagrammatic models was any better than simply writing the code in a general-purpose language, and therefore would resist the introduction of these techniques. Second,

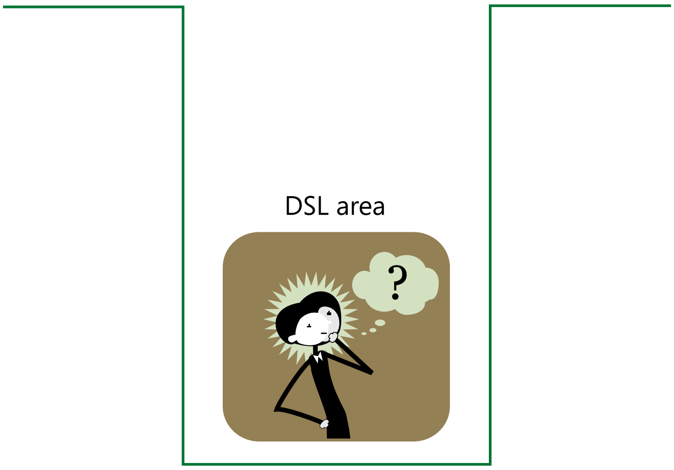


Figure 2 The customization pit

a lot of code was generated to bridge the abstraction gap between the models and the execution platform. Any mistakes or inefficiencies in this generation step would tend to be corrected not by fixing the generator but by fixing the generated code, thereby breaking the link between the model and the solution and rendering the pattern inoperative.

We may derive some important conclusions from this analysis. First, it's important for the meaning of the models to be readily apparent to people familiar with the domain. The language must be designed carefully to fit the intended purpose. We will return to this topic later. Second, to successfully deploy such approaches it is very important to win the hearts and minds of working developers, so they can see that the pattern or tool will help them to get their work done and will adopt it. Finally, the generation process must be efficient, and it must be straightforward to remedy errors in it and to customize it.

An increasingly important motivation for considering the use of DSLs is the sheer diversity and interconnectedness of today's systems. Like it or not, any system of significant size involves a combination of many different kinds of technologies and representations: programming languages, scripting languages, data definition and representation langua ges, control and configuration languages, and so on. Given a particular feature in the requirements of the system to be built and deployed, it is quite unavoidable for different aspects of that feature to be scattered across all of these different technologies and representations. This problem cannot possibly be solved by individual improvements in any of these different technologies—it must be addressed in a holistic way, by finding a level of representation that spans all of the implementation components and technologies. The DSL pattern provides a means to do this.

The benefits of DSLs can be considerable. DSLs can enable much better communication with stakeholders than lower-level technologies. Changes in requirements can be represented by changes in the model, and thereby implemented rapidly. Changes in the technological platform can be incorporated by manipulating the integration step of the pattern, leaving the modeled representation unchanged. The volume of code to maintain is smaller, and bugs in the generated code can be fixed by fixing the code generator.

These benefits do not come for nothing. To achieve them requires tooling up for the pattern. Implementing a DSL from scratch, whether textual or graphical, is a major enterprise and not to be undertaken lightly. To alleviate these costs is the objective of an emerging category of tools called language workbenches. A *language workbench* is a set of tools that are targeted specifically at the creation and deployment of new DSLs into existing development environments. Language development is itself a domain that is highly amenable to the application of the DSL pattern—and so a crucial aspect of a language workbench is that it is bootstrapped—that is built using itself. An example language workbench is the DSL Tools, part of the Visual Studio SDK, which enables the rapid development of graphical DSLs integrated into Visual Studio 2005. We'll take a more detailed look at the DSL Tools later.

Kinds of DSL

A *domain* is a subject area or area of concern to a particular set of stakeholders in the system. Domains might be horizontal, technical domains, such as user-interface, data persistence, communication, or authentication. Or they might be vertical, business domains, such as insurance, telephony, or retail. Domains can overlap. Domains can be parts of other domains. A domain is bounded by the concerns of its stakeholders, and as these concerns evolve, so does the domain. Hence, domains are dynamic. When stakeholders see expressions or models in the language, they must immediately recognize them as directly expressive, useful, relevant, and empowering.

Languages may be textual, diagrammatic, or a combination. Modern high-performance personal computers with bitmapped displays are well equipped to implement diagrammatic languages, which are often much more expressive in relatively nontechnical domains than textual languages. As they say, "a picture is worth a thousand words."

Figure 3 The customization staircase

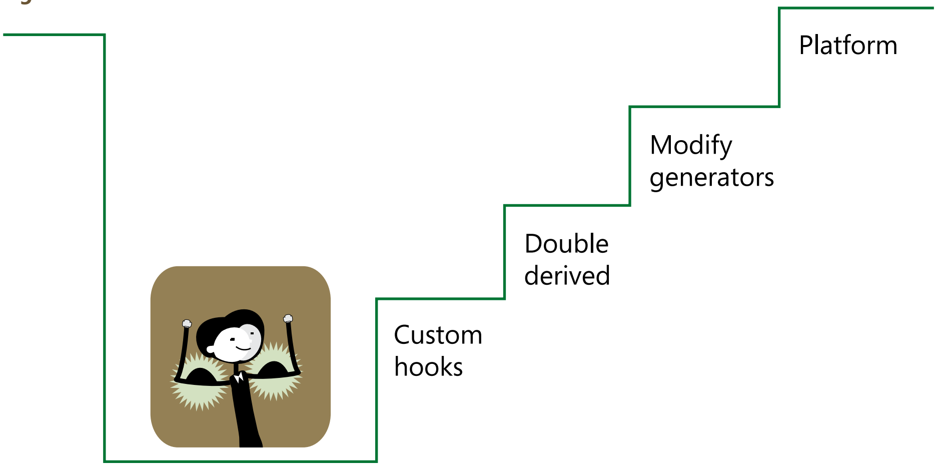
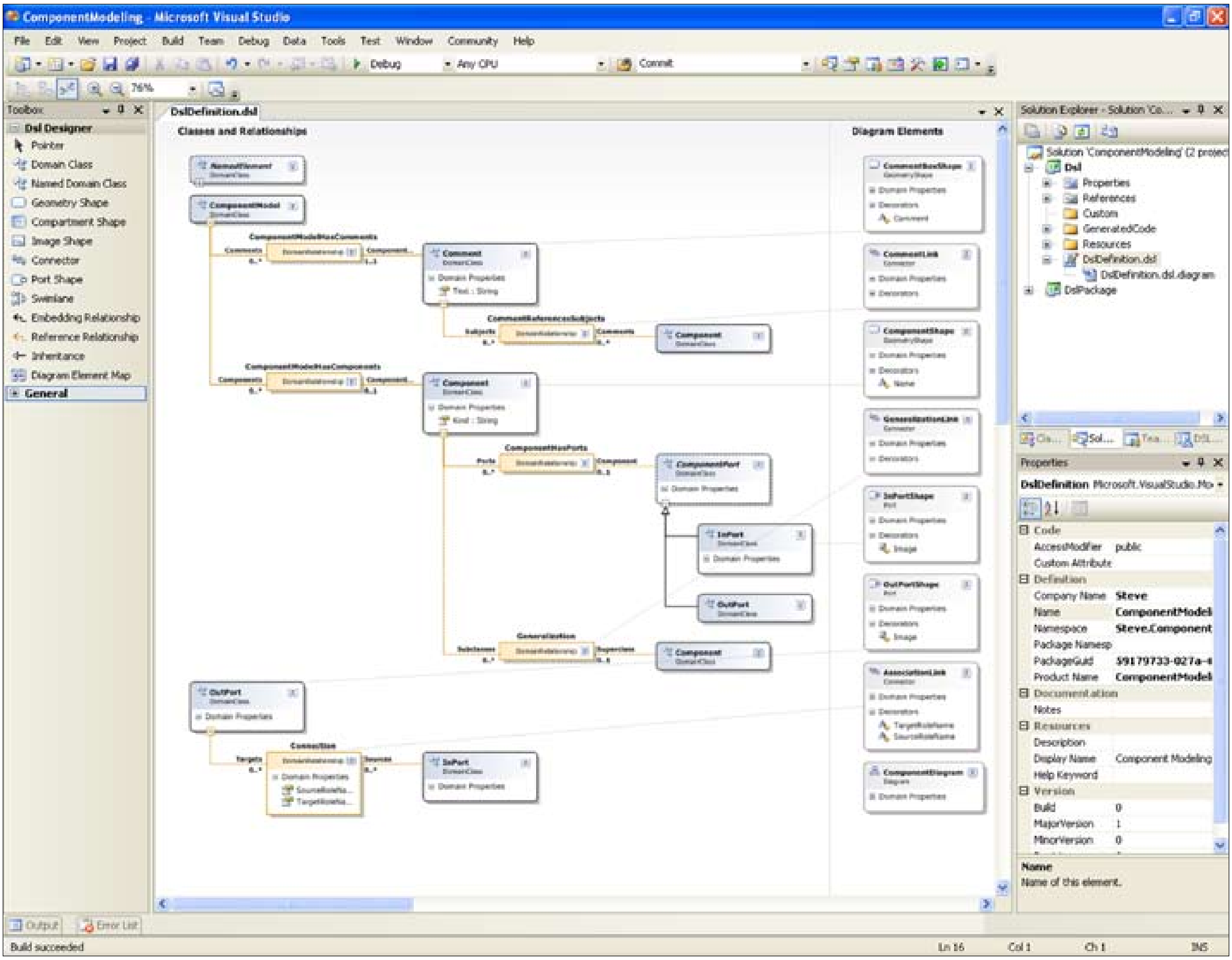


Figure 4 A starting solution for component models



Whether textual, diagrammatic, or a combination, a DSL must be implemented to make it useful. Implementing a DSL means building a tool that allows users to edit expressions or models in the language. Such a tool would not normally stand alone. Because a DSL typically addresses only a portion of the entire problem at hand, the DSL tool must be tightly integrated into the development environment.

Integrating the DSL

Figure 1 shows that the language must be integrated into the platform. One aspect of this integration is that the expressions in the language—models—must be converted into a form that is executable by the platform. This aspect is straightforward if the platform is designed to directly interpret the models. More commonly, however, it is necessary to transform the models into a form that can be interpreted. Typically, this transformation involves the generation of code that can be compiled and linked into an executable that runs against the platform.

An important advantage of interpreting models directly is that no compilation step is required, which makes it straightforward to deploy new models and to remove old ones, even in the context of a running system. Code generation on the other hand has advantages, especially early in the evolution of a DSL:

- It is simple to implement.
- Existing mechanisms for compiling, linking, and debugging the code can be used.
- It is straightforward to customize the generated code, and thus extend the scope of the DSL.

It is important that a DSL should be customizable. Figure 2 illustrates what can happen with a noncustomizable DSL. The scope of solutions that can be addressed by using the DSL forms in an area called the *customization pit*. Although it is simple to use the language to solve problems in this area, as soon as it is necessary to step outside of this area, users encounter an insurmountable cliff because they would have to modify the platform itself, which is not often feasible. Modification might be acceptable with a mature DSL in a mature domain, but in other cases it can be a major obstacle to success.

With a code-generation approach, it is straightforward and highly desirable to turn this cliff into a staircase, which is eased greatly by the abstraction facilities offered by modern general-purpose programming languages, especially static type checking, inheritance, virtual functions, and partial classes (see Figure 3). The latter are a particular useful feature of the C# language that allows the definition of a class to be spread across multiple files, which are compiled and linked together into a single class definition. This definition makes it simple to generate part of a class and write the rest by hand, and if subsequent regeneration is needed, then the hand-written part is preserved without any difficulty.

Note that such facilities were not widely available in mainstream languages during the 1980s, which meant that it was much harder to engineer this kind of customizability into CASE tools.

The first step of the staircase can be enabled by inserting explicit customization hooks into the DSL itself. When one of these hooks is enabled, then instead of generating complete code a stub is generated that requires the user to handwrite some code that completes the program. If they do this incorrectly, the compiler's error messages will tell them how to correct what they did.

The second step of the staircase can be enabled by generating code in a "double-derived" pattern. Instead of a single class, a pattern of two classes is generated. The base class contains all of the generated method definitions as virtual functions; the derived class contains no method definitions but is the one that is instantiated, which allows the user, in a partial class, to override any of the generated functions with their own version. Of course the use of virtual functions incurs a run-time penalty, but usually the benefit of customizability outweighs this cost.

The third step of the staircase is enabled by making the code generators themselves available for substitution. This availability might be used when retargeting the language onto a new platform, or adding major feature areas, or fixing bugs in the generators.

The final step of the staircase is to modify the platform itself, which as already noted is not often a feasible option.

Referring back to the DSL pattern introduced in Figure 1, there may be certain kinds of platform configuration that do not require the full power of a DSL. For example, there may be just a few configuration options that can be input using a simple form or wizard. Or perhaps the configuration of the solution requires the selection of features from a list, much as the installation of a software package often involves the selection of which features the user wants to have. In such simple cases the full power of a DSL is unnecessary. DSLs come into their own to configure aspects of the solution that involve significant complexity and validation in their own right, in which case the features described in this article can be used to create the DSLs cheaply and effectively.

Validation and Error Handling

A key advantage of the DSL pattern is that the model can be validated before integrating it into the platform. Constraints on how design elements may be connected and named can be enforced at the level of the model, which can catch many kinds of errors much earlier than would otherwise be the case. For example, architectural design rules such as layering, avoiding circularity of dependencies, consistency of user interaction, or ensuring that the design matches the limitations of the implementation can be enforced.

Validation can be either hard or soft. *Hard validation* means that the user, creating a model, is simply unable to create an invalid model by the way that the modeling tool responds to interactions. *Soft validation* is run in a batch, often when the model is saved or prior to code generation, or on explicit request, and will report errors to the user and refer them to the source of the error. Soft validations are typically cheaper to implement, but hard validations can add significantly to the productivity of the user experience.

Forward and Reverse Generation

It is frequently proposed that a model should be automatically generated from a solution, that is, that code generation should also work in reverse. This proposal is often called "reverse engineering" and is a feature claimed by several CASE tools.

In reality, you can only extract a model from code if the model is little more than a diagram of the code. That extraction can be useful—it's what sequence diagrams, for example, were invented for—but it is not the approach we are addressing with DSLs. A good DSL is close to the domain model and comprehensible in the terms of the domain. The templates that transform the model to software are created by the developers in that domain, capturing their expertise in developing software in that domain. This approach is very different from the typical "round-trip" tool, where there is only one way, or a very few ways, of mapping a model to code.

Extracting just one design aspect from the code of a system cannot be done unless that aspect has been kept carefully separate, and the code is marked in such a way that the aspect can be extracted. In practice, facilities of this kind are normally intended to allow model-driven development and hand-written development to be seamlessly mixed, so that when code has been generated, developers can modify this code by hand, and the result can be transformed back into the model to keep the model synchronized with the code.

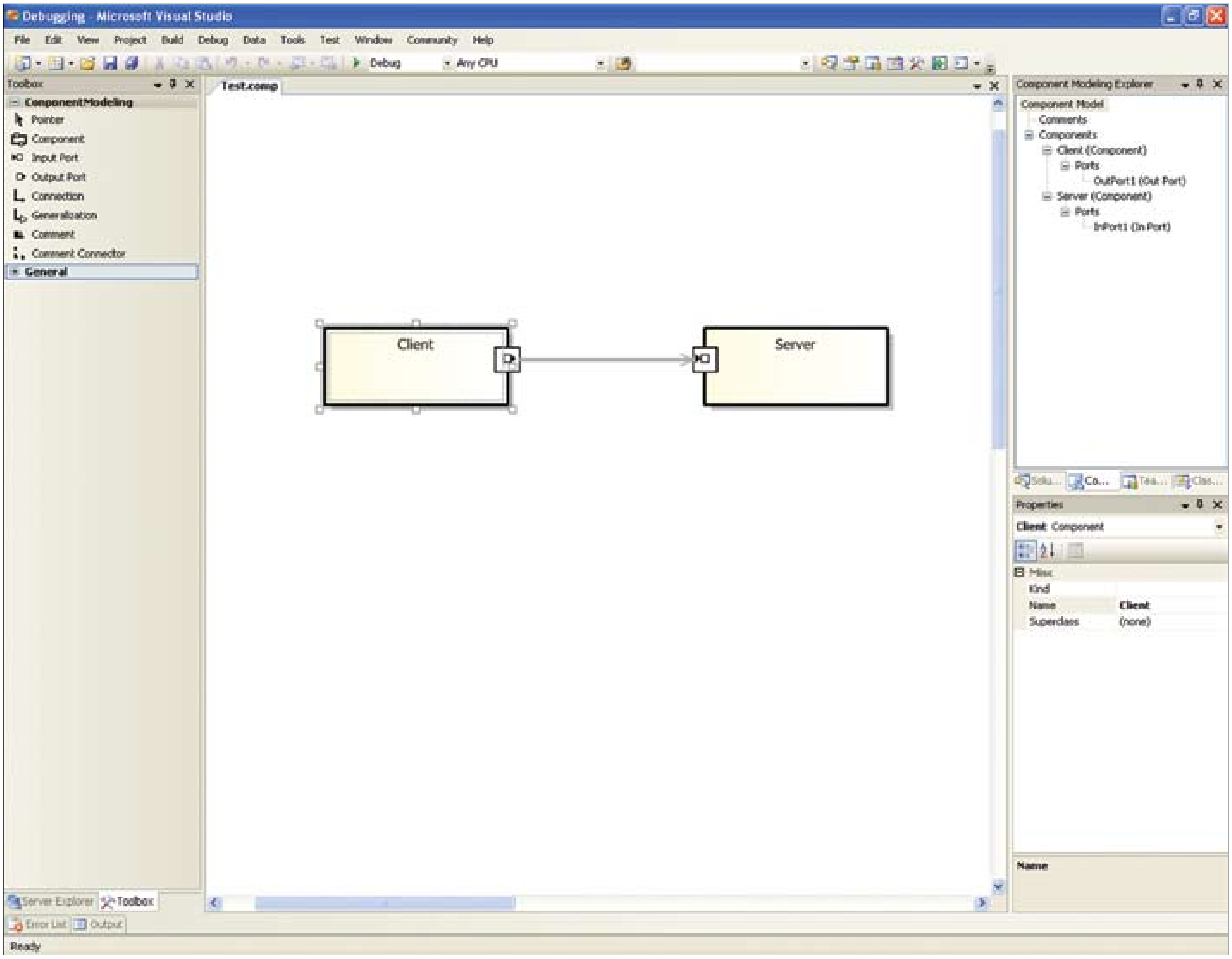
> "AN INCREASINGLY IMPORTANT MOTIVATION FOR CONSIDERING THE USE OF DSLS IS THE SHEER DIVERSITY AND INTERCONNECTEDNESS OF TODAY'S SYSTEMS"

By far the easiest solution to this problem is to keep the generated and hand-written code physically separate by using facilities such as partial classes, as already explained. It can also be convenient to generate "starter" code intended to be customized by hand; in this case the code generator must avoid overwriting such code in later generation steps. If elements of the model have been changed sufficiently to cause the hand-customizations to be rendered syntactically incorrect, then this rendering will be picked up by the compiler; and since this is by far the most common case, caused by events such as changing names and structures, there is a lot of mileage in this simple approach.

Instead of using partial classes, the generated code can be marked using comments that can be read by the code generator and that delimit generated code from hand-written code. This approach separates the two areas of code effectively at the cost of including machine-readable comments that can reduce the human readability of code.

A more ambitious approach relies on the structure of the code and the model being extremely similar, either because the model is a very direct representation of the code, or because the code is

Figure 5 Testing the component DSL



Kind	
Name	Client
Superclass	(none)

Name

structured by a very specific and constrained pattern. In such cases a model can be extracted by parsing the code, and the hand-written and generated areas distinguished by the basic structure. Even in these cases, though, there can be considerable ambiguity about what is intended at the level of the model when particular changes are made to the code. Making reverse synchronization work effectively for such cases typically incurs a much higher implementation cost for the tooling than the simpler forward-only approach.

> “TO DESIGN A DSL, IT IS CRUCIAL TO HAVE THE INVOLVEMENT OF THE DOMAIN EXPERTS BECAUSE VERY OFTEN THE BASIC INSPIRATION FOR A DSL WILL BE FOUND ON THEIR WHITEBOARDS”

Using DSL Tools to Build a DSL

The DSL Tools constitute a component of the Visual Studio 2005 SDK that makes it straightforward to implement a DSL, or in other words to build a domain-specific modeling tool. With the DSL Tools, the DSL author defines the concepts of the language, the shapes used to render those concepts on the diagrammatic editing surface, and the various ancillary components that are used to load and save the model to integrate the new tool into Visual Studio and to generate code and other artifacts from the models created using the tool.

To start creating a DSL, the author creates a new Visual Studio project, selecting the Visual Studio template Domain Specific Language Designer. A wizard offers a choice of starting points that provide complete working languages for authors to modify to their needs, rather than having to start from scratch. After selecting the Component Models starting point and defining a few basic parameters, the author is placed into a Visual Studio solution (see Figure 4).

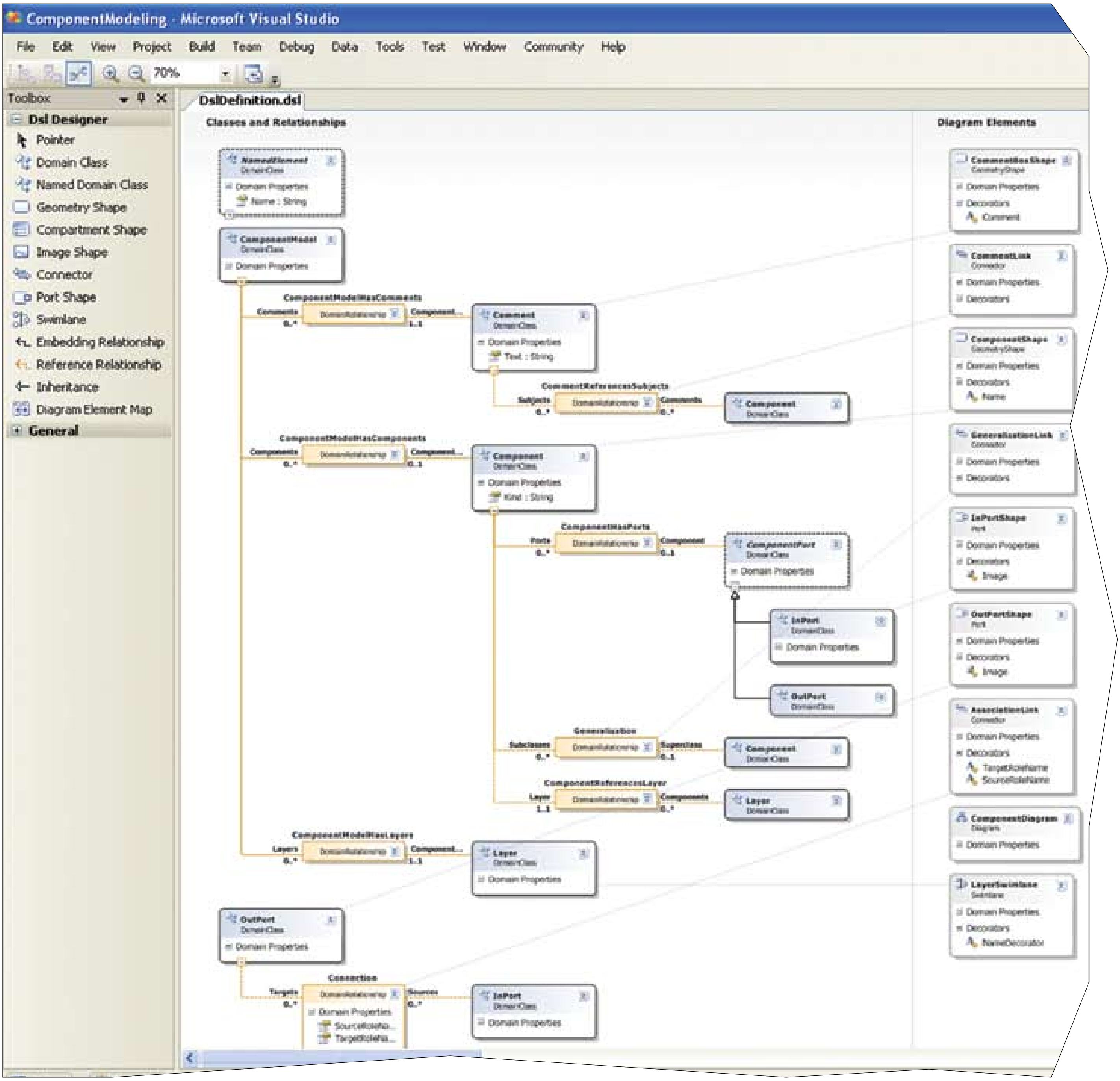
The central area of the screen consists of a diagram divided into two parts, labeled Classes and Relationships and Diagram Elements. Contained in these areas are shapes that represent the constituents of the definition of a DSL. The Classes and Relationships define the concepts that the DSL represents, which in this case are Components, Ports, and Connections. The Diagram Elements define the shapes that will appear in the diagram of the resulting modeling tool.

The other parts of the screen are arranged as follows: on the left is the Toolbox, containing elements that can be dragged onto the diagram to create new classes, relationships, diagram elements, and so on. At the top right is the DSL Explorer, which offers a tree-structured view of the complete language definition. This view shares an area of the screen with other Visual Studio windows including the Solution Explorer, Class View, and Team Explorer. At the lower right is the Properties browser, which offers a detailed drill-in to the properties of the element selected currently on the diagram or the DSL Explorer. Within Visual Studio, users can arrange all of these windows to their taste, including undocking them and distributing them across multiple displays.

The solution shown in Figure 4, which you will recall was offered as a starting-point for the DSL author, defines a complete working language. Two steps are required to see it working. The first step is to generate all of the code and configuration files needed to make the tool, which is done by clicking Transform All Templates in the Solution Explorer. The second step is to press the F5 key, which builds and registers the solution and launches a second copy of Visual Studio to test the DSL. Figure 5 shows the result of opening a file called Test.comp and using the Toolbox to add a couple of components to the diagram.

At this point, authors have many options for how to modify and extend the language. They can delete unwanted parts of the lan-

Figure 6 Swimlanes added to the DSL definition



guage definition. They can add new domain classes and relationships or add properties to existing domain classes and relationships, to represent additional concepts. They can add new shapes and connectors to extend and alter the way that the language's concepts are displayed diagrammatically to its users. They can create code generators that transform a model built using the language into code or configuration data. They can create new validation rules to represent the domain's constraints. They can customize the language, through its extension points, to offer different kinds of user-interface options for the model builder such as forms, wizards, or text editors.

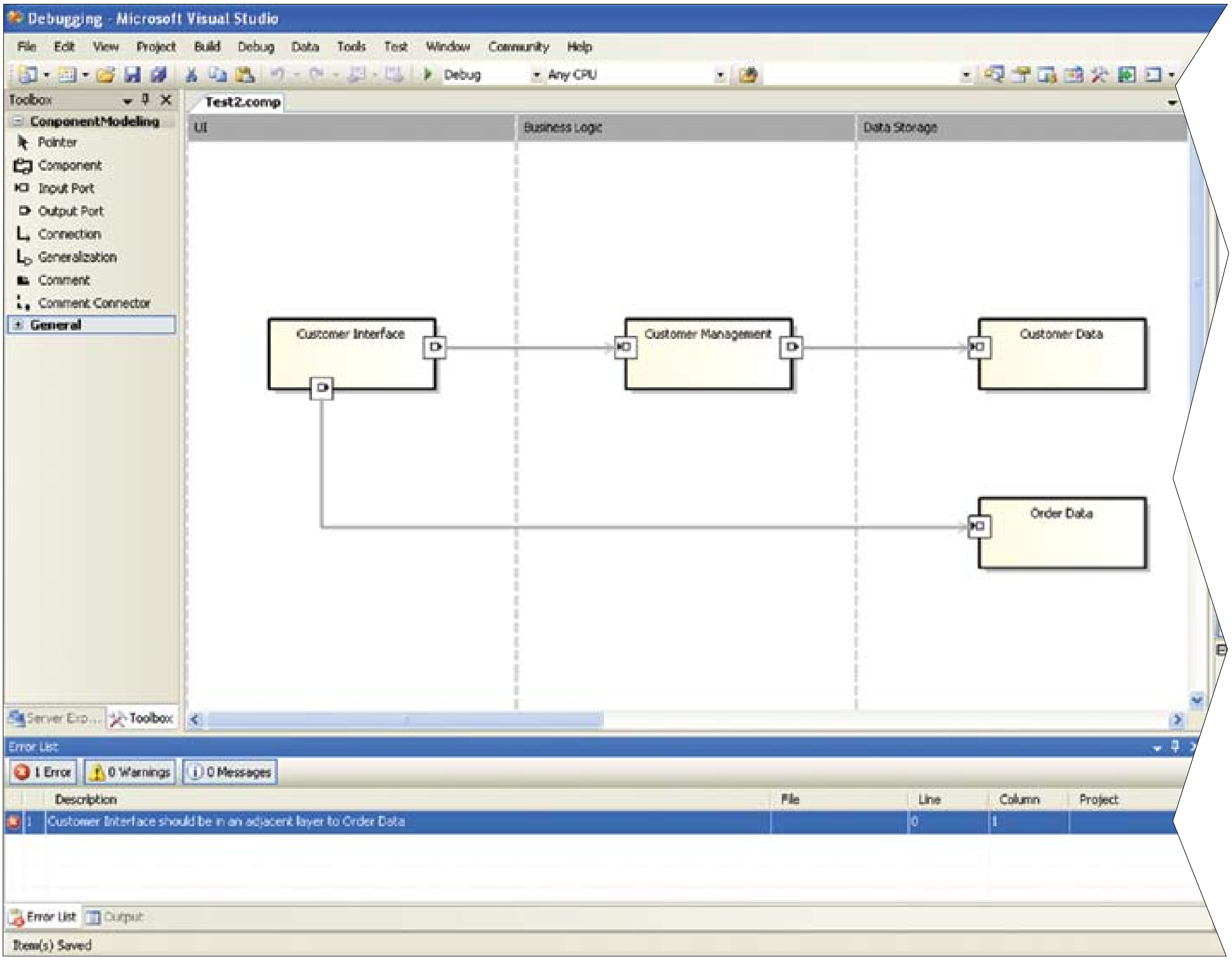
Let us look at how to create a simple extension to the component modeling tool (see Figure 6). We will add "swimlanes" to the tool, to represent architectural layers. These layers can be used to represent design constraints, such as restricting a component to communicate only with components in adjacent layers.

First, we add the concept of a layer to the domain model. This concept is involved in two relationships: a single ComponentModel contains any number of layers, and a Component refers to the layer with which it is associated. The domain class Layer is made a subclass of NamedElement so that it acquires a name property. Then we drag a swimlane off the Toolbox into the diagram area, call it LayerSwimlane, and give it a NameDecorator in which to display the name of the layer. We use the DSL Details tool to declare that a ComponentShape has a layer as its parent and specify how to merge a component into the model when it is dropped onto a layer. Finally, we associate the Layer domain class with the LayerSwimlane shape using the Diagram Element Map tool. These concepts and relationships are shown in Figure 6.

At this point, the designer can be regenerated and tested. Once again, click Transform All Templates, followed by pressing the F5 key, which launches a second copy of Visual Studio. Now the component modeling language has swimlanes, and when components are placed in a swimlane, they get associated automatically with the corresponding layer (see Figure 7).

At this point, the language author can define new validation constraints that will ensure, for example, that each component only communicates with a component in an adjacent layer. This approach can be done by means of *validation methods* defined on partial classes for the relevant domain classes—Component, in the example. Using the DSL Tools, validation methods can be defined that implement the validation logic. All of the calling and error-reporting logic is implemented by the language framework. The warning resulting from implementing such a validation can also be seen in Figure 7.

Figure 7 Component DSL with swimlanes and validation



	Description	File	Line	Column	Project
1	Customer Interface should be in an adjacent layer to Order Data		0	1	

Designing a DSL

Domain-specific modeling has been applied successfully in numerous domains, including mobile telephony, automotive-embedded devices, software-defined radio, financial applications, industrial automation, workflow, Web applications, and others. Several interesting case studies from various vendors can be found at the DSM Forum Web site (see Resources).

To design a DSL, it is crucial to have the involvement of the domain experts because very often the basic inspiration for a DSL will be found on their whiteboards. They will sketch out the way that they think about the important problems in their domain, often using diagrams consisting of shapes connected by lines of various kinds. Working with the DSL developers, these ideas can be translated into domain models mapped to shape models that can be implemented using the DSL Tools.

A fundamentally important aspect of the design of a domain model is its set of validation constraints. For example, it is likely to be necessary that various names are unique within their context, so that elements can be identified uniquely. Also there might be existence constraints; for example, if part of the model represents a mapping or transformation, there must be things at either end. There might be topological constraints, such as the presence or absence of cycles. These constraints must be identified and implemented because to generate code or otherwise implement the model into its environment, it must be valid. As noted earlier, constraints can be implemented either as hard validations that are implicit in the tool's user interface or soft constraints that are run as a batch when the model is opened, saved, code generation is attempted, or on explicit request.

To create the code generators, it is first necessary to have a complete, working, tested implementation of the desired target code. This code must be analyzed to determine which parts of it can be derived from elements in the model and what kinds of patterns must be applied to do this derivation. Sometimes this will require the target code to be refactored to simplify or clarify these patterns. Note that refactoring is a transformation of the code that preserves its behavior, while restructuring it to make it easier to generate or modify. For refactoring to be successful it is necessary to have a suite of tests, which the code must pass before and after refactoring.

Also note that the generators need to operate over only valid models. It should not be necessary to implement the generators defensively so that they also handle invalid models because only valid models should be used as a source for code generation.

At this stage it is also important to consider the customization options that will be offered for the language. If there are places in the code where the user will be required to handwrite their own logic, these places must be identified, and suitable techniques must be used to make it easy for the user to complete their coding task, such as generating a call to a nonexistent function. It may also be useful to generate starting stubs for the user to fill in. It is also often useful to provide more general customization techniques for allowing unforeseen modifications, such as the double-derived technique discussed earlier.

When designing the diagrammatic structure of a DSL, it can be useful to take inspiration from the conventions established by the Unified Modeling Language (UML). For example, if there is an inheritance-like concept in the DSL, it would probably be perverse to represent it other than using an open triangle pointing at the more general element. For this reason, the starting languages offered by the DSL Tools are based diagrammatically on those of UML, although they use simpler underlying domain models. In our experience, the popularity of UML lies primarily in the fact that it offers a standard set of diagrammatic conventions, and not in the details of how these conventions are implemented.

Another approach to designing a DSL, for users who already have a UML tool, is to use UML itself as a starting point. By decorating the UML elements with stereotypes and tagged values, their meanings can be modified to correspond more directly to the desired domain. This approach can be successful in domains that are close to the intended meaning of the UML elements, but does make the creation of code generators and validation tools considerably more complicated than the simpler approach of designing a purpose-built domain model for the desired language.

DSLs and Software Factories

Although DSLs can be useful as a stand-alone tool, especially in very constrained domains, their use is most compelling as part of a complete software factory. You may think of a DSL as a software power tool, which is put together with other tools, guidance, and automation to constitute a complete factory. The software factories vision is explained by other articles in this issue, and in the popular book by Greenfield and Short (see Resources).

When a DSL is deployed as part of a factory, it must be integrated deeply with the other factory components. For example, menus and other UI gestures within the language may launch tools and actions associated with other parts of the factory. Conversely, menus and gestures within other parts of the factory might launch, or otherwise interact with, the DSL. In consequence, the entire factory should appear to its users as a seamless whole, intended for solving the user's problem, rather than a ragbag of loosely-integrated tools.

To enable these integrations, it is important for the DSL to offer powerful, dynamic integration points, such as the ability to run (and undo) commands that act on the model and its associated artifacts, the generation of simple APIs for interacting with the models, and the serialization of models in XML files that enable processing by readily-available tools. All of these factors have been taken into account in designing the DSL Tools, which are part of the overall software factory platform and authoring environment. •

Resources

Domain-Specific Modeling (DSM) Forum
www.dsmforum.org

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools
Jack Greenfield, et al. (Wiley 2004)

About the Author

Steve Cook is a software architect in the enterprise frameworks and tools group at Microsoft. He is one of the designers of the Domain-Specific Language Tools in the Visual Studio SDK. Previously he was a distinguished engineer at IBM and represented them in the specification of UML 2.0. He has worked in the IT industry for more than 30 years, as architect, programmer, consultant, author, researcher, and teacher. He is a member of the editorial board of the *Software and Systems Modeling* journal, a fellow of the British Computer Society, and holds an honorary doctor of science degree from De Montford University.

Measuring Success with Software Factories

by Marcel de Vries

Summary

Software factories and Visual Studio Team System (VSTS) can be used together to improve quality, predictability, and productivity of software projects. Using the VSTS data warehouse and reporting capabilities, the software factory builder can determine reliably which aspects of product development need improvement and how to modify the software factory to improve them. The author assumes you already know software factory nomenclature and concepts like viewpoints, views, factory schema, and factory template.

Building software today is hard. Systems get more complex and larger every day. We face rapidly changing technology while trying to keep pace with the demands of business customers who want us to write more software better and faster. Is it really possible to be more productive while producing better quality software? Can greater productivity be sustained across maintenance and upgrades without degraded quality or significant rewriting?

Many of these problems arise because we learn too little from the projects we have done. Few teams regularly reuse solutions or keep track of the things that went well and the things that went wrong. As a result, there is not enough knowledge transfer between projects. Lessons already learned are relearned by new developers. Since most projects fail to deliver on time and within budget, we can see that we also have a predictability problem.

It is possible to build software on time, within budget, and with adequate quality. However, there must be an organizational awareness that the current approach to building software is grossly inefficient. Without awareness of existing problems there will be no drive to improve. To start building software systems predictably, we must make a cultural change. We need to make it easier for practitioners to know what to do, when to do it, why to do it, and how to do it—and we must automate more of the rote and/or menial aspects of their work.

What we are talking about is industrializing software development, applying techniques long proven in other industries to our own industry, in the hope of making things better for our customers and ourselves.

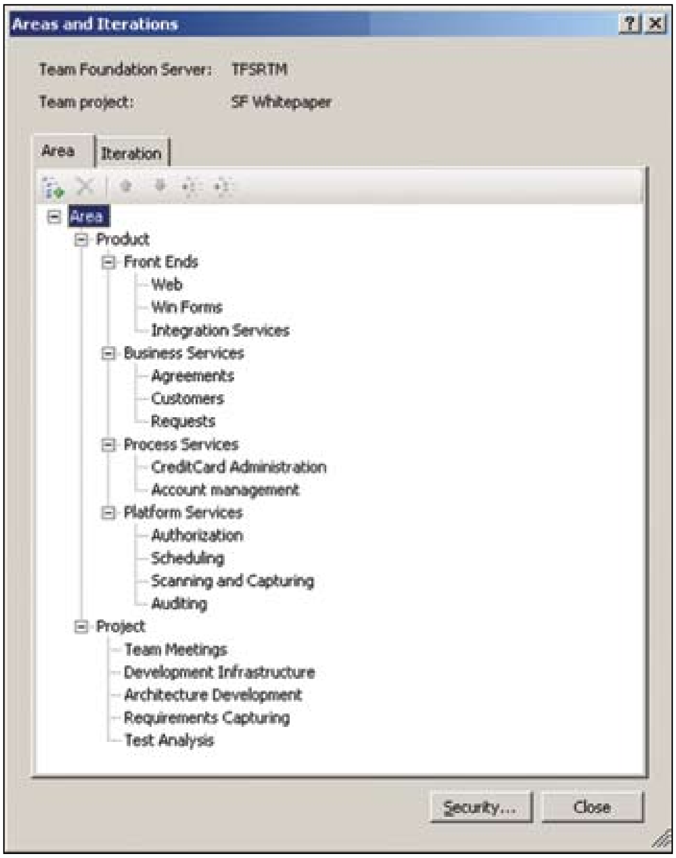
Quality and Productivity Measures

As it turns out, the factory schema provides a useful mechanism for organizing metrics. Since each viewpoint targets a specific aspect of the software-development process, we can use viewpoints to define targeted measures of productivity and quality. Using those measures, we can gather data for specific aspects of the software-development process. By analyzing the data, we can then determine which viewpoints need to improve, how to improve them, and what we can gain by improving them.

To implement this approach, we need a way to express product size, time and budget spent, and product quality to be able to quantify predictability, productivity, and quality for each viewpoint. By measuring each viewpoint, as well as overall factory performance, we can determine how each viewpoint affects overall factory performance, and therefore how much to invest in better supporting a given viewpoint.

For example, we might provide simple guidelines for viewpoints that do not significantly affect overall efficiency, and sophisticated domain-specific language (DSL)–based designers for viewpoints that do. This

Figure 1 An area definition that reflects viewpoints



Team Foundation Server: TFSRTM

Team project: SF Whitepaper

- Area
  - Product
    - Front Ends
      - Web
      - Win Forms
      - Integration Services
    - Business Services
      - Agreements
      - Customers
      - Requests
    - Process Services
      - CreditCard Administration
      - Account management
    - Platform Services
      - Authorization
      - Scheduling
      - Scanning and Capturing
      - Auditing
  - Project
    - Team Meetings
    - Development Infrastructure
    - Architecture Development
    - Requirements Capturing
    - Test Analysis

process helps us get the best return on investment in terms of predictability, productivity, and quality. It helps us compare the results to the goals set initially before we started factory development.

One of the aspects of software development we need to improve is productivity. However, to quantify productivity we need a metric that we can use to express productivity in terms of software product volume built in a span of time. When we are able to predict the size of the system and to measure product-size growth during development, we can better predict the time required to complete the project, and we can measure productivity in terms of hours spent per unit of product size. By measuring the growth and size, we are able to identify differences between the actual and planned values and to start analyzing and managing the differences when they become apparent.

At this point, you may be wondering how we can predict product size and growth with enough accuracy to make this kind of measurement and analysis useful. It certainly does not seem possible if we are developing arbitrary applications one project at a time. If we are using a software factory, however, we have two advantages that significantly improve predictability.

First, we are developing a member of a specific family of products with known characteristics, not just an arbitrary application. Because a factory allows us to describe a product family and its salient features—and more importantly to refine that description as experience is gained over the course of multiple projects—we know much more about an application being developed using a factory than we do about an arbitrary application.

Second, we are developing the application by applying prescriptive guidance supplied by the factory. By standardizing the way we do some things, a factory tends to remove gratuitous variation from the development process, making it much more likely that product size and growth will follow similar patterns from one application to the next.

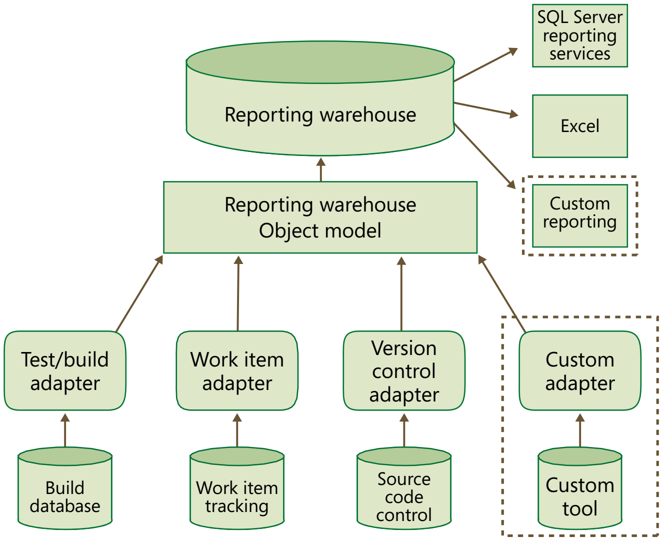
Quantification Standard

If we want a metric that can help us express size and productivity, we need an objective quantification. This objective quantification can be accomplished by using a method that is standardized. One of those methods is functional size measurement as defined in the ISO 24570 standard. This ISO standard uses function points as a way to express the size of the software system based on functional specifications. It specifies a method to measure functional size of software, gives guidelines on how to determine the components of functional size of software, specifies how to calculate the functional size as a result of the method, and gives guidelines for the application of the method. These function points can be considered as a "gross metric" to determine the size of a system and to estimate effort and schedule. During development this metric can be used to determine whether the project requires more or less work relative to other similar projects.

Function-point analysis leverages the knowledge of building database-oriented applications and can be applied whenever we build a system that uses data manipulation in a database. Function points are calculated around the knowledge of the number of estimated tables our application will have and the number of data-manipulating functions as data retrieval and data update functions. From this result we can calculate the number of function points that expresses the size of our product.

Once we have expressed our estimated product size, we can learn how much time it takes to implement one function point or even use historical data already available to make predictions on how much time it should cost to implement a function point. A software factory can influence the time spent to implement a function point (productivity), the number of defects per function point (quality), and the accuracy of our estimations.

Figure 2 Team Foundation Server's data warehouse architecture



For example, suppose we applied function-point analysis and determined that the system we are going to build has an estimated size of 500 function points. As we start building this system, we can determine that it takes 6500 hours to build. From that result we can express our productivity as 13 hours (h)/function point (fp).

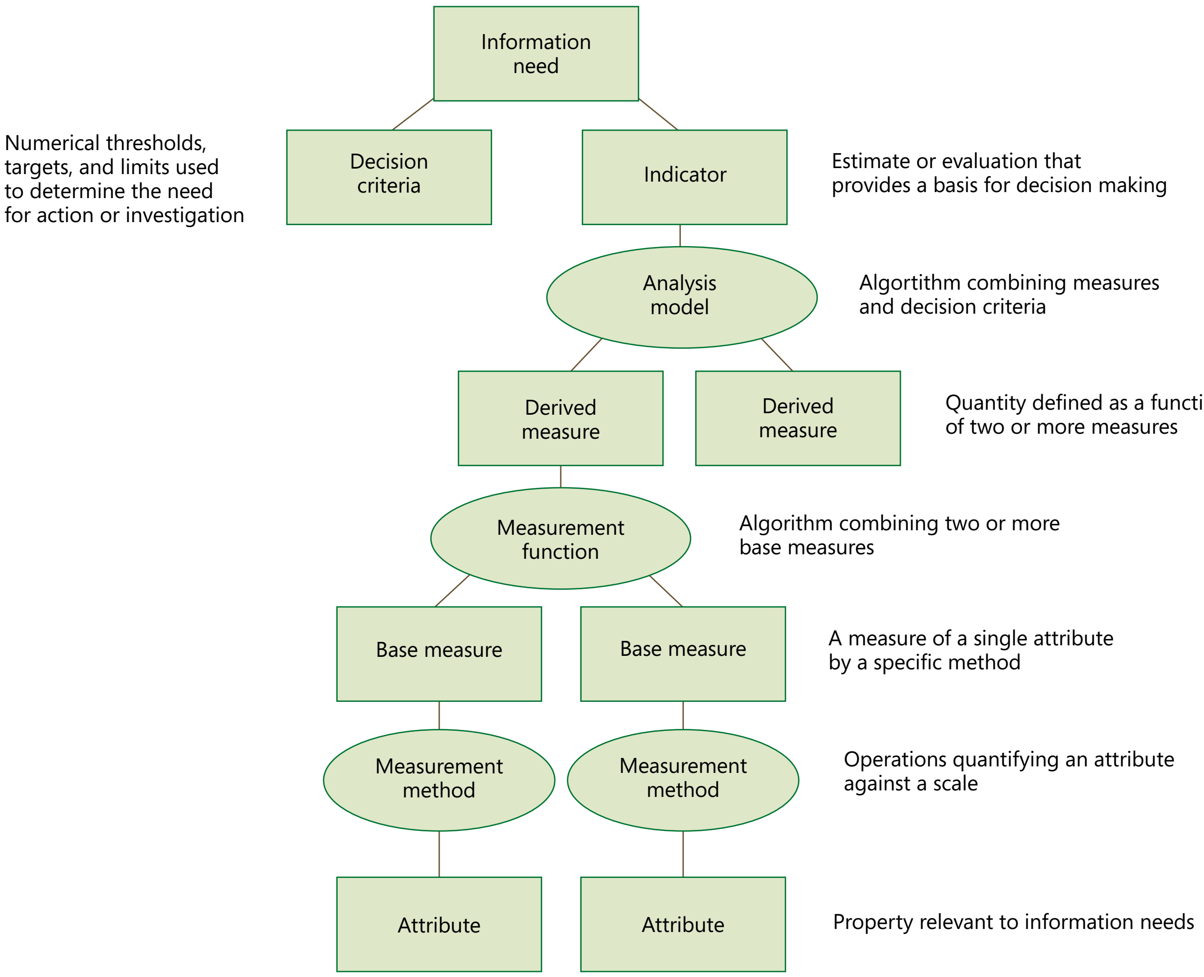
If we also keep track of the defects we found in the product during development, user acceptance test, and production, we can also express that number as a quality metric. Suppose we found 500 bugs during development, 50 during the acceptance test, and 5 after going into production. We could express this calculation as having 1 defect/fp during development, 0.1 defect/fp at acceptance test, and 0.01 defect/fp in production.

It gets really interesting when many of these defects can be traced back to a specific viewpoint of your factory. From that discovery we learn that the viewpoint has a high contribution to the overall number of defects, and we can focus our attention and analyze what might need improvement within this viewpoint. From this kind of analysis, we can determine which viewpoints to improve and how to improve them to reduce the number of defects the next time the factory is used.

The great thing about having a quantification of the number of defects against a metric such as function points is that we now can set goals for the improvements we want to achieve by our investments. For example, I want the number of defects/function point to go down by 20 percent for the "front-end applications" viewpoint. Performing defect and function-point analysis on a per-viewpoint basis gives us a powerful tool for improving our product development process because it helps us determine where the bottlenecks lie, and therefore where to invest and how to invest to obtain better results.

When we start using function points, we can initially use historical data from surrounding organizations found in the literature to do our first estimations. Historical data is useful because it accounts for organizational influences, both recognized and unrecognized. The same idea applies to the use of historical data within the software factory. Indi-

Figure 3 The structure of a measurement construct


Information need
Decision criteria
Indicator
Numerical thresholds, targets, and limits used to determine the need for action or investigation
Estimate or evaluation that provides a basis for decision making
Analysis model
Algortithm combining measures and decision criteria
Derived measure
Derived measure
Quantity defined as a function of two or more measures
Measurement function
Algorithm combining two or more base measures
Base measure
Base measure
A measure of a single attribute by a specific method
Measurement method
Measurement method
Operations quantifying an attribute against a scale
Attribute
Attribute
Property relevant to information needs

vidual projects developed using a software factory will share a lot with projects developed using the same software factory. Even if we do not have historical data from past projects, we can collect data from our current project and use it as a basis for estimating the remainder of our project. Our goal should be to switch from using organizational data or industry average data to factory data and project data as quickly as possible (see Resources).

Apply Visual Studio Team System

Now consider how to enable our product development team to use the factory to create the required work products. This ability starts with a development environment that supports the whole product life cycle from birth to discontinuation, such as Visual Studio Team System (VSTS). Using VSTS is a key to enabling our product development teams to benefit from the approach described previously.

Currently, VSTS does not understand software factories. However, because VSTS is so configurable and extendible, we can set it up manually to support a software factory by mapping various parts of the factory schema onto various configuration elements or extension points.

Recall that a software factory contains a schema that describes its organization. The factory schema defines a set of interrelated viewpoints, and each viewpoint describes related work products, activities, and assets for users in a specific role. We can use this information to configure VSTS for developing applications.

A viewpoint can be mapped to a concept that VSTS calls an *area* in one or more iterations. The role associated with a viewpoint can be mapped to one or more VSTS project roles. In practice, multiple viewpoint roles will probably be mapped to a single VSTS project role. The activities defined by a viewpoint can be added as work items in those areas at project creation, and directly assigned to the appropriate role. They can also be documented by customizing the process guidance, and custom work items can be created to track them and to link them to work products.

Content assets, such as guidelines, patterns, and templates can be added to the project portal document libraries. Executable assets, such as tools and class libraries, can be placed in the version control system. To measure and improve the performance of our factory, we can add metrics to the VSTS data warehouse.

The keys to configuring VSTS are the Project Creation wizard and the process template. The Project Creation wizard is a tool for creating projects in Team Foundation Server. It uses a file selected by the user called a *process template* to configure the server for the project. The template contains several sections, each describing the way a specific part of the server will be configured. With the process template, for example, we can define work item types, areas, iterations, and roles and assign the appropriate rights to each role; customize version control; set up the project portal; and do many other things to customize the development environment and the development process.

Process Configuration

VSTS uses work items to track the work that needs to be done to create a given product. *Work items* describe the work that needs to be done, identify the party accountable for that work at a given point in time, and can be of different types designed to describe different kinds of work. For example, a bug can be described by a work item of type Defect that contains information pertinent to fixing a bug, such as the description of the bug, reproduction steps, estimated time to analyze or fix the bug, and so on. Work item types are created or modified by changing the XML definitions loaded into the server and used at the time the project is created. They can also be modified after project setup.

Work items can be linked to a so-called area of a project and to an iteration. *Areas* provide a way to book the work on a specific part of the solution that is of interest when we want to run reports on the data accumulated in the data warehouse. Areas in VSTS closely match the concept of viewpoints in a software factory, as both represent areas of interest or concern.

When we map areas of interest in tracking a work item to our factory viewpoints, we can use these metrics to provide the productivity and quality measures for specific viewpoints.

One very good starting point in defining viewpoints for a factory is a set of common viewpoints that tends to appear in many factories. Two of those common viewpoints that prove particularly useful in configuring VSTS are System Engineering and Project Engineering. In the System Engineering area we can make a subtree containing the architectural viewpoints that describe salient parts of our system. This description will help us identify which parts of the system have the most significant impact on productivity (time spent) and quality (number of defects). The Project Engineering area is also interesting because it can help us find anomalies in the way activities have been formalized in the project, and it can help us decide whether or not to improve the process definition at certain points. Figure 1 shows an example of areas and iterations that reflects the schema for a simple factory that builds service-oriented administrative applications with multiple front ends.

The area tree can become pretty deep if we try to incorporate every viewpoint defined by our factory. It is very important that we do not explode the tree into many different levels. Keep in mind that it needs to be very simple, allowing team members to easily identify the areas to which work items should be linked. The more deeply nested the tree, the harder it becomes to find the right area for a given work item. If it becomes too hard, developers will simply book work items near the root of the hierarchy, defeating the purpose of creating a deeply-nested tree.

The Team System data warehouse keeps track of all kinds of information about the development of the solution. One section of the data warehouse holds information about work items, which is interesting from a factory perspective, as described earlier. Other sections hold information about tests, daily builds, and other VSTS features. The data warehouse can be extended in two ways to support measurement.

First, we can change the fields kept in the warehouse for a specific work item type by modifying the work item type definition, either by

Figure 4 Base and derived measures for software size growth

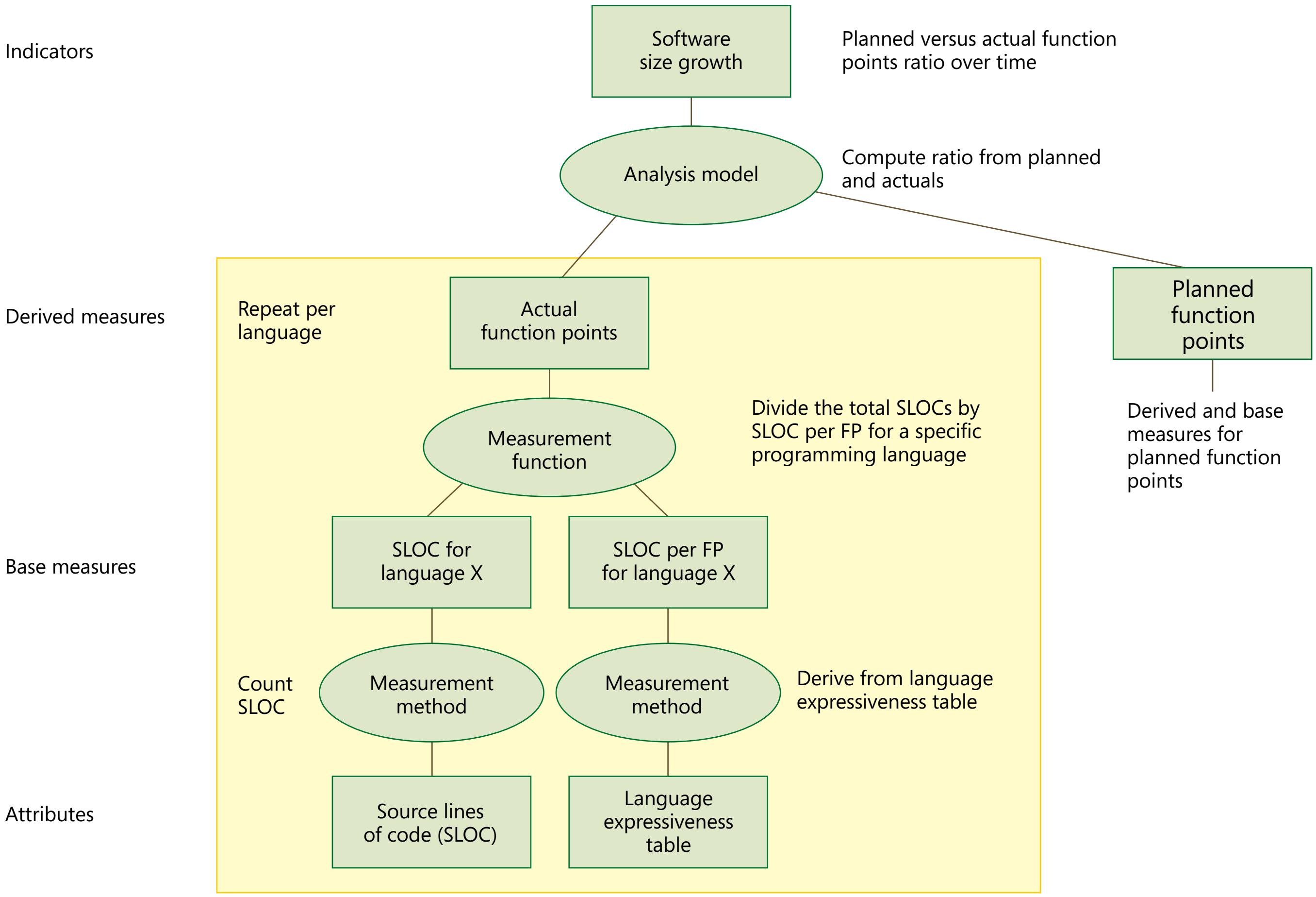
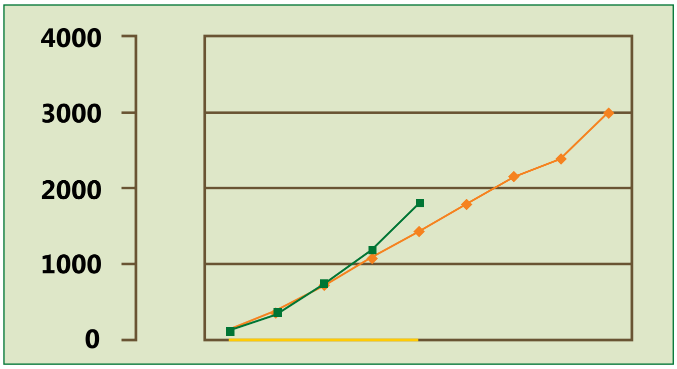

Indicators
Software size growth
Planned versus actual function points ratio over time
Analysis model
Compute ratio from planned and actuals
Derived measures
Repeat per language
Actual function points
Planned function points
Derived and base measures for planned function points
Measurement function
Divide the total SLOCs by SLOC per FP for a specific programming language
Base measures
SLOC for language X
SLOC per FP for language X
Count SLOC
Measurement method
Measurement method
Derive from language expressiveness table
Attributes
Source lines of code (SLOC)
Language expressiveness table

Figure 5 Graphical representation of a planned-versus-actual indicator for software growth



changing the fields it contains or by adding the fields to new facts or dimensions in the warehouse. When a field is marked as reportable in the work item type definition, it will be added dynamically to the data warehouse. Of course, if we want to show reports on these additional fields, we will also need to create reports for the data and upload them to the reporting server to make them accessible to other team members.

Second, we can incorporate data generated by custom tools. If our factory provides custom tools that generate data, and we want to use the data in the data warehouse, we can add a custom data warehouse adapter to the Team Foundation Server (see Figure 2).

For example, to measure the size of each solution in terms of number of lines of code, build a custom tool that counts the lines of code in a file and a custom data warehouse adapter. Also add a step to the daily build that runs the custom tool over the sources in the current solution and places the result in a file. The custom data warehouse adapter then picks up the information from the file and makes calls to the data warehouse object model provided by Team System to add the information to the data warehouse. Custom data can be viewed using custom reports.

Using Measurement Constructs

So far, we have looked at how to define a factory, how to refine a factory using measurement and analysis, and how to configure VSTS to support a software factory. Before we can put all these insights together to build and refine software factories with VSTS, we need to know one more thing—how to collect the right information.

What we need are formal definitions of the relationships between the things we are measuring and the information we need to support refinement. Those definitions are called measurement constructs. *Measurement constructs* are combinations of base measures, derived measures, and indicators. A measurement construct describes an information need, the relevant entities and attributes, the base and derived measures, the indicators, and the data collection procedure.

A *base measure* captures information about a single attribute of some software entity using a specified measurement method. A base measure is functionally independent of all other measures. A *derived measure* is defined as a function of two or more base and/or derived measures. A derived measure captures information about more than one attribute. An *indicator* is a measure that provides an estimate or evaluation by applying an analysis model to one or more base and/or derived measures to address specified information needs. Indicators are the basis for measurement analysis and decision making. Additional rules, models, and decision criteria may be added to the base measures, the derived measures, and the indicators. Figure 3 illustrates the structures of a measurement construct (see Resources).

Key terms on software measures and measurement methods have been defined in ISO/IEC 15939 on the basis of the ISO international vocabulary of metrology. The terms used in this discussion are derived from ISO 15939 and *Practical Software Measurement* (see Resources).

Use these steps to define a measurement construct that we can add to our Team Foundation Server data warehouse.

1. ***Define and categorize information needs.*** To ensure that we measure the information we need, we must understand clearly our information needs and how they relate to the information we measure. Experience shows that most information needs in software development can be grouped into one of the seven categories defined by ISO 15939: schedule and progress, resources and cost, product size and stability, product quality, process performance, technology effectiveness, and customer satisfaction. An example of an information need in the product size and stability category might be: "Evaluate the size of a software product to estimate the original budget."

   These information needs can be used to measure the properties of a specific viewpoint in a software factory. They must be prioritized to ensure that the measurement program focuses on the needs with the greatest potential impact on the objectives we have defined. As described earlier, our primary objective is usually to identify the viewpoints whose improvement will yield the best return on our investments. Since viewpoints can nest, we can often roll up measurements to higher-level viewpoints. For example, if we had a User Interface viewpoint containing viewpoints like Web Part Development and User Authorization, we might roll up the customer satisfaction measurements from specific Web parts to the User Interface level.
2. ***Define entities and attributes.*** The entities relevant to the information need, "Evaluate the size of a software product to appraise the original budget estimate," for example, might be a development plan or schedule, and a *base-lined* set of source files. The attributes might be function points planned for completion each period, source lines of code, and a language expressiveness table for the programming languages used.
3. ***Define base measures and derived measures.*** Specifying the range and/or type of values that a base measure may take on helps to verify the quality of the data collected. In our example we have two base measures, the estimated size of the software product and the actual size. The scale for both base measures will range from zero to infinity. A derived measure captures information about more than one attribute (see Figure 4).
4. ***Specify the indicators.*** To use an indicator correctly, its users must understand the relationship between the measure on which it is based and the trends it reveals. The measurement construct should therefore provide this information for each indicator: *guidelines for analyzing the information*, for our example we might provide an analysis guideline like "Increasing software size growth ratio indicates increasing risk to achieving cost and schedule budgets."; *guidelines for making decisions based on the information*, for our example we might provide a decision-making guideline like, "Investigate when the software size growth ratio has a variance of greater than 20 percent."; and *an illustration of interpreting the indicator*, for our example we might provide an illustration (see Figure 5) and describe

it like this: “The indicator seems to suggest that the project production rate is ahead of schedule. However, after further investigation, it turns out that the actual size of one item was larger than planned because of missing requirements that were not identified until initial testing. Resource allocations, schedules, budgets, and test schedules and plans are impacted by this unexpected growth.”

5. ***Define the data-collection procedure.*** Now that we know how to relate the base measures to the information needs, we must define the data-collection procedure. The data-collection procedure specifies the frequency of data collection, the responsible individual, the phase or activity in which the data will be collected, verification and validation rules, the tools used for data collection, and the repository for the collected data.

Add Constructs to the Data Warehouse

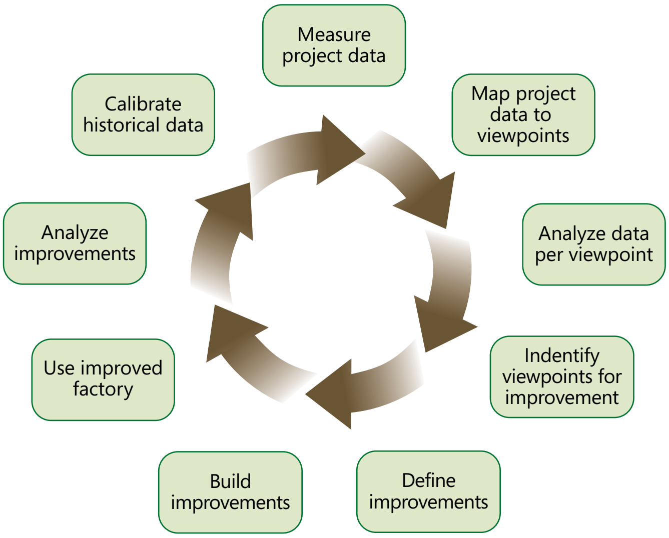
As described, each measurement construct needs to define at least the information needs, the entities and attributes, the base measures and derived measures, the indicators, and a data-collection procedure. To map this to the Team System data warehouse, we need to determine how to obtain the required information, either by modifying work item type definitions to add fields and to mark them as facts or dimensions, or by building a custom tool and a custom data warehouse adapter that collects data produced by the tool. We also need to determine how to display the indicators, usually by creating custom SQL Server 2005 report server reports.

When we have mapped our factory onto VSTS, we can start using it to build solutions. It will guide our team in building the solutions, and it will provide us with information based on the measurement constructs we have defined and implemented.

Once we have a baseline in place with initial data, we can run a continuous software factory development loop that analyzes the performance of each viewpoint, uses that information to determine what to improve, build the improvements, and then repeats the process. This virtuous cycle can be used to target a variety of measures. A key part of this process is estimating the cost of making a given improvement, estimating the gain in productivity likely to result from making the improvement, and estimating whether or not the results justify the investment. After implementing the improvement and incorporating it into the factory, we can measure whether or not it met the goals we set in terms of the reduction in hours/function point (see Figure 6).

The motivation for this discussion is a desire to change the grossly inefficient way we build software today with “one-off” or project-at-a-time development. Our customers see that we struggle to deliver projects on time, within budget, and with the expected features. We can help ourselves and our industry as a whole by capturing the knowledge we gain from experience and transferring it to other projects using software factories. We learned how to define a factory and how to measure its performance in terms of productivity and quality. By quantifying the sizes of the products we build, measuring the time spent to build them, and registering the number of defects found, we can describe the performance of our factories.

Figure 6 An iteration loop for factory development


Measure project data
Map project data to viewpoints
Analyze data per viewpoint
Indentify viewpoints for improvement
Define improvements
Build improvements
Use improved factory
Analyze improvements
Calibrate historical data

The mapping from the factory schema to VSTS is done using the customization and extensibility points in VSTS. We can set up VSTS by placing the assets identified by the factory schema in the version control repository or the Team Foundation Server portal. We can use the portal to provide process guidance for activities described by the factory schema. We can use the Project Creation wizard to arrange the initial setup of our factory, and we can use feature modeling to create a mapping to define forms to add to the wizard. A large portion of the initial project is done using the process templates, and we can modify the templates to support our factories.

By implementing measurement constructs in the VSTS data warehouse, we can gather metrics that describe software factory performance in terms of productivity and quality. Over time we can use these metrics to constantly improve our factories and to gain not only productivity and quality, but also to gain predictability by removing excess or gratuitous variability. The result of implementing software factories with VSTS is more successful projects and greater customer satisfaction. •

Resources

International Organization for Standardization and International Electrotechnical Commission ISO/IEC
www.standardsinfo.net/isoiec/

Practical Software Measurement: Objective Information for Decision Makers, John McGarry et al. (Addison Wesley Professional 2002)

Software Estimation: Demystifying the Black Art, Steve McConnell (Microsoft Press 2006)

Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools, Jack Greenfield et al. (Wiley 2004)

About the Author

Marcel de Vries is an IT architect at Info Support in the Netherlands and Visual Studio Team System MVP. Marcel is the lead architect for the Endeavour software factory targeted at the creation of service-oriented enterprise administrative applications used at many large enterprise customers of Info Support. Marcel is a well-known speaker on local events in the Netherlands including developer days and Tech-Ed Europe. He also works part time as trainer for the Info Support knowledge center. Contact Marcel at marcelv@infosupport.com, and you can read his blog at http://blogs.infosupport.com/marcelv.

A Foundation for the Pillars of Software Factories

by Tom Fuller

Summary

The complexity involved in designing and developing solutions has increased dramatically over the past 30 years. As your application portfolio evolves, there are process strategies that can help your organization overcome the problems that plague software development today. Promoting reusability by adopting production line methodologies will ensure broader success of the systems delivered using these processes. Reusability rarely happens by accident, and using strategic processes like architecture-driven development will make discovery of reusable components an intentional step as opposed to an opportunistic one.

The most valuable artifacts that any architect can produce are those that can be applied across numerous problem domains. This versatility is why patterns, frameworks, guidelines, reference models, and automation tools are core deliverables from any process iteration. Delivery strategies that focus on architecture must extract and apply proven techniques for solving challenging application problems. Architects will typically be embedded at the project level but remain knowledgeable of enterprise scope. Over time, the collection of these extracted best practices forms a library for other teams to use as they compose their solutions.

It is immediately evident that the software factory pillars and the delivery goals of an architecture-driven process are in sync. Using standards-based deliverables, like software factory schemas and pattern languages, to group and describe your enterprise architecture components can take your enterprise architecture to the next level. Capturing reusable artifacts with these templates gives your organization a consistent way to deliver reusability. The methodology will then extend beyond the construction and delivery of the product and focus on post-delivery return on investment measurement, education, and the long-term road map for these deliverables.

Introducing architectural guidance within any software development life cycle will improve the overall effectiveness. It is this process and delivery shift that is at the heart of the software manufacturing revolution. This article explains how to use an architecture-driven process and the software factory pillars to change how you structure teams and deliver solutions. The goals here are to explain what it means to use that process and what type of expected deliverables result from each transition phase of the software development life cycle. Additionally, the article will explain the benefits of intentional discovery, implementation, and measurement of reusable architecture.

Managing Increased Complexity

Anyone who has worked in the software industry will tell you that developing business productivity software is very challenging. Often, companies become overwhelmed by the lack of consistency and bloated costs that typical software cycles create. There are four syndromes that contribute to this increased difficulty:

- *The moving-target syndrome:* This syndrome is an unavoidable aspect in the software industry. There will always be a constant evolution of frameworks, patterns, strategies, and technologies. This continuum will often make your effective solution today an ineffective one at some point in the future.
- *The perfect-storm syndrome:* It is always difficult for any company to find the right number of developers, at the right time, with the appropriate vision and budget. If any one of these items is not correct the solution will suffer and can potentially become ineffective.
- *The Goldilocks syndrome:* One of the joys in solving logical problems is discovering the most innovative and efficient solution for a technical problem. However, this delivery rarely considers the right solution based on cost. As a result, many solutions are overengineered or underengineered. Very often this engineering is a matter of perspective, and striking a balance between the best solution and the *right* solution is challenging.
- *Grandpa's favorite-chair syndrome:* It is in our nature to gravitate toward things that we understand. That is why software is often architected to avoid as much change as possible. The limited risk that comes from using code that has already been proven to work is immeasurable. This approach will result in solutions that are added to in unnatural ways.

Adding to this complexity is the migration from consolidated, monolithic applications to highly-scalable, distributed systems. As the strategy has changed (see Figure 1), so has the focus. Today we need to find an even higher level of abstraction than objects or components. It is based on this need that we introduced repeatable architecture patterns and frameworks. The all-important transition to an architecture-driven process is a catalyst for the discovery and publication of these reusable artifacts.

Software is not the first industry to see its output increase in complexity over time. It only takes a moment to consider how much more complicated current automotive or construction deliverables are now versus 50 years ago. How have these industries managed to meet the high demand for their increasingly complex product? The answer lies in the shift away from pure craftsmanship and toward manufacturing in both instances. By capturing and repeating best practices for building well-known aspects, the limited resources available for development are able to focus on those things that are truly unique.

Based on the preceding information, it is clear that the software industry is faced with a similar dilemma. The complexity has increased, and at the same time the demand has skyrocketed. The concept of software manufacturing is not a new idea. In fact, there are many well-respected software engineers that have been giving it thought since the late 1970s. However, until recently most of the frameworks, patterns, and strategies were still very immature.

We stand at the cusp of a major revolution in how business productivity software is delivered. As businesses begin to use architecture to drive their delivery of software they will find ways to isolate consistent portions of their enterprise applications and capture them in a way that can be reapplied through automation. These concepts provide the best chance to date to help deliver quality software while managing all of the inherent complexity that comes with it.

Define the Vision/Scope for All Process Iterations

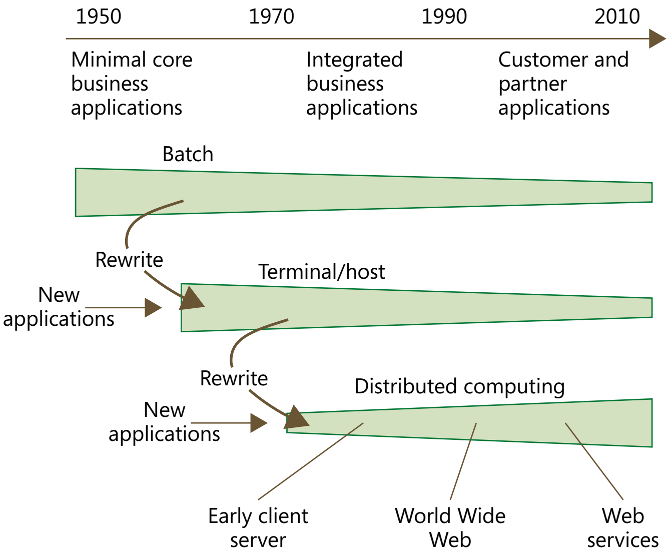
One of the first steps for any organization interested in moving to an architecture-driven process is to define the vision and scope for the enterprise architecture. Without this definition, it becomes almost impossible for solution architects to make good decisions about where and when to introduce architectural patterns. You might think that service-oriented architecture (SOA) is an example of the vision for your enterprise architecture. In reality, SOA is an example of a delivery strategy that can help you to adhere to the vision for your enterprise architecture.

A vision statement should be concise and devoid of any implementation biases. The vision statement should "paint a picture" of where the architecture team wants the applications to evolve to. Here are some examples that could be used to help focus the architecture team as solutions are being delivered using architecture:

- All application deliverables will focus on quality through embracing and extending proven enterprise architecture artifacts. Over time, new solutions should be built completely through composition and customization of enterprise architecture frameworks.
- All application deliverables will efficiently use resources within the enterprise infrastructure to solve business productivity demands. Using tailored tools and processes, 75 percent of a custom application will be constructed, tested, and deployed automatically.

The scope of your enterprise architecture is separate from the vision. What needs to be considered when defining the scope of the enterprise architecture is whether or not the practices and patterns that are being managed by the solution architects are within specific technology disciplines, business areas, or application styles. The broader the scope the more challenging it is to manage complexity. However, if the scope is too narrow, you will risk decreasing the impact your architecture patterns can have. The best way to manage scope is to determine a grouping strategy based on variation. For example, the strategies and patterns of the engineering and infrastructure group may be very different than those of the application-delivery group. As long as there is a shared strategy for how to consistently apply those patterns, managing them separately is acceptable, which starts to show the reason vision and scope for enterprise architecture are so important.

Figure 1 Time-phased trending in application architecture patterns



In most, if not all, organizations it takes a combination of people with different spheres of concern to deliver applications successfully. If the burden of managing the architecture vision and scope is not shared, then the leadership and direction will become fragmented and inconsistent. Other strategists in the organization will spend less time debating issues if synergies are found between the various scoping groups. This synergy is without a doubt one of the most important steps in starting to build architecturally sound applications.

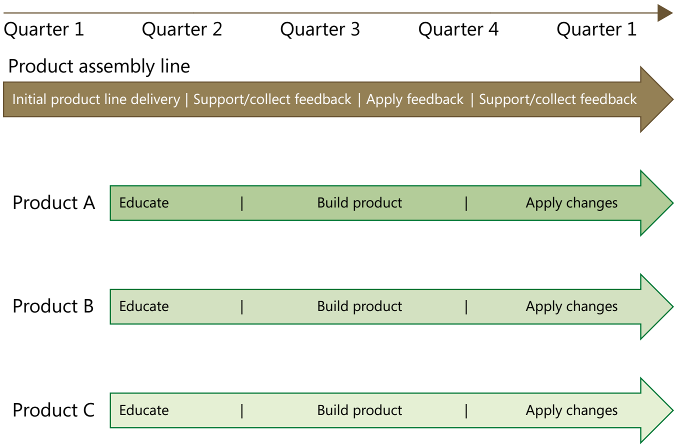
Building Your Assembly Line

An architecture-driven process is focused primarily on shifting the control for delivering solutions to the architecture team. Specifically, the solution architect that is embedded in the delivery team will be responsible for determining how quickly an application can move through the phases of development. This determination is primarily in an effort to work on strategically delivering applications more efficiently in the future. So how then do solution architects "prove their worth" within each of these iterations? This value is where software factories and production line delivery of applications is key. The architecture team is working constantly to construct and improve on the software assembly line within one of the predefined enterprise architecture scope groups.

The key components of a software factory or product line are all focused on one key goal: abstract those portions of the application that do not vary, and guide the creation of variants by using pragmatic constraints. Here are the four core pillars of the software factory initiative (see Resources):

1. *Software product lines:* Architects must focus on how to find those portions of an application that can be abstracted because they are consistent. Once they are discovered they should be delivered ahead of the products that will use them. This approach promotes an intentional step of finding and delivering reusability. In all likelihood these assets will fall in line with the scoping groups defined for your enterprise architecture.

Figure 2 The product line development approach


Quarter 1
Quarter 2
Quarter 3
Quarter 4
Quarter 1
Product assembly line
Initial product line delivery | Support/collect feedback | Apply feedback | Support/collect feedback
Product A
Educate | Build product | Apply changes
Product B
Educate | Build product | Apply changes
Product C
Educate | Build product | Apply changes

2. *Guidance in context:* There are a couple of levels of variability when it comes to components within the software product line. One level is that which can be automatically built "hands free." These components are usually very low level and require essentially no decision making by the product-delivery team. The next level is that which can vary in a controlled way, which is where guidance comes in. When a product developer can choose from a set of constraints to build an application component from a finite set of variations, the software factory should support that. This variability is not the whole picture though. Also consider the "in context" portion of this process. When you can provide context-sensitive guidance there are benefits to be gained from providing something as simple as tailored help!
3. *Architecture frameworks:* Frameworks (often described using a factory schema) within your software factory provide a way to group all of the building blocks that will be used by the product developers. There are a number of components in any software deliverable that are potentially reusable. The framework will capture and deliver best practices in an effective way.
4. *Model-driven development (MDD):* Models provide a mechanism for representing complicated software components using visual abstractions. This mechanism typically helps simplify the design, development, and support of those components. Making models a critical component of your software factory requires you to think differently about design documentation. Models must always reflect the current running code if they are to remain useful in the support and maintenance of software. Historically that has not been the case, and MDD is an effort to fix that shortcoming.

As an enterprise architecture proponent, each of these pillars is critically important. Every product that is delivered will look to extend and/or consume numerous portions of the software factory. As a solution architect these are the tools you bring to the table to help drive every product toward well-established best practices. Without these tools every product has to be built from scratch. This style is often referred to as "one off" and is considered very inefficient.

As an enterprise matures in its architecture-driven practices so do the components of the software factory. Collecting new enhancements and delivering new versions of the factory components facilitates perpetual innovation through the architecture team. The hardest part about transitioning toward an architecture-driven process and a software factory approach is starting. We discussed previously the scope definition for the enterprise architecture. Scope continues to be a basic element of concern for all software factory deliverables. In the case of business productivity applications economies of scope and scale can both be achieved:

- *Economies of scope:* To justify the cost of an architecture component it has to be useful to a number of products. One strategy for identifying components that exhibit economies of scope is to group products based on implementation style. For example, if your applications will be built using a distributed model focusing on Web services across tiers, then it will be beneficial to build a factory component that can help guide the delivery of a Web service. Very often these stereotypes are common across every application. Another possible economy of scope benefit is the delivery of enterprise services. If a large number of applications depend on the same data or business subprocesses, then it may be an ideal candidate for a services-based approach.
- *Economies of scale:* It is very rare to find circumstances where business productivity applications can benefit from economies of scale. If a factory component is going to have an economy of scale your organization will need to benefit from that component being created the same way multiple times. An example could be enterprise data dictionaries that are made available as part of every new data dictionary. If the component can be reapplied "as is," then it can be said to exhibit economies of scale.

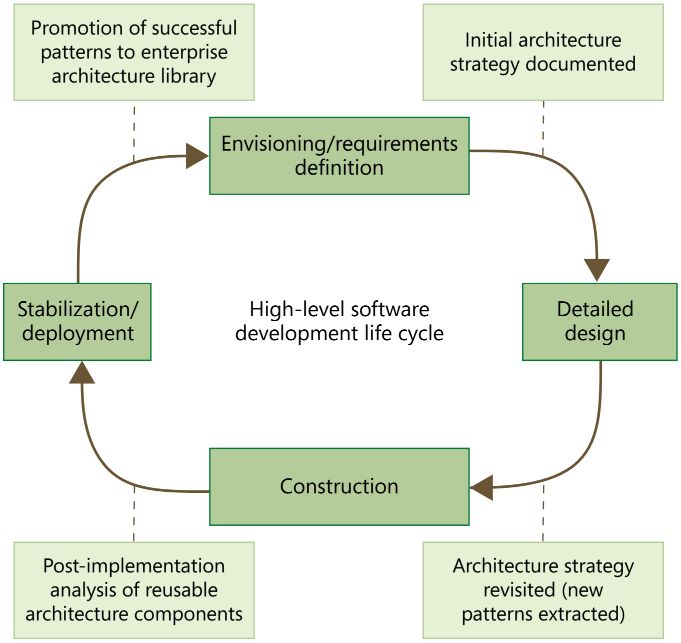
Even when a solution architect is in control of the progress of product delivery, it is still challenging to find reusable factory components. This difficulty is especially true when you consider cost justification. A product line methodology is required to discover factory components with broad enough scope. When a factory product line phase is introduced, the building blocks can be built, and the architecture team can get ahead of the delivery curve. Without this methodology, there are always difficult cost/benefit decisions that determine the architectural direction.

What eventually forms are two separate delivery teams focused on completely different aspects of software. One team is responsible for envisioning, designing, delivering, and training of reusable assets in the scoping group. The other team(s) is responsible for learning, consuming, and giving feedback on those components. The most important item to note in Figure 2 is that the product assembly line team must be given the opportunity to get ahead of the product delivery teams. Once established, it becomes easier to drive work through the feedback loop that will ensue. If the teams begin too much work without a clear understanding of the foundation then it will be impossible to avoid one-off project development.

This concept of grouping applications based on commonality into product groups or families is a critical step in moving toward an architecture-driven process and software factory development. These product families should have dedicated architecture resources and common infrastructure. These families can also help provide consistency in the patterns that are discovered and applied in an enterprise. Typically, the context and forces will remain consistent within a product group. Architects are able to benefit from the lessons learned when applying these solutions to similar application contexts. This consistency translates into effectiveness within the architecture group.

It is impossible to talk about processes or reusability without mentioning agility. Does driving delivery through architecture create an agile process? In the traditional sense, an architecture-driven process would probably not be considered an agile process. However, if you are looking at the fundamental goal of an agile process, architecture-driven pro-

Figure 3 Architecture-driven processes require architecture-focused deliverables during every transition


Promotion of successful patterns to enterprise architecture library
Initial architecture strategy documented
Envisioning/requirements definition
Stabilization/ deployment
High-level software development life cycle
Detailed design
Construction
Post-implementation analysis of reusable architecture components
Architecture strategy revisited (new patterns extracted)

cesses and software factories do translate into higher productivity and adaptability levels. Based on this fact, it is safe to say that driving a process with architecture can foster agility in application delivery.

Always remember that developing a technical solution is simply a series of refinements and abstractions. Depending on your frame of reference you may be attempting to decompose or refine a business problem, or you may be trying to design higher-level abstractions to demonstrate what low-level machine instructions should do with user input. Developers and analysts will always struggle to find the sweet spot when it comes to abstraction and refinement. Too much refinement and time is wasted; too much abstraction and complexity is increased—another application of the Goldilocks syndrome.

In the end, a business productivity application needs to be able to respond to change. Rarely does a product team know the final picture of a solution during the initial release of an application. Embracing change through planning for iterations is paramount to the success of any delivery process. Architecture-driven processes naturally move the product-family delivery team toward iterative development by introducing a feedback loop for the factory components.

Less code typically results in higher productivity. Architecture-driven processes and software factories are built on the concept of abstracting what is already known and guiding what varies, which is a powerful strategy in managing complexity. Consistent and reusable software combined with an up-front understanding that an application will change is what being agile is all-about!

Prerequisites for an Architecture-Driven Process

There are a number of prerequisites before you can begin an architecture-driven process, some have been mentioned previously, and others are listed here as a sort of readiness checklist:

- A role on the project must be focused on the application architecture. This role requires knowledge of existing enterprise architecture patterns and being dedicated to helping deliver applications that meet an enterprise architecture vision.
- A long-term vision for architecture of the enterprise should be established. This vision will help to simplify decision making and support making good decisions that will integrate well with the overall enterprise vision. This vision is articulated through enterprise architecture road maps, business capability matrices, and enterprise framework maturity models.
- Well-defined deliverables must be established for each of the transitions. These can include architecture strategy documents, pattern templates, reusable component analysis reports, and an enterprise architecture library.
- There must be agreement from the project sponsors that transitions among phases cannot take place if the agreed-upon architecture deliverables are not complete. Otherwise, the drive toward expedient delivery can often short-circuit any architecture-driven effort.
- An agreed-upon strategy for resolving architectural anomalies should exist before starting. This agreement will help to mitigate the risk of becoming paralyzed by any lack of "buy in" for the enterprise architecture initiatives. Additionally, this risk-aware approach will help avoid application teams succumbing to antipatterns to meet expedient delivery demands.

Once these prerequisites have been satisfied you can safely begin delivering architecture-focused applications. The tactical introduction of architecture deliverables will ensure that you take the time to proactively build or consume reusable architectural components.

As applications are delivered in any organization, some type of delivery process must be followed. The high-level steps shown in Figure 3 represent those that are commonly found in all development life cycles. It is in the transitions between these phases where architecture should become a focus, and it is this focus that will help your enterprise transition away from one-off and *siloed* application delivery and toward the cohesive development of applications that adhere to an enterprise architecture vision.

Transition 1 (envisioning to detailed design): In this early phase of the project it is critical to start looking for already existing architectural assets (patterns, services, framework components, and guidelines) that can be consumed by the new application. This search will bring to the surface questions about availability, performance, and maturity of these existing components. As an architect, the focus on reuse should help to drive the initial architecture strategy documentation. Delivering a plan that helps the application deliver a high-quality application that leverages as much of the existing enterprise architecture as possible is how you measure your success. Deliverables might include:

- Architecture strategy overview: an enterprise architecture component consumption report, expected enterprise architecture variants, and planned architecture pattern usage report
- Service-level change requests for existing components
- Recommendations for new enterprise architecture components

Transition 2 (detailed design to construction): Once the initial strategy is in place the detailed design can be built to realize the high-level architecture vision. Through a series of refinements, the architecture strategy is either adopted or modified based on the context in which it is applied, which is where the architect role on the project becomes amplified. Helping the project team to make practical decisions about the modifications to the enterprise architecture standards should be his or her focus. In most cases the existing standards should be used as is to avoid inconsistency and added complexity.

It is not always possible to use the existing architecture assets in their current state, and revision requests are sure to be needed. There are also new architectural patterns and styles that can be discovered during this phase. Key deliverables during this transition include architectural component change requests, a new architectural pattern definition, recommendations for new enterprise architecture components, and a revised architecture strategy overview.

Transition 3 (construction to stabilization): Once the application has been built, the focus shifts to quality and post-implementation analysis. To continue to gain widespread acceptance of the architecture patterns being applied, the successes and failures should be documented and communicated to the sponsors of the application. As the patterns being used mature, the likelihood of failures diminishes. The goal here is to show how much time was saved and how much quality was introduced by the focus on architecture. Deliverables include a reusable, assets-consumed overview; a change/extension cost analysis; and an existing asset improvement report.

Transition 4 (stabilization to next iteration): An effort to communicate best-of-breed solutions back to your company is a critical part of being focused on improvement. Once the application has stabilized, the architect should go through an exercise that helps to educate the enterprise to the new patterns and antipatterns discovered during this application's life cycle. Promoting best practices and cross-training other project teams are the only ways to ensure perpetual knowledge growth in your organization. The key deliverables in this phase are mainly for enterprise education, and they include a revised, enterprise architecture library catalog; a cross-team, architectural best practices session; and training materials for new architectural patterns.

Taking the Next Step Toward Industrialization

Once an organization has practiced architecture-driven delivery and considers the process mature, there are steps that can be taken to automate the delivery of applications. All of the architectural components that are delivered as part of the product line are ideal candidates for automation. These components will quickly go through a number of iterations and their consistency will become clearer as the product line team has to adapt to product-level variations.

Not until you understand these variations and establish the constraints should you remove the code completely from the product developers. Understanding what to abstract and automate is complicated and requires some level of trial and error. Many code-generation techniques appear on the surface to be beneficial, but always remember that a developer's confidence in generated code can be lost in an instant. Managing this perception requires a partnership with product developers through some process iterations and guided adoption of best practices.

This next step requires sophisticated tool support. These tools must be capable of providing an open API for developing model-driven tools, integrated wizards for guidance, and context-based capabilities to seamlessly incorporate architecture best practices into the product developers' workspace. These tools have the ability to change the productivity levels of application developers immensely.

Over time the percentage of well-known application components will increase and in parallel so will the automation benefit. Architects will always be attracted to automation for architecture components, but be careful to first understand what it is that you are automating. Most modern tools do a great job at creating separation between automated tool output and custom code, but that separation does not fix the issue of perception or accuracy.

To maintain momentum during product delivery you cannot solve the same problems over and over. The cure comes in the form of creating and guiding consumption of codified architectural best practices. Adopting a process that is driven by architecture will shift application delivery into a more proactive "prepare for the future" mindset. With the increased complexity of and demand for business productivity applications comes a need to transition away from one-off development. Most, if not all, engineering disciplines have learned this lesson and will mimic the success of previous iterations. The application development world is no different. An architecture-driven process will facilitate the collection of the key building blocks of productivity.

The seemingly unattainable goals of software industrialization are quickly becoming a reality. The software factories movement makes great strides in giving companies a way to measure the improvement or value proposition for architecture. Once a repeatable artifact is discovered its *replayability* and *variability* have to be evaluated before absorbing the cost of building a guidance package or a designer. Discovering consistency, increasing productivity, and embracing change are the backbone of agility in application development. Combined, architecture-driven delivery and software factories will help lead us toward the next generation of software development. •

Resources

"Combine Patterns and Modeling to Implement Architecture-Driven Development,"
www-128.ibm.com/developerworks/ibm/library/ar-mdd2/

Japan's Software Factories: A Challenge to U.S. Management, Michael Cusumano (Oxford University Press 1991)

Research Directions in Software Technology, Peter Wegner, "Conference Proceedings, 3rd International Conference on Software Engineering" (MIT Press 1978)

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Jack Greenfield et al. (Wiley Publishing 2004)

Software Product Lines, Paul Clements and Linda Northrop (Addison-Wesley Professinal 2002)

About the Author

Tom Fuller is CTO and senior SOA consultant at Blue Arch Solutions Inc. (www.BlueArchSolutions.com) and an architectural consulting, training, and solution provider based in Tampa, Florida. Tom was recently given the Microsoft Most Valuable Professional (MVP) award in the Visual Developer – Solution Architect discipline. He is also the current president of the Tampa Bay chapter of the International Association of Software Architects (IASA), holds a MCSD.NET certification, and manages a community site dedicated to SOA, Web services, and Windows Communication Foundation (formerly "Indigo"). He has a series of articles published recently in *The Architecture Journal*, *Active Software Professional Alliance*, and *SQL Server Standard* publications. Tom also speaks at many user groups in the southeastern U.S. Visit www.SOApitstop.com for more information, and contact Tom at tom.fuller@bluearchsolutions.com.

A GSI’s Perspective of Software Factories

by Steve Eadie

Summary

With the growing expense and complications of building software, we are continually seeking to increase our productivity while maintaining user satisfaction, but that goal is a global concern that is valid to just about any company of any size. How does a Global Systems Integrator (GSI) manage to become more agile and cost effective, while its developers may be spread across many geographic regions? How does a company know what it knows? And more importantly, how does it leverage this knowledge in a consistent way? These may seem like very strange questions and ones that would make a smaller company do a quick pulse check and wonder, “what’s the problem?” The truth is that the larger the company, the more assets are collected and the easier it becomes to overlook these assets in a rush to solve a client’s problems. We fail to build systems in a consistent manner. Although we can take some amount of pride when a system is delivered on budget and on time, how many projects fail even though *we know what we know*? Let us look at ways in which a GSI can take advantage of a software factory approach.

In today’s software development landscape the story of overruns, defects, security holes, and project failures is all too common. Software development is expensive and labor intensive, and we still end up building one-off applications, hand-stitched with minimal reuse. We already know how to exploit economies of scale to automate production; however, this exploitation does nothing for software development. It is also possible to exploit economies of scope. By reusing designs and components we can build many similar but distinct systems.

Today there are a few universal needs in our changing landscape. We have a need to deploy applications/solutions at an ever-increasing pace with solutions that are growing evermore complex as our clients demand richer features. We need to connect applications/systems in increasingly unprecedented ways, master new technologies, and as always gain a competitive edge. We need to figure out how to take a requirement and have it built into a quality system anywhere on the planet with total confidence that our customers’ expectations of quality, cost, and reliability will be met. The business drivers that are pushing us toward a paradigm shift are:

- Greater agility
- Faster time to market
- Greater productivity
- Better and sustainable quality
- Greater efficiency
- Reduced Complexity

Software factories attempt to address this paradigm shift.

Four Pillars of Software Factories

First, let us define software factories. Jack Greenfield of Microsoft defines a *software factory* as: “a highly customized development environment and technology platform supplying key artifacts and processes to accelerate common life-cycle tasks for a specific type of solution or product” (see Resources).

Four “pillars” comprise a software factory: software product lines, architecture frameworks, model-driven development, and guidance in context. These pillars combine to become more than the sum of their parts to propel us toward software industrialization (see Figure 1).

Software factories are not a new idea but are more the pulling together of several initiatives under one roof and providing the tooling for this methodology. Work has been done in the area of software product lines by Paul Clements and Linda Northrop at the Software Engineering Institute (SEI); model-driven development, specifically

Figure 1 The four pillars of a software factory

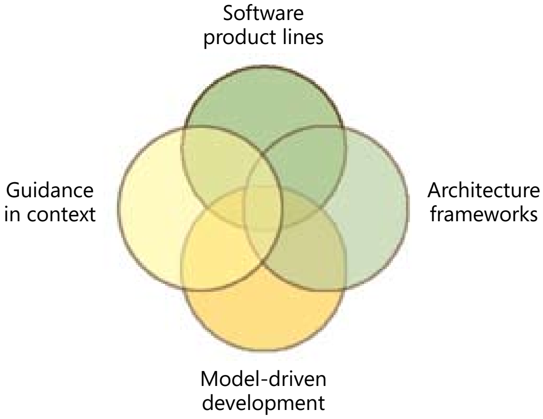
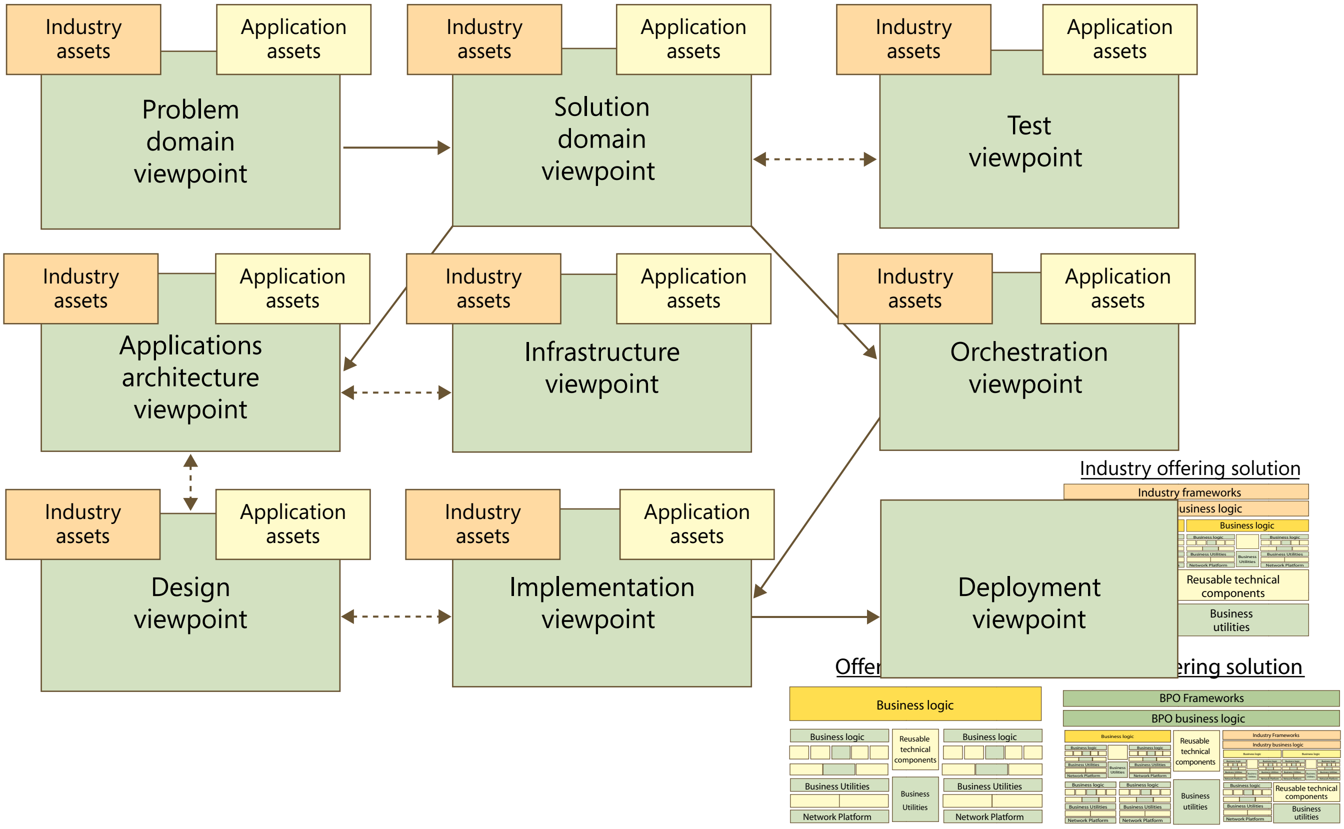


Figure 2 Assembling a client solution from viewpoints



domain-specific languages (DSL), by Microsoft; generative programming by Krzysztof Czarnecki and Ulrich Eisenecker; architecture frameworks from the Microsoft Patterns and Practices Group; and guidance in context, using Microsoft's Guidance Automation Toolkit (GAT).

A *software product line* as defined by Paul Clements and Linda Northrop in their book, *Software Product Lines* (see Resources), is: "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (see Resources).

They go on to tell us that software product lines are designed to take economic advantage of commonality in systems. Product lines also amortize the investment in core assets such as:

- Requirements and requirements analysis
- Domain modeling
- Software architecture and design
- Performance engineering
- Documentation
- Test plans, cases, and data
- Knowledge and skills
- Processes, methods, and tools
- Budgets, schedules, and work plans
- Components

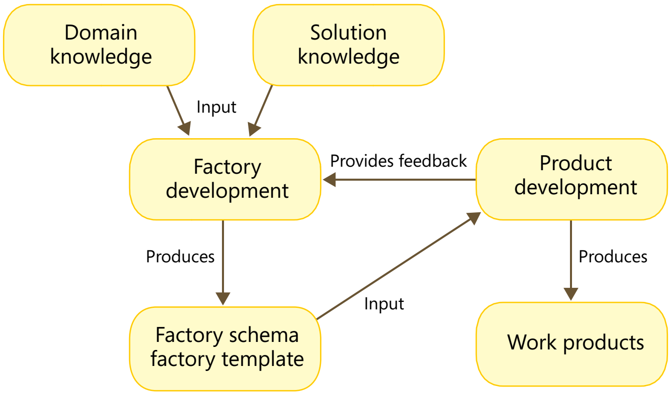
There are three essential activities in fielding a product line: core asset development and product development using those core assets, and both of these are supported by the third activity—technical and organizational management. The core assets can be existing mined assets or assets built from scratch for the product line, the core assets with associated processes for each asset along with production requirements, a product line scope, and a production plan that are then used to build the product.

Architecture frameworks define viewpoints that identify and separate the key stakeholder concerns. They organize tools, process, and content by that viewpoint. They relate and integrate life-cycle phases, system components, and levels of abstraction. The architecture framework is also often described as the schema or process framework for the software factory. This framework can be used to guide the workflow and activities for the project until all of the activities are complete.

"SOFTWARE FACTORIES ARE NOT A NEW IDEA BUT ARE MORE THE PULLING TOGETHER OF SEVERAL INITIATIVES UNDER ONE ROOF AND PROVIDING THE TOOLING FOR THIS METHODOLOGY"

Model-driven development (MDD) is a technique to use metadata captured in models to automate software development tasks. Modeling software has long been regarded as one of the best ways to build a system, as it used to capture the intent of the system.

Figure 3 The factory life cycle



However, models are often discarded and have been seen as second-class citizens compared to code. In software factories models are used to formally express the artifacts of a certain domain and to capture the intent that tools can interpret to provide more advanced forms of automation.

They can be used to create highly-focused, custom languages for specific problems, platforms, or tasks. Metadata that is captured by models is used to drive automation rather than referenced only once in a while. In software factories MDD is handled using the domain-specific language (DSL) toolkit. DSLs allow you to create a modeling environment that describes accurately the product being built and also ensures a more accurate translation of the model by team members, which translates into the code generated from that model.

Guidance in context is used to describe what to do and to provide help in doing it. The guidance is partitioned for common use cases, and is attached to steps in the process and to parts of the architecture. This guidance is context sensitive, and it only appears when you need it. Tooling such as GAT allows installable packages containing organized sets of configurable guidance assets for common use cases. The guidance also can have preconditions and postconditions to allow the workflow to vary subject to constraints.

What Does a Software Factory Mean to a GSI?

Now that we have established what it is, what does a software factory mean for a GSI? Pretty much we are in a time of offshore/best-shore development, where the goal for large companies is to reduce their costs to remain competitive. In reducing these costs the company will move their clients' work across the globe—hopefully without sacrificing quality. This transfer, however, is not always as easy as it seems. Architects and industry experts are a scarce resource. It can certainly be seen that there are not enough experts to go around. To make the project successful, or to bail out of a project that is in trouble, we deploy these experts to where the work is being done. This deployment is not the best use of their time, since once the development is under way they should be moving on to the next big thing on the company's horizon. Instead we find them hand holding a potentially unfamiliar development team far away from home or the client.

Software factories are a way to capture that expert knowledge and distill it so that it is available wherever the work is taking place. Our architect's requirements are baked into the integrated development environment (IDE). We have guidance, best practices, and reference implementations, test cases, and so on for the product our development team is building. The development team not only knows exactly what steps it has to take, but how and when to take them. They also get the expertise gained from every product built before using that particular factory, which adds up to quite a lot of experts being sent to help a project.

A software factory can be viewed as a series of viewpoints, and at each one of those viewpoints we can associate assets such as common services, process, guidance, test plans, and so forth. For a GSI we have these assets distributed among many teams across the globe, but by pulling together the best of those assets to populate each viewpoint our factory starts to take shape (see Figure 2).

Take a look at SharePoint Web Parts development, for example. It is possible to capture all of a company's knowledge on this site and distill that into a factory by taking the best practices, guidance, reference implementations, and test cases and placing them within an applicable viewpoint (the test viewpoint and test cases, for example). Note that if these assets do not exist then they can be mined by accurately mapping a current project and building up a first iteration of the factory. Once built, the factory can then be tested on subsequent projects and refined and extended through real-world use. Subsequent iterations of the factory will be able to reduce pain points in using the factory as well as mine rote tasks and automate those tasks (see Figure 3).

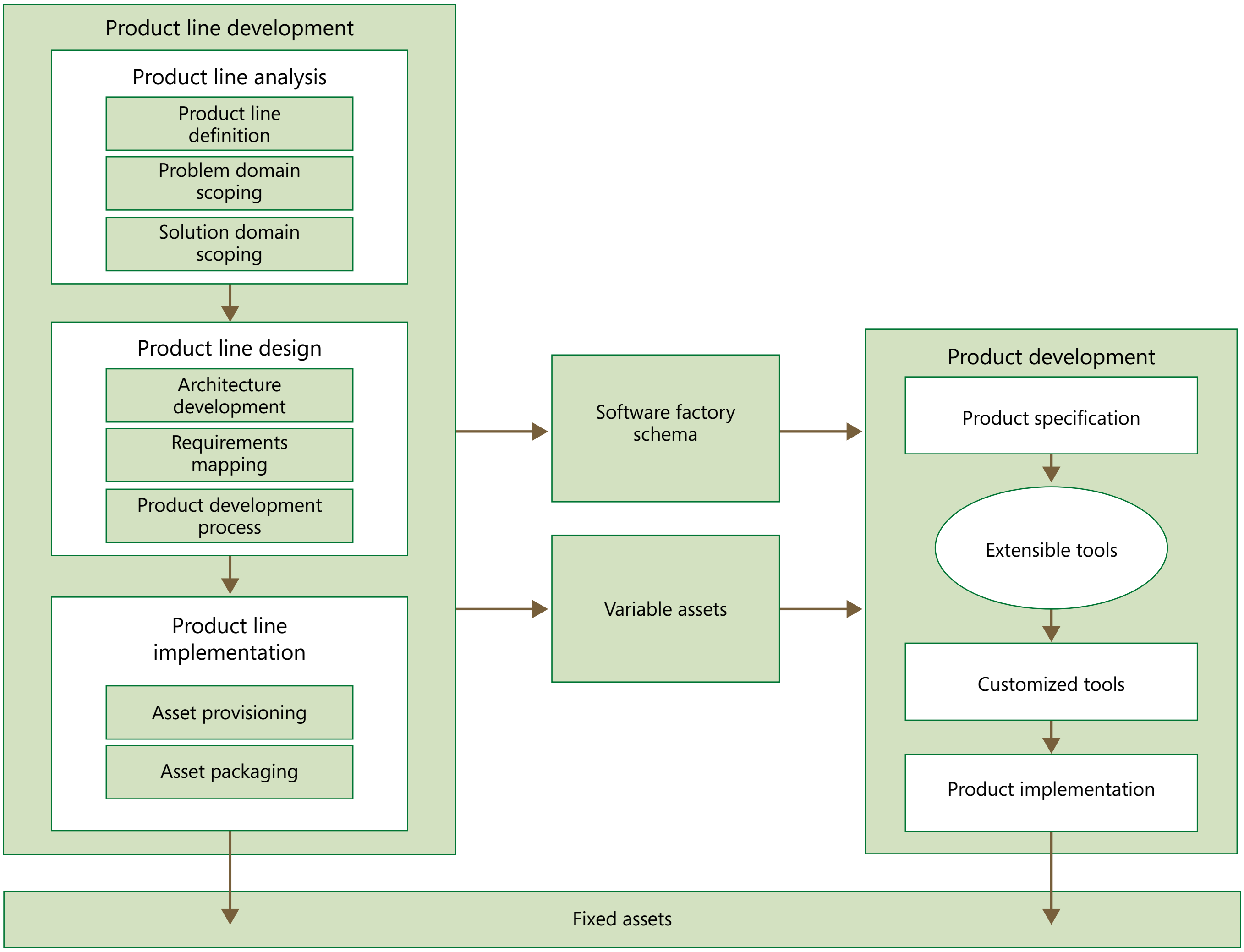
Factories, Factories, Factories

Factories can be of two types, horizontal and vertical (see Figure 4). *Horizontal factories* are those that help internally. They become the helper factories, those that don't target a specific industry but are broader and in general can be consumed by many other factories—for example, the Enterprise Framework Factory (EFx) by Jezz Santos at Microsoft. A *vertical factory*, on the other hand, is targeted toward an industry or domain—for example, finance, government, or health care. An example of a vertical factory would be the HL7 Factory talked about by Mauro Regio and Jack Greenfield from Microsoft in *The Architecture Journal* 6 (see Resources).

> "IT IS NOT NECESSARY TO BOIL THE OCEAN WHEN BUILDING YOUR OWN FACTORY. IT BECOMES MORE A PROCESS OF BUILDING OR CONSUMING HIGHLY-FOCUSED FACTORIES AND BUILDING THEM UP INTO A LARGER FACTORY TO MEET THE BUSINESS NEEDS OF YOUR CLIENTS"

It is also possible to capture industry knowledge in the viewpoints as well. We could state that around 70–80 percent of any client's base needs in a given industry are the same, and that the 20–30 percent difference leads to customization of the solutions to meet the individual customer's needs. Your organization probably already uses industry-level frameworks that capture the commonality among various businesses within a specific industry and handle the common portions of this information. For example, if you have been working for a specific

Figure 4 A Software factory



type of client such as the government for a number of years, you may already have the necessary knowledge to capture the essence of a government application.

If you can capture the commonality of industries using their business processes and the common enterprise components, variability for a given client can be captured by mining their business rules. To provide fit and alterability requires having the right parts to allow configurable systems. Examples of these parts include common processes, common services, data models, components, and objects. To fit the customer, a system must also be extensible and interoperable with existing systems. Thus, customizability in the fullest extent means being configurable, extensible, open, and supporting interoperability with legacy and other systems.

Once the commonality and variability are defined, a software factory becomes the mechanism to deploy and execute them with high levels of productivity. This mechanism enables a software consumer to rapidly configure, adapt, and assemble independently developed, self-describing, location-independent components to produce families of similar but distinct systems. Software factories would allow the workers in the field—architects, developers, or testers—to consume assets in a more prescribed way.

Where to Begin?

You are ready to start. Here is a check list of what you will need: Common processes, common services, data models, components, objects, patterns, models, and test plans, scripts, and data.

Identify where your current pain points are as a business. Do you require a lot of SharePoint skills, or is VB6-to-.NET migration becoming a challenge? Once you have identified this gap you can take a thin slice of this pain point and start to build up a factory. Start collecting the best practices for this area, and document what a group is doing while building a product. For the V1 of this product you will just be collecting factory assets. Once collected, tease out the areas of variability, and put in place your recipe for building this type of product.

After you have completed this iteration of your factory, test it out using the same team that built the V1 product. Is this iteration better? What are the problems? What are the metrics? By building the factory in iterations your success rate is higher, and you do not have to take a team away from delivering real work to find out if a factory-style approach is right for you. Factories themselves are extendable, and any enhancements to the factory would be handled by industry specialists. This approach allows for versioned factories. With every product built using the factory, the factory gets refined and improved.

Now that you have built a narrow, highly focused factory, now you want to build a larger factory to encompass more of a specific industry. Building a factory is no small task, but it need not be too onerous. The Microsoft Patterns and Practices Group is providing software factories of their own, such as the Service Factory, the Mobile Client Factory, and the Smart Client Factory. There are also factories such as the Project Glidepath Software Factory by Michael Lehman at Microsoft—which is designed to help ISVs build, deliver, and market applications for Windows Vista and the .NET Framework 3.0—and the EFx Architectural Framework Factory, which is a Microsoft Consulting Services offering to provide a comprehensive architectural model for enterprise-strength, service-oriented applications and services. These factories can either be used "as is," or they can be consumed by your own industry-specific factory and tailored to your individual needs. You can add your own factories to this recipe until you have covered the scope of a particular industry.

As you can see, it is not necessary to boil the ocean when building your own factory. It becomes more a process of building or consuming highly-focused factories and building them up into a larger factory to meet the business needs of your clients.

Success for a software factory can be gauged by strategic software reuse through a well-managed, product-line approach that achieves business goals listed previously. However, it is worth highlighting just how a software factory would support these goals:

- *We want simple, standard solutions.* A software factory supports this goal by creating the factory once and stamping out many products from the template.
- *Applications are designed and implemented faster; therefore, they can go to market faster.* Up to 80 percent of the work can be automated by a software factory.
- *Deliver the right industry and technology expertise, at the right price, at the right place, and at the right time.* Domain knowledge and expertise is baked into the software factory; the experts are at hand all the way through the development life cycle.
- *High-quality applications/solutions are done right the first time.* The software factory is tried, tested, and proven.
- *Deliver what the customer ordered.* A software factory meets the customers' business needs sooner, with less risk, and by tailoring a product as opposed to developing a custom project.

Software factories allow projects to jump-start from a well-defined baseline. Teams know what to do and how to do it. They have trusted, proven assets to help them do it quickly and correctly. This foundation makes future maintenance easier, improves quality and consistency, and also makes change impact easier to predict. New hires need less hand holding, budgets and schedules are more accurate, and a factory captures knowledge for future use.

Software factories also attempt to address the fact that there are just not enough architects to go around. Everyone gains by baking the knowledge of the experts into a factory. The company gains by having the experts working on the most important task, rather than hand holding development teams. The development teams no longer have to do so many rote and menial tasks; therefore, they are freed up to work on more interesting and challenging tasks. Of course, do not forget that clients gain by getting software built to their specifications, more rapidly, and within budget. •

Resources

Carnegie Mellon Software Engineering Institute
Software Product Lines
www.sei.cmu.edu/productlines

"Design and Implement a Software Factory," *The Architecture Journal,* Journal 6 (Microsoft Corporation 2006)

Generative Programming: Methods, Tools, and Applications, Krzysztof Czarnecki and Ulrich W. Eisenecker (Addison-Wesley Professional 2000)

Jezz Santos – Enterprise Framework
The EFx Architectural Guidance Software Factory
http://blogs.msdn.com/jezzsa/articles/677177.aspx

MSDN
Microsoft Visual Studio Developer Center
Domain-Specific Language (DSL) Tools
http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools

Visual Studio Team System Developer Center
Guidance Automation Extensions and Guidance Automation Toolkit
http://lab.msdn.microsoft.com/teamsystem/workshop/gat

Microsoft Corporation
gotdotnet codegallery
Patterns and Practices: Mobile Client Software Factory
http://practices.gotdotnet.com/projects/mobile

Patterns and Practices: Smart Client Software Factory
http://practices.gotdotnet.com/scbat

Web Service Software Factory Community
http://practices.gotdotnet.com/svcfactory

Project Glidepath
www.projectglidepath.net

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Jack Greenfield et al. (Wiley 2004)

Software Product Lines: Practices and Patterns, Paul Clements and Linda Northrop (Addison-Wesley Professional 2002)

About the Author

Steve Eadie is a software architect at EDS and is a graduate of the EDS Top Gun Program at Microsoft—an intensive three-month development program that seeks to develop the next wave of architects skilled in delivering Microsoft technologies. Steve resides in the United Kingdom, where he has responsibility for software factories and domain-specific languages. His main interests include model-driven development, software methodologies, software architecture, patterns, and mobile computing.

The Perspective-Based Architecture Method

by Lewis Curtis and George Cerbone

Summary

The Perspective-Based Architecture (PBA) method is the result of over two years of field-based development. PBA has included contributions, case studies, and revisions from customers, international bodies, and field teams. The ultimate goal of the PBA method is to promote high-quality decision making, which is accomplished through three stages containing structured, focused questions. Step one is a series of questions capturing the perspective (what is your environment and where is it going?). Step two examines the impact of alternatives or existing proposals on the captured perspective (from step one). Step three examines the impact of the final proposal on the captured perspective (from step one). This simple structure works with all processes and methodologies, technologies, and documentation standards.

Information technology has promoted a revolution in the modern world, enabling organizations to operate at faster, more productive levels. Automated collaboration software enables employees to work, communicate, and innovate anywhere, at any time, with anyone. Automated analysis engines can store and apply complex algorithms to large amounts of data from a variety of sources. Diverse organizations can communicate and integrate at speeds thought impossible ten years ago. Personal computers and mobile devices can enable ad hoc communities requiring almost zero administration.

With these innovations come increased competitive expectations. Markets that are used to innovating in the span of years are being challenged by newcomers that realize the power of innovating in weeks. This increased pressure has created the need for "extreme time to market" as businesses are forced by the marketplace to react more quickly to changes. Businesses are expecting employees to multitask to levels never before experienced in the workforce. Finally, as the Internet removes the friction associated with integration, companies are expected to integrate across geographic, lingual, political, and economic boundaries.

As organizational desires have grown and the technology sophistication has improved, IT architects are expected to make quick and cost-effective decisions for the organization. As the velocity of new solutions for the business increases, the complexity of aligning solutions with existing, current, and projected IT environments has created a quagmire drowning all who dare enter it. At the same time, business leaders are discovering IT architectural decisions are often some of the most important decisions impacting their organization, not just at the time that the decision is made, but also impacting subsequent decisions and plans.

In response to this enormous complexity, IT architects have developed terminology, techniques, methodologies, and processes to help them organize and manage complex architectural designs. Over the last 50 years these elements have significantly helped IT professionals decompose complex behavior and determine when and how to address specific issues and how to make and communicate decisions. Furthermore, a variety of quality checks and team modeling structures have been developed to encourage more precise and efficient activity.

For example, one mechanism that architects have developed for building reusable, decomposable structures is the design pattern. Design patterns are leveraged to build a predictable, reusable, and reliable solution. Many patterns are intended to give answers to common repeatable scenarios that we call answer patterns. The answer patterns have generally promoted a more consistent, predictable, and shorter time-to-market capability for many organizations.

Today's Challenges

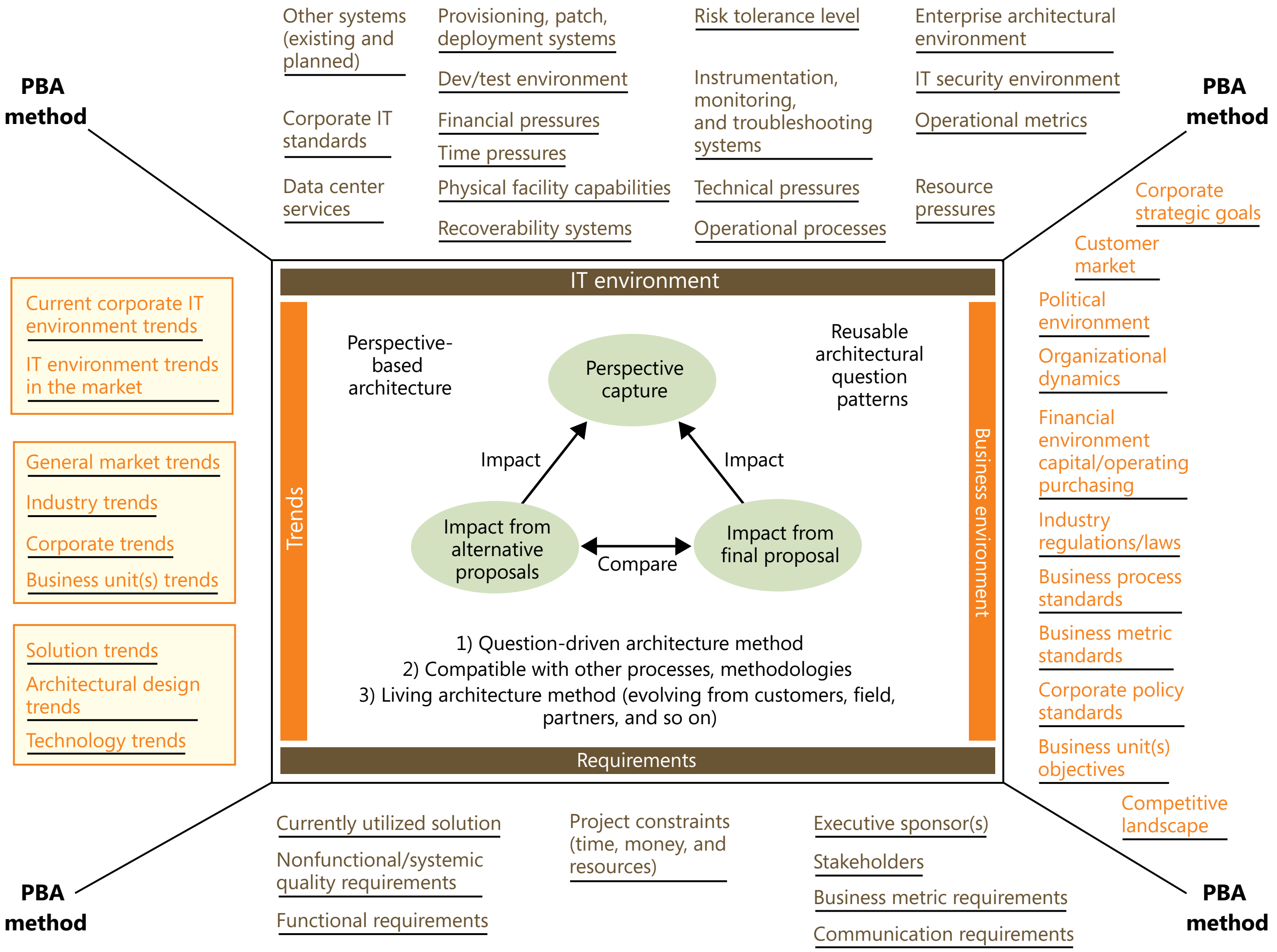
However, while methodologies, processes, and answer patterns have significantly helped architectural efforts, many projects still fall far short of expectations even when using the best architectural compositions. Current models promote repeatable activity but not necessarily better decisions. They focus on the process of decision making, not the inputs or outputs. Existing methodologies and processes have addressed several questions:

- When to make decisions?
- Who makes a decision?
- How to document decisions?
- How to decompose parts of a decision?

Answer patterns have focused on: what are common responses for common scenarios?

These are important questions, and existing methodologies have delivered good repeatable processes, communication methods, and answer patterns. However, we have not focused on the important questions architects should answer when engaging in any project. We have addressed the *how*, but not the *what*. Simply, if you don't ask the right questions, you will not look for the right answers. This situation is a critical weakness in IT architecture today. By focusing only on answer pat-

Figure 1 An overview of the PBA method


PBA method
PBA method
PBA method
PBA method
Other systems (existing and planned)
Corporate IT standards
Data center services
Provisioning, patch, deployment systems
Dev/test environment
Financial pressures
Time pressures
Physical facility capabilities
Recoverability systems
Risk tolerance level
Instrumentation, monitoring, and troubleshooting systems
Technical pressures
Operational processes
Enterprise architectural environment
IT security environment
Operational metrics
Resource pressures
IT environment
Current corporate IT environment trends
IT environment trends in the market
General market trends
Industry trends
Corporate trends
Business unit(s) trends
Solution trends
Architectural design trends
Technology trends
Trends
Perspective-based architecture
Perspective capture
Reusable architectural question patterns
Impact
Impact
Impact from alternative proposals
Compare
Impact from final proposal
1) Question-driven architecture method
2) Compatible with other processes, methodologies
3) Living architecture method (evolving from customers, field, partners, and so on)
Business environment
Requirements
Corporate strategic goals
Customer market
Political environment
Organizational dynamics
Financial environment capital/operating purchasing
Industry regulations/laws
Business process standards
Business metric standards
Corporate policy standards
Business unit(s) objectives
Competitive landscape
Currently utilized solution
Nonfunctional/systemic quality requirements
Functional requirements
Project constraints (time, money, and resources)
Executive sponsor(s)
Stakeholders
Business metric requirements
Communication requirements

terns and repeatable processes and methodologies, we have continually forced important projects to use designated answer patterns and methodologies that fit a particular frame of thinking. When these automated activities fail to fit with the current reality, architects operate in darkness, stumbling and experimenting to find the way to the light. The solution to this problem lies in changing the frame that defines our architectural thoughts.

The solution encompasses a new breed of architectural tools and structures that focus on addressing these critical questions:

- What are good questions to ask when making good, discriminating architectural decisions?
- How do I promote cohesive, well-thought-out decisions being considered on my project?
- How do I avoid common pitfalls when making architecture decisions?

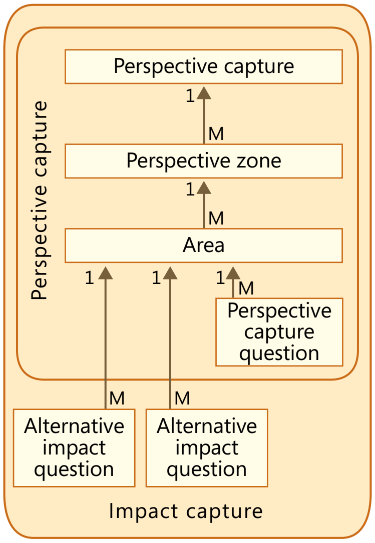
Successful senior IT architects often possess three fundamental capabilities when making decisions: knowledge, experience, and perspective.

Knowledge is the first area of development for any new IT architect. IT architects must possess a good fund of general knowledge spanning many disciplines. They must study and analyze vigorously to keep their knowledgebase as current and timely as possible.

Experience represents the second stage of an architect's career. Knowledge by itself does not an architect make. Architects attain experience observing best practices and pitfalls. Experience can be obtained from firsthand activity or reusable, knowledge-based communication from the experiences of others. Reusable experiences have been the catalyst to promote repeatable solutions, design patterns, and consistent architectural operations, which is the core source of answer patterns. Answer patterns are an attempt to codify and transfer experience, without the "pain" of firsthand discovery. Organizations have promoted best practices as a technique to capture answer patterns and processes of successful architects. While this technique has been very positive for the community, the answer pattern structure and process can also promote confusion when professionals fail to find patterns working for their specific engagement.

However, experience alone often isn't enough to promote quality decision making and analysis. Inspired by the book *Blink: The Power of Thinking Without Thinking* by Malcolm Gladwell and one of the books Gladwell's book was based on, *Simple Heuristics That Make Us Smart*, we found highly successful architects were able to make excellent analysis and architectural decisions in a short amount of time. When examining these architects, we found they often had equivalent levels of experience and knowledge as teams of architects working on the problem for a much longer time.

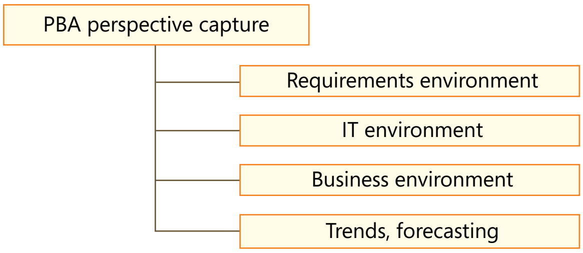
Figure 2 The PBA method's question structure meta model


Perspective capture
Perspective capture
1
M
Perspective zone
1
M
Area
1
1
1
M
Perspective capture question
M
M
Alternative impact question
Alternative impact question
Impact capture

Question rules for the PBA method:

1) The question must be applicable across technologies, methodologies, time, vendors, processes, and industries.
2) Each area question must be focused for one area only.
3) Each question must be addressed to current professional standards for an IT architect.

Figure 3 The four core zones



So what was the difference? How are some architects able to produce higher levels of analysis and architectural proposals with the same level of experience and knowledge? The answer was their perspective. They often started with quality questions and viewed architectural decisions from an ecosystem impact view rather than a technical capability view. Because they often emphasized examining the impact of decisions on an organization, they had refined their intuitive capability to see impact patterns within a short amount of time. It was this finding that inspired the study and development of a method that promoted this capability in our architects.

Simply, perspective represents the frame of understanding an architect brings to an engagement. Perspective represents an architect's ability to understand the impact of his or her discriminating decisions on the environment. This skill is the most challenging that an architect must attain. Most peer-respected IT architects demonstrate significant perspective-based skills. They operate with a more comprehensive frame of understanding. Developing and organizing that frame of understanding is the purpose of the PBA method (see Figure 1).

Introducing the PBA Method

Technology leaders making discriminating, mission-critical decisions for their organizations understand the importance of a mature perspective. Those who can see the larger picture and ask quality questions often develop quality solutions for their organizations.

When Lewis's grandfather led teams setting up satellite information communication systems in Turkey and other precarious areas during the Korean and Cold Wars, he faced monitoring and management issues, significant security issues, scalability issues, operational process issues, time pressures, and critical organization activities that were dependent on the reliability of the systems. Lives depended on these systems' operation. The issues faced in this system are nothing new, but the lessons learned are often forgotten, which is not unusual. In interviewing professionals outside the field of information technology, we found perspective is the most challenging and most desired attribute for a seasoned professional.

Therefore, the Perspective-Based Architecture (PBA) method is designed to be very simple. It does not care when you make a decision,

Figure 4 The requirements environment

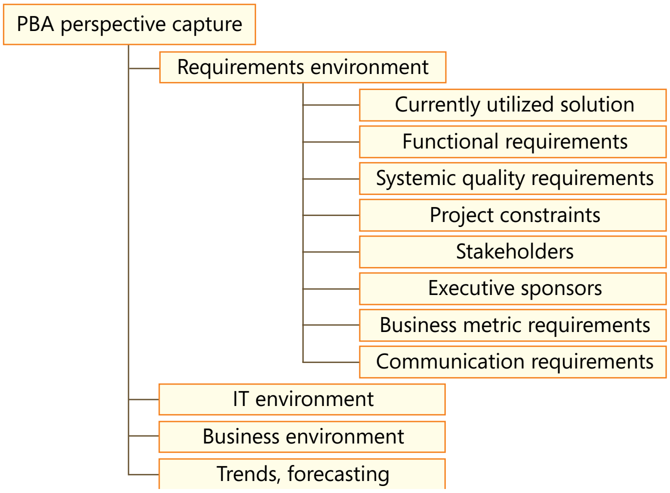
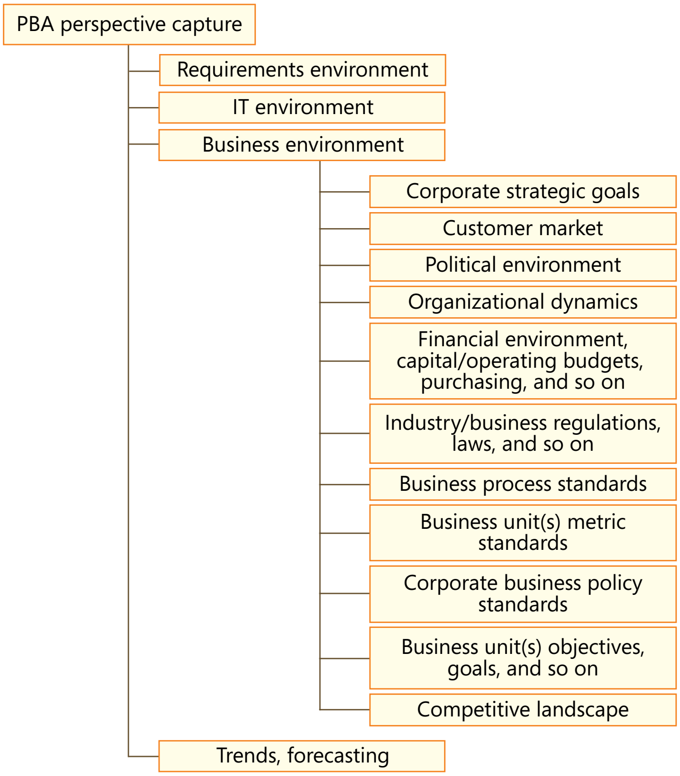


Figure 5 The business environment


PBA perspective capture
Requirements environment
IT environment
Business environment
Corporate strategic goals
Customer market
Political environment
Organizational dynamics
Financial environment, capital/operating budgets, purchasing, and so on
Industry/business regulations, laws, and so on
Business process standards
Business unit(s) metric standards
Corporate business policy standards
Business unit(s) objectives, goals, and so on
Competitive landscape
Trends, forecasting

Focusing Question Areas

Organizations are expanding the scope of IT architectural responsibility to more individuals. Here are some sample questions that are formulated correctly and organized for applying the PBA method specifically to address specific endeavors proactively.

Requirements Zone: Functional Requirements
Attaining good functional requirements is often fundamental toward arriving at any good solution. Working with customers and field teams for the last two years, we found there are several fundamental questions that must be answered no matter what methodology, process, or technology is used and no matter for which vendor (see Figure 9). We found also that by interviewing retired IT professionals working on past projects, management assumed they could always answer these fundamental questions. Of course, the quality and depth of answers coming from IT architectural professionals varies widely.

1. Who will interact with this solution (people, systems, and so on)?
2. How will these people and/or systems interact with this solution?
3. What do these people and/or systems expect functionally and to be observable from the solution?
4. Why do they want or need this specific functionally observable behavior delivered?
5. How will these functional expectations evolve over time?
6. What is the expected functional lifespan of this solution?

Business Environment Zone: Political Environment
Political power is the influence and control by individuals or groups to direct money, resources, time, strategy, and tactics of an organization toward its desired goals. Understanding and capitalizing on a political environment through executive sponsors and stakeholders using direct and indirect ethical influencing capabilities represents important skills of a seasoned corporate IT architect.

The political ecosystem represents a significant factor in the success of any ongoing IT initiative. It is critical that architects carefully understand the political dynamics and make decisions cohesive with that particular political landscape. Architecture decisions influence and are influenced by political forces within any organization. Here are some common questions that should be addressed by the architecture team, organized by section:

Perspective capture
1. Who are the significant political influencers in the organization?
2. Which political influencers could impact this solution?
3. What are the motivators of these political influencers?
4. Are there any political landscape issues that should be known?

Current or alternative impact
1. How do (or could) technology decisions of the current or alternative architecture design affect the political landscape?
2. How does (or could) the current or alternative architectural development plan impact the political landscape?
3. How do (or could) the current or alternative architectural deployment and transition plans affect the political landscape?
4. How does (or could) the process and role ownership model for the current or alternative architectural decisions affect the political landscape?
5. How will (or could) the executive sponsor(s) be affected by this potentially altered political landscape from the current or alternative architecture decisions?

Proposed impact
1. How does (or could) the proposed architectural development plan impact the political landscape?
2. How do (or could) the proposed architectural deployment and transition plans affect the political landscape?
3. How does (or could) the process and role ownership model for the proposed architectural decisions affect the political landscape?
4. How will (or could) the executive sponsor(s) be affected by this potentially altered political landscape from the proposed architecture decisions?
5. How do (or could) technology decisions of the proposed architecture design affect the political landscape?

IT Environment Zone: IT Management Systems
Most enterprise organizations have systems, processes, techniques, staff, and so on to troubleshoot, conduct some sort of root-cause analysis, and repair a solution's system component when needed. The need to reduce mean time to repair (MTTR) is crucial for a company to handle normal and abnormal exceptions in their day-to-day activities. The solution must align itself well with the current and forecasted troubleshooting and repair environment that organizations use to keep their IT operations successful:

Perspective capture
1. What systems and processes exist for troubleshooting and root-cause analysis in the IT organization?
2. How are these systems and/or processes for troubleshooting and root-cause analysis used in the IT organization?
3. What solutions are managed by these systems and processes for troubleshooting and root-cause analysis?

Requirements Zone: Current Solution(s)
As IT reuse is vital for organizations, leveraging existing capabilities is important as long as they align with the business's tactical and/or strategic goals. It is always important to understand what is currently providing any of the functional and/or nonfunctional needs of the initiative:

Perspective capture
1. What systems currently provide any of the scoped functional needs for the recognized stakeholders?
2. What is the architectural design of these systems?
3. When were these solutions designed and deployed?
4. Who designed and deployed these solutions?
5. Why were these solutions originally designed and deployed?
6. What is the general impact of these systems on the organization?

Trends, Forecasting Zone: Industry Vertical Directions/Trend(s)
While the general market directly impacts industry trends, these trends dramatically impact the organization and the success of most IT solutions. In other words, understanding industry trends is crucial toward increasing the business viability of the solution being designed:

Perspective capture
1. What are the relevant industry vertical directions/trends?
2. Why are these vertical industry directions/trends significant for the business?
3. How is the business addressing these vertical industry directions/trends?

what your title is, how you document decisions, or what taxonomy you use when examining an architecture. It can complement any structure needed. Simply, the purpose was not to reinvent existing professional structures but rather to complement current models, methods, and processes to drive more successful projects (see Figure 2).

To organize the questions, the PBA method divides the perspective into four broad zones: requirements; business environment; IT environment; and trends, forecasting (see Figure 3). Seasoned professionals will notice the importance of forecasting and time, two areas often overlooked in decision making. These zones by themselves are too broad for question-structure organization. Therefore, each broad zone contains many more focused areas of concern. This decomposition enables the architect to differentiate focused areas more easily. Each focused area of concern contains a number of questions for perspective capture, examining the impact from alternative proposals and examining the impact of the final proposal.

While this method might appear complicated for some, it's actually quite simple. Common questions were grouped together by subject to allow greater clarity and organizational understanding. (See the sidebar, "Focusing Question Areas" for question area examples organized by zone.)

The requirements zone represents common core questions and areas of concern for the specific project being considered (see Figure 4). Historically, this zone is where most architects have focused their time and energy on these areas of concern:

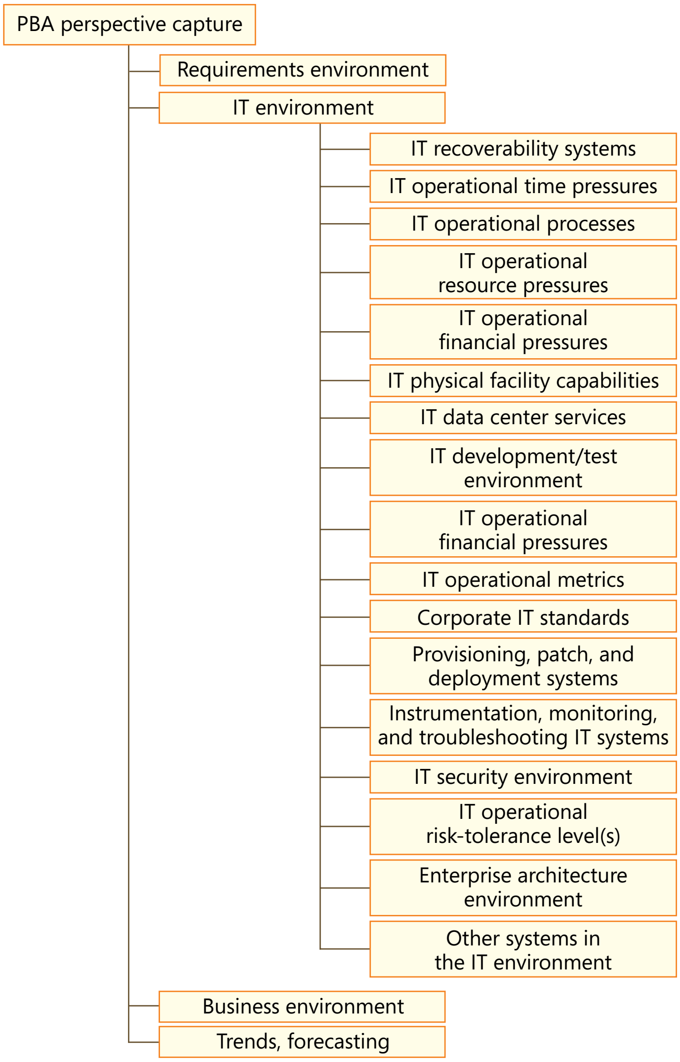
- Current utilized solution(s)
- Functional requirements
- Systemic quality (nonfunctional) requirements
- Project constraints (time, resources, and money)
- Stakeholders for the solution
- Executive sponsor(s)
- Business metric requirements related to the solution
- Communication requirements for the solution and project team

The business environment zone represents critical questions for the business. These sections focus on relevant parts for the operationally successful organization in the market (see Figure 5). This section does not consider the IT environment; rather, it is entirely business focused. The business environment must be addressed in the language, taxonomy, and structure the business really uses for this analysis to be successful. Focused areas of concern are:

- Corporate strategic goals
- Customer market
- Political environment
- Organizational dynamics
- Financial environment
- Industry business regulations/laws
- Business process standards
- Business unit(s) metric standards
- Corporate business policy standards
- Business unit(s) objectives
- Competitive landscape

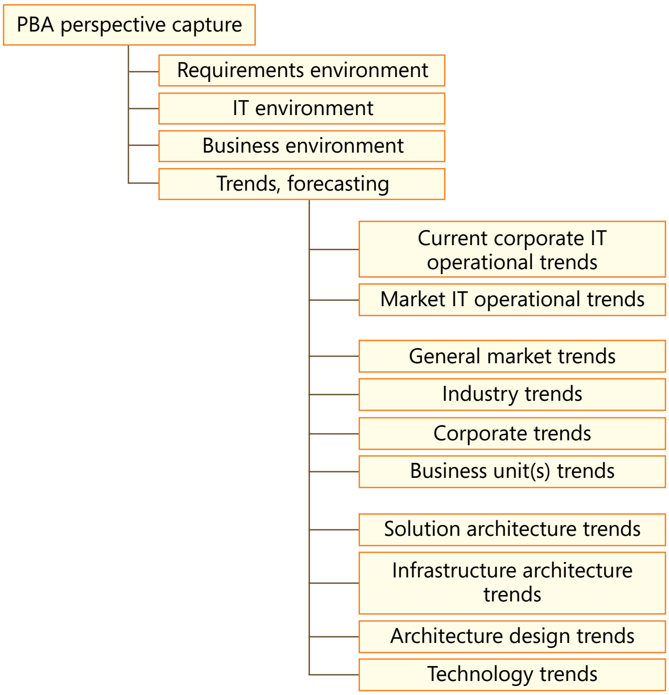
The IT environment zone represents the operational IT structures, processes, issues, and systems for all important technology activity for the organization (see Figure 6). Similar to the business environment, IT questions must be addressed using the language of IT. Focused areas of concern are:

Figure 6 The IT environment


PBA perspective capture
Requirements environment
IT environment
IT recoverability systems
IT operational time pressures
IT operational processes
IT operational resource pressures
IT operational financial pressures
IT physical facility capabilities
IT data center services
IT development/test environment
IT operational financial pressures
IT operational metrics
Corporate IT standards
Provisioning, patch, and deployment systems
Instrumentation, monitoring, and troubleshooting IT systems
IT security environment
IT operational risk-tolerance level(s)
Enterprise architecture environment
Other systems in the IT environment
Business environment
Trends, forecasting

- IT recoverability systems
- IT operational time pressures
- IT operational processes
- IT operational resource pressures
- IT operational financial pressures
- IT physical facility capabilities
- IT data center services
- IT development/test environment
- IT operational technical pressures
- IT operational metrics
- Corporate IT standards
- IT provisioning, patch, and deployment systems
- Instrumentation, monitoring, and troubleshooting IT systems
- IT security environment
- IT operational risk tolerance level(s)
- Enterprise architecture environment
- Other systems in the IT environment (existing and planned)

Figure 7 The trends, forecasting environment



The trends, forecasting zone considers the time dimension and represents future directions for the IT environment, business environment, forecasted technologies/solutions, and architectural patterns relevant to the current solution being considered by the architecture team (see Figure 7). Architects must be able to honestly and effectively forecast and analyze trends properly. Focused areas of concern are:

- Current corporate IT operational trends
- Market IT operational trends
- General market business trends
- Applicable industry trends
- Specific corporate trends
- Specific business unit(s) trends
- Solution architecture trends
- Infrastructure architecture trends
- Architectural design trends
- Technology trends

The PBA Method Process

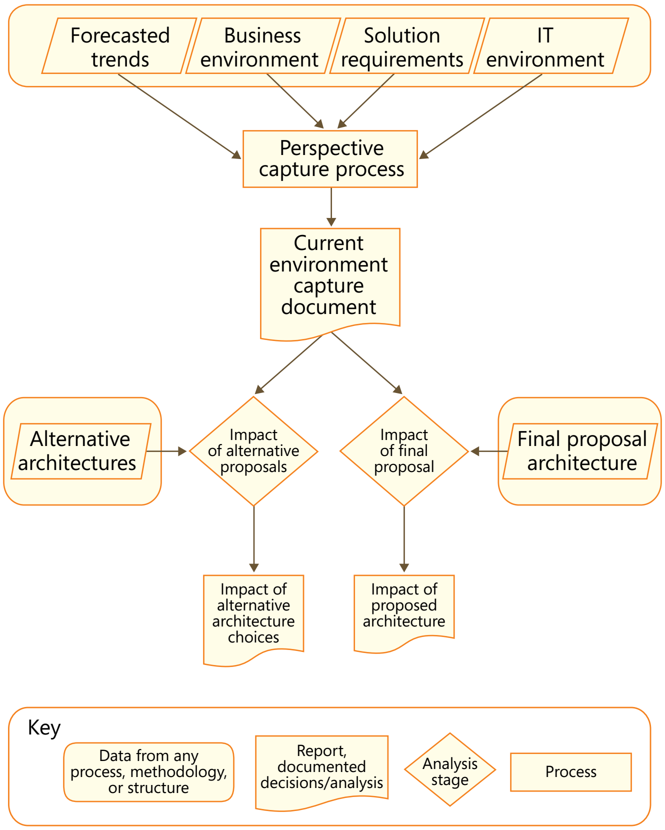
The PBA method is designed to be simple and align with most processes being used today. The process involves capturing the perspective and then using that capture to understand the impact of various architectural decisions (see Figure 8). Using the PBA method proactively with your favorite design methodology or process helps promote better decision making during the architectural design experience rather than in hindsight. Let us take a look at the process.

Step one – capture the perspective. Each subarea has detailed questions asking the architectural team to describe what they understand specifically. An interesting feature of the PBA method is that "I don't know" is an acceptable answer. For areas where they do not have answers, they write, "I (or we) do not know the answer." This feature enables the entire team to transparently understand the perspective more comprehensively, while being honest about areas of uncertainty, and promotes cross-team collaboration and open communications for critical engagements. With a unified collaborative perspective capture, the team can begin examining the impact of multiple proposals more effectively.

Step two – examine the impact of alternative proposals. This step involves a series of structured impact questions for each subarea in the perspective capture and focuses on alternative or existing proposals (depending on the situation) and their impact on the captured perspective. Again, architects must document both what impact they understand and what impact areas they are uncertain of.

Step three – Examine the impact of the final proposal. This step involves a similar (to step two) series of structured impact questions, except this step focuses on the final proposal. By using the same perspective capture (as in step one), the impact analysis promotes a more balanced and open approach and drives decisions encouraging best-fit models for the specific organization or business. While answering the question areas, architects can use any tools, methodologies, or architectural processes needed for the project. Often, an architect will use the information for comparison analysis and communicating decision justifications. Something we found more interesting was that the structure promoted higher-quality decisions aligning with the business, organizational structures, and trends rather than merely promoting a favorite technology, product, or technique. In other words, because architects had to address the perspective from a more comprehensive viewpoint, their decisions aligned much more effectively.

Figure 8 Process structure for the PBA method


Forecasted trends
Business environment
Solution requirements
IT environment
Perspective capture process
Current environment capture document
Alternative architectures
Impact of alternative proposals
Impact of final proposal
Final proposal architecture
Impact of alternative architecture choices
Impact of proposed architecture
Key
Data from any process, methodology, or structure
Report, documented decisions/analysis
Analysis stage
Process

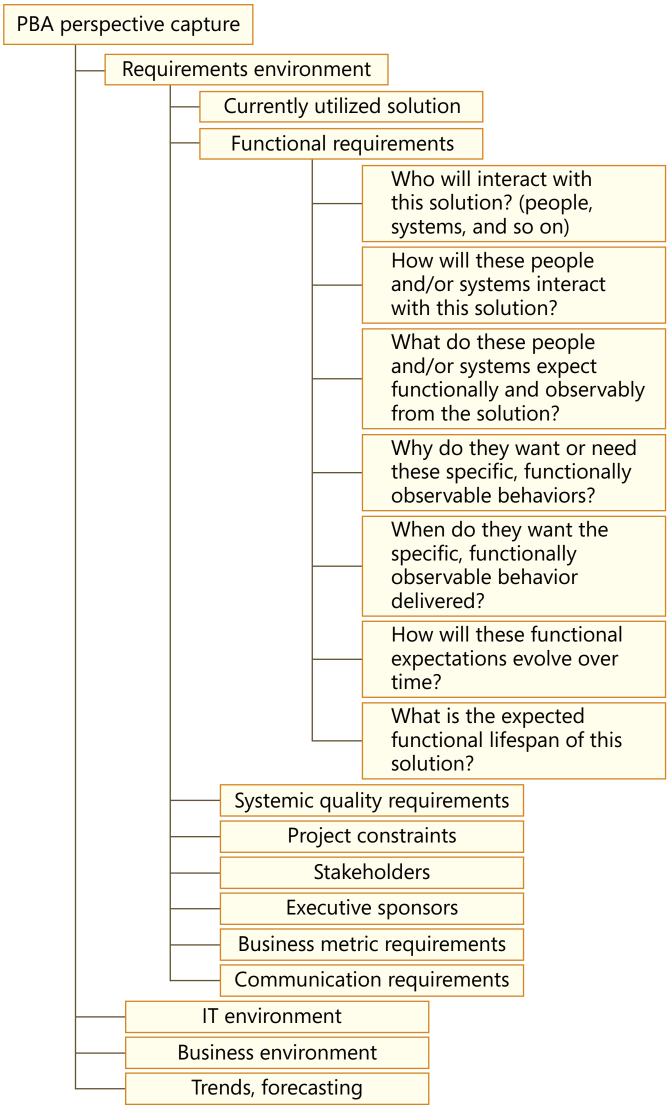
A common question for the process is: "Do step two and step three need to be sequential?" Not at all. Architects can reverse the order of step two and three or even conduct them in parallel, depending on specific needs. The most important point is doing step one first!

We found as organizations moved into their second project using PBA, many perspective-based questions where already answered, or they changed very little. The new project team operates with more impact awareness moving toward decisions and analysis at a faster pace. Architectural teams become incrementally more effective through this continuous learning model.

Does It Work?

Working with Federated Systems Group under the direction of Brian Derda, we conducted an unbiased case study program. It was controlled by Federated Systems Group with architects from three different companies working under very short deadlines. Using a Groove collaboration workspace populated with material from the PBA method, the team rapidly assembled high-quality strategic and tactical analysis with better recommendations and decisions as evaluated by both Federated Systems Group and Microsoft.

Figure 9 Sample functional requirements area questions


PBA perspective capture
Requirements environment
Currently utilized solution
Functional requirements
Who will interact with this solution? (people, systems, and so on)
How will these people and/or systems interact with this solution?
What do these people and/or systems expect functionally and observably from the solution?
Why do they want or need these specific, functionally observable behaviors?
When do they want the specific, functionally observable behavior delivered?
How will these functional expectations evolve over time?
What is the expected functional lifespan of this solution?
Systemic quality requirements
Project constraints
Stakeholders
Executive sponsors
Business metric requirements
Communication requirements
IT environment
Business environment
Trends, forecasting

Furthermore, we found the PBA method promoted dialogue between diverse organizations and companies, reduced confusion between team members, and promoted a higher level of trust with transparent analysis exposure. Finally, we discovered the PBA method was very easy to learn, and teams began engaging with this framework within a very short time frame (see Resources).

> "THE PBA METHOD IS DESIGNED TO BE SIMPLE AND ALIGN WITH MOST PROCESSES BEING USED TODAY"

Successful and respected architects already implicitly use many of these questions today in their architectural analysis to make good decisions. The PBA method focuses on enabling architects to operate more efficiently with a perspective-based framework. This efficiency is accomplished through a reusable question model with well-defined subject areas encapsulated in a simple and easy-to-learn structure. Of course, making difficult decisions often means answering challenging questions, and while the structure is simple, the questions can be quite challenging.

Furthermore, the PBA method represents a living architecture method. It grows, adapts, and evolves from field, international body, customer, and partner participation. This approach is very different from past architecture methodologies and processes.

Finally, it can work side by side with anything. The power of the PBA method is its simplicity and focus. We invite you to become involved in leveraging and evolving the PBA method (see Resources). •

Resources

Blink: The Power of Thinking Without Thinking, Malcolm Gladwell (Little, Brown and Company 2005)

Perspective-Based Architecture
The PBA Method
www.perspectivebasedarchitecture.com

Simple Heuristics That Make Us Smart, Gerd Gigerenzer, Peter M. Todd, and the ABC Research Group, General editor: Stephen Stich, Rutgers University (Oxford University Press 2000)

Skyscrapr
ARCast: Perspective-Based Architecture (from Microsoft and Federated)
www.skyscrapr.net/blogs/arcasts/default.aspx?ID=278

About the Authors

Lewis Curtis is an infrastructure architect evangelist for developer and platform evangelism, east region, Microsoft Corporation. He is a Microsoft Certified Architect in infrastructure and a member of the MCA board of directors. Contact Lewis at lewis.curtis@microsoft.com or at http://blogs.technet.com/lcurtis.

George Cerbone is an infrastructure architect evangelist for developer and platform evangelism, financial services, Microsoft Corporation, and a Microsoft Certified Architect in infrastructure. Contact George at george.cerbone@microsoft.com or at his blog at http://blogs.msdn.com/gerbone/.

THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

THE ARCHITECTURE JOURNAL 5
THE ARCHITECTURE JOURNAL 6
THE ARCHITECTURE JOURNAL 7
THE ARCHITECTURE JOURNAL 8
Search ARC Journal
PDF versions: English Español Português Français Deutsch Русский 汉语 日本語 한국어
cture Journal Home > Issue 8

ARC

Microsoft®

THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

098-106629

Subscribe at: www.architecturejournal.net