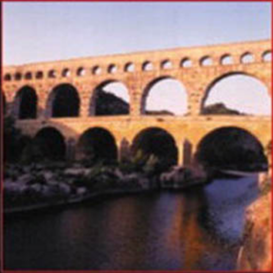
The Addison-Wesley Signature Series

Beyond Software Architecture

Creating and Sustaining Winning Solutions

Luke Hohmann

A Martin Fowler Signature Book



Forewords by Martin Fowler and Guy Kawasaki

Library of Congress Cataloging-in-Publication Data

Hohmann, Luke.
Beyond software architecture: creating and sustaining winning solutions / Luke Hohmann.
p. cm.
ISBN 0-201-77594-8 (pbk. : alk. paper)
1. Computer software. 2. Computer architecture. I. Title.
QA76.754.H65 2003
005.1—dc21 2002152562

ISBN 0-201-77594-8
Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.
9th Printing May 2008

Contents

This page intentionally left blank

Foreword by Martin Fowler

Architecture has become a very slippery word in the software business. It's hard to come up with any solid definition of what it means. I see it as a fundamentally subjective term—when people describe their software architecture they select the important parts of their systems, how these parts fit together, and the key decisions they made in designing their systems. Architecture is also seen as a technical issue, with the implication that the key decisions that need to be made are technical decisions.

In talking with Luke over the last few years I've really enjoyed the fact that he talks about the kinds of things that are often sadly omitted from most architectural discussions—yet are every bit as important. Such things as the marketing view of a system, licensing terms, branding, deployment, billing. All of these issues have important technical and business implications. Senior technical people need to think about this stuff, or otherwise a technically capable system could fail to be good business decision.

Many of these issues matter most to people who sell software to other entities. But even if you're an architect of an in-house IS shop these issues are just as likely to trip you up. Licensing agreements with your vendors can make a big difference to the costs of the software you deploy, billing may become important if your business decides it wants to introduce a charge-back scheme, branding helps affect your visibility to the business side of your company.

Luke writes from the perspective of someone who has dealt with both the technical and business sides of software development. It's a duality I find appealing because it's led him to ponder issues that often don't get talked about. He shows that it's often the things you don't think to worry about that hurt you the most and in the process provides the advice you need to deal with them. As a result this book is a much needed compliment to the technical aspects of software design.

Martin Fowler,
Series Editor

This page intentionally left blank

Foreword by Guy Kawasaki

First, you have to understand that I think writing software is more art than science. How is it, then, that I would write a foreword for a book that is focused on the nuts and bolts, instead of the creative?

The creative process of both art and software is highly overrated. Clear your mind of visions of sitting around in a comfy chair as the software muse penetrates your mind and thousands of lines of elegant code flow effortlessly from your brain. If you've created software, you know it isn't like this.

Art is hard. Debugging is even harder. This is a book that will help you and your team become better artists. It is about discipline, teamwork, perspiration, and, yes, inspiration. I hope you read it, and it enables you to create great art (a.k.a. software) that changes the world.

Guy Kawasaki,
CEO
Garage Technology Ventures

Preface

Many excellent books have been written on software architecture. These books, which, among other things, define, classify, and describe software architectures, define notations for representing and communicating architectural choices, and provide guidance on making good architectural decisions, have enduring value. Unfortunately, while these books may help you build a successful *architecture,* they fall short of the goal of helping you create a *winning solution*. To create a winning solution, you need to move beyond subsystems and interfaces, beyond architectural patterns such as Front Controller or Pipes and Filters, and beyond creating third-normal-form relational databases. You need to move beyond software architecture and move toward understanding and embracing the business issues that must be resolved in order to create a winning solution.

An example of one such business issue concerns technical support. It is inevitable that some of your customers are going to have a problem with your software. The choices you've made long ago in such areas as log file design, how the system is integrated with other systems, how the system is configured, or how the system is upgraded will determine how well you can solve their problems. *Beyond Software Architecture* helps you move beyond software architecture and toward creating winning solutions by discussing a wide range of business issues and their interrelationship with architectural choices.

This book presents a unique perspective that is motivated and informed by my experiences in creating single-user programs costing less than $50; software systems used in academic research; utilities to diagnose and fix problems associated with internally developed systems; and distributed, enterprise-class platforms costing millions of dollars. Along the way, I've played a variety of roles. I've been an individual contributor, a direct manager, and a senior member of the corporate executive staff. At various times I've either worked in or led engineering, product marketing and management, quality assurance, technical publications, and first- and second-line support organizations. I've managed teams and projects across multiple cities and continents.

The common thread tying all of this software together is that it was created to provide value to some person. Research software, for example, serves the needs of the researchers who are trying to understand some phenomena. Enterprise application software, dealing with everything from customers to supply-chain management, is designed to serve the needs of a well-defined set of users and the businesses that license it in a sustainably profitable manner. Similar comments apply to every other kind of software, from games to personal contact managers, inventory management systems to graphic design tools.

The issues identified and discussed in this book affect every kind of software. Their presentation and discussion occur most often in the context of enterprise application software, where I have spent most of my professional career. While they have no universally accepted definition, enterprise applications typically meet one or more of the following characteristics:

- They are designed to support the needs of a business, at either a departmental or larger organizational unit.
- They are relatively expensive to build or license ($50,000–$5,000,000 and up).
- They have complex deployment and operational requirements.
- They can be operated independently, but the needs of the business are often best served when they are integrated with other enterprise applications.

Even if you're not creating an enterprise application, you will find this book useful. Creating sustainable software solutions—meeting customer needs over a long period of time through multiple releases—is a challenging, enjoyable, and rewarding endeavor, certainly not limited to the domain of enterprise applications!

Although I will often refer to software architecture and discuss technical matters, my discussions won't focus on such things as the best ways to diagram or document your architecture or the deeper design principles associated with creating robust, distributed Web-based component systems. As I said earlier, there are plenty of books that address these topics—in fact, almost *too* many, with the unfortunate side-effect that many people become so focused on technical details that they lose sight of the business value they're trying to provide.

Instead of concentrating on purely technical choices, *Beyond Software Architecture* helps you create and sustain truly winning solutions by focusing on the practical, nuts-and-bolts choices that must be made by the development team in a wide variety of areas. I have found that focusing on practical matters, such as how you should identify a release or integrate branding elements into your solution, reduces the often artificial barriers that can exist between developers and the business and marketing people with whom they work.

These barriers prevent both groups from creating winning solutions. I cringe when engineers take only a techno*logy* view without due consideration of *business* issues, or when marketing people make “get-me-this-feature” demands without due consideration of their underlying technical ramifications. When either side takes a position without due consideration of its impact, the likelihood of creating and sustaining a winning solution drops dramatically.

What is especially troubling is that these arguments seem to be made in support of the idea that technical issues can somehow be separated from business issues, or that business issues can somehow be separated from technical issues. At best this is simply wrong; at worst it can be a recipe for disaster. Developers are routinely asked to endure the hardships of design extremes, such as a low-memory footprint, in order to reduce total system cost. Entire companies are started to compete in existing markets because investors are convinced that one or more technological breakthroughs will provide the competitive advantage necessary for success. Not surprisingly, investors are even more eager to invest when the technological breakthrough is accompanied by a similar breakthrough in the business model being offered to customers.

Managing the interrelationship between technology and business will be a recurring theme throughout this book. Handle only the former and you might have an interesting technology or, perhaps, an elegant system,—but one that ultimately withers because no one is using it. Handle only the latter and you’ll have a paper solution that excites lots of people and may even get you funding—but one that doesn’t deliver any sustainable value. Handle both and you’ll have a winning solution. While creating new technologies or elegant systems can be fun, and designing sophisticated new software applications or business models can be exciting, both pale in comparison to the deep satisfaction that comes from creating winning solutions and sustaining them.

Acknowledgments

Thanks to all of the people that have helped me create this book. I am especially indebted to Don Olsen, Haim Kilov, Rebecca Wirfs-Brock, Myron Ahn, Rob Purser, Ron Lunde, Scott Ambler and Dave Smith for their detailed reviews. Steve Sweeting, Craig Larman, Todd Girvin, Erik Petersen, Sandra Carrico, Adam Jackson, Tony Navarette, Chris Reavis, Elisabeth Hendrickson, James Bach and Alan Shalloway all provided detailed reviews of one or more chapters. Some of you were especially critical, and the book is better for it.

A very special thanks goes to Bob Glass for working with me to create the title. Once again a phone call was better than a lot of e-mail.

Ron, I imagined that writing a chapter with you would be fun. It was!

Steve Dodds, Lee Sigler, and a host of students and colleagues have provided me with inspiration and friendship as I undertook this project.

Special thanks to my good friend and original publisher Paul Becker, who patiently waited for me to complete it. Paul, it has been a few years since I completed my first book. Thanks for your willingness to wait until I had something to say.

I am deeply indebted to the superb professionalism and hard work of the Addison-Wesley production and marketing organization. They took a raw manuscript and helped shape it into a book. Thanks to Mary O'Brien, Elizabeth Ryan, Marilyn Rash, Chris Guzikowski, and Dianne Wood.

I have undoubtedly forgotten to mention one or more individuals who have helped in the creation of this book. This was, by no means, intentional. I'm also certain that there are several areas in which the book can be improved based on your experience. Tell me what I've missed, or how I can improve the book—and I will.

Luke Hohmann
luke@lukehohmann.com

Chapter 3

The Difference between Marketecture and Tarchitecture

Chapter 1 presented an overview of software architecture. Chapter 2 followed with a discussion of product management. This chapter returns to architecture and clarifies how the marketing and technical aspects of the system work together to achieve business objectives.

Who Is Responsible for What?

Software systems can be divided architecturally along two broad dimensions. The first is the *marketecture*, or the "marketing architecture." The second is the *tarchitecture,* or the "technical architecture." I refer to the traditional software architect or chief technologist as the *tarchitect* and the product marketing manager, business manager, or program manager responsible for the system as the *marketect*.

The *tarchitecture* is the dominant frame of reference when developers think of a system's architecture. For software systems it encompasses subsystems, interfaces, distribution of processing responsibilities among processing elements, threading models, and so forth. As discussed in Chapter 1, in recent years several authors have documented distinct styles or patterns of tarchitecture. These include client/server, pipeline, embedded systems, and blackboards, to name a few. Some descriptions offer examples of where these systems are most appropriately applied.

Marketecture is the business perspective of the system's architecture. It embodies the complete business model, including the licensing and selling models, value propositions, technical details relevant to the customer, data sheets, competitive differentiation, brand elements, the mental model marketing is attempting to create for the customer, and the system's specific business objectives. Marketecture includes—as a necessary component for shared collaboration between the marketects, tarchitects, and developers—descriptions of functionality that are commonly included in marketing requirements documents (MRDs), use cases, and so forth. Many times the term *whole product* is used to mean marketecture.

The $50,000 Boolean Flag

One "heavy client" client/server architecture I helped create had a marketing requirement for "modular" extension of system functionality. Its primary objective was that each module be separately priced and licensed to customers. The business model was that, for each desired option, customers purchase a module for the server that provided the necessary core functionality. Each client would then install a separately licensed plug-in to access this functionality. In this manner, "modules" resided at both the server and client level. One example was the "extended reporting module"—a set of reports, views, and related database extract code that a customer could license for an additional fee. In terms of our pricing schedule, modules were sold as separate line items.

Instead of creating a true module on the server, we simply built all of the code into the server and enabled/disabled various "modules" with simple Boolean flags. Product management was happy because the group could "install" and "uninstall" the module in a manner consistent with their goals and objectives for the overall business model. Engineering was happy because building one product with Boolean flags is considerably simpler than building two products and dealing with the issues that would inevitably arise regarding the installation, operation, maintenance, and upgrade of multiple components. Internally, this approach became known as the "$50,000 Boolean flag."

The inverse to this approach can also work quite nicely. In this same system, we sold a client-side COM API that was physically created as a separate DLL. This allowed us to create and distribute bug fixes, updates, and so forth, very easily; instead of upgrading a monolithic client (challenging in Microsoft-based architectures), we could simply distribute a new DLL. Marketing didn't sell the API as a separate component, but instead promoted it as an "integrated" part of the client.

Moral? Maintaining a difference between marketecture and tarchitecture gives both teams the flexibility to choose what they think is the best approach to solving a variety of technical and business problems.

Early Forces in Solution Development

A variety of technical and market forces shape a winning solution. These range from the core technologies to the competitive landscape to the maturity target market. What makes these forces so interesting is that they are always changing: Technology changes, the competitive landscape changes, markets mature, and new markets emerge.

Three particularly influential forces in the early stages of development are the *ilities*, the problem domain, and the technology base. As shown in Figure 3-1, driving, and being driven by, these forces are the target market, shown at the upper right, and the development organization, shown at the upper left. Product management is shown in the center to emphasize its collaborative, leadership role in resolving these forces.

The strength of the affinity that the target market and developers have with various forces is represented by single or double arrows. The final solution, including the marketing and technical architectures, lives in the "space" defined by all of the forces that shape its creation.

The *problem domain* is the central force in the development of a winning solution. Any given problem domain, such as *credit card transaction processing*, *automotive power systems*, or *inventory management*, immediately evokes a unique set of rules, nomenclature, procedures, workflows, and the like. Included in my definition of the *problem domain* is the ecosystem in which the solution exists, including customers, suppliers, competitors, and regulatory entities. Understanding the problem domain is a key prerequisite for both the marketect and the tarchitect if they wish build a winning solution. This is why most every development methodology places such a strong emphasis on gathering, validating, and understanding requirements as well as modeling the solution. This is also why effective product development places such an emphasis on the concept proposal and the business plan.

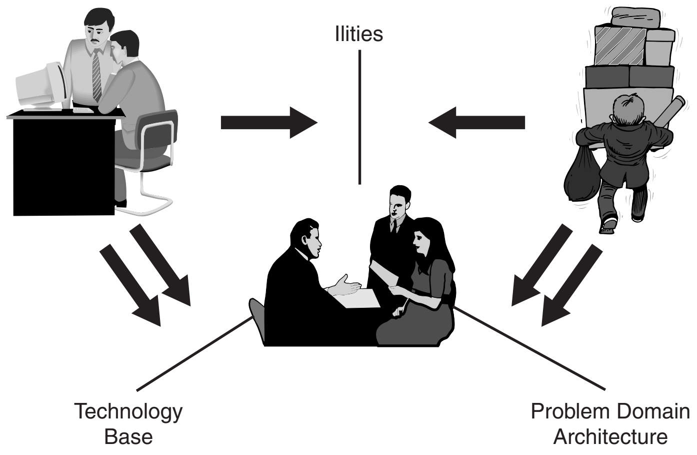


FIGURE 3-1 Forces shaping software architectures

The interplay between the marketect and the tarchitect in this process is quite interesting. Recall from Chapter 2 that the marketect's primary job is clarifying and prioritizing market needs; the tarchitect's primary job is to create a technical solution that will meet these needs. If the marketect is convinced that speed is paramount, as opposed to flexibility or usability, then the tarchitect will make certain choices that emphasize speed. Simply meeting the prioritized requirements, however, is insufficient to produce a successful tarchitecture. For this, the tarchitect must also bring his or her own domain experience to the tarchitectural design.

The requirement of extensive domain knowledge for a tarchitect is so strong that few developers can be promoted to this position until they have considerable experience and skill building systems within the specified domain. My rule of thumb is that, before someone can be considered a tarchitect, he or she must have done one of the following:

- Been a key member of a team that has created, from scratch, at least one major system in the given domain and has experienced the effects of that system through at least two full releases after the initial release (three releases total).
- Been a key member of a team that has made major architectural changes to an existing system and experienced the effects of these changes through at least two full release cycles after the changes were introduced.

You're not an architect in your very first job. You're not an architect after the first release. There is simply no substitute for sticking with a problem long enough to receive and process the feedback generated through customer use of your system. Do this long enough and you may gain sufficient experience to become an architect.

Ilities are the various quality and product attributes ascribed to the architecture. As Bass [98] points out, they fall within two broad dimensions: those discerned by observing the system at runtime and those *not* observed by observing the system at runtime. The former, including such attributes as performance and usability, are directly influenced by the target customer. The latter, such as testability and modifiability, are secondary attributes that govern the future relationship with the target customer. Because these secondary attributes are often informally specified, if they are specified at all, the discipline in tarchitectural design and associated system construction is critically important.

Care must be taken when the marketecture or marketect routinely accepts lesser *ility* attributes than those desired by the development team. When a developer wants to fix a bug or improve performance, but marketing thinks the product can be safely shipped without the fix or that the current performance is acceptable, tempers can flare, especially as you get closer to the projected release date. Keep things cool by creating forums that allow both development and marketing to express their points of view. For example, marketing needs to present arguments that a particular choice is "good enough" for the target customer.

I've found it especially helpful to have customers participate in forums. I vividly remember one customer demanding that we allow her to ship a beta version of our software three months before the scheduled delivery date. Our software was a core

Sometimes "The Hard Way" Is the Only Way

Most of the time the only way to learn a domain is through long-term relationships with customers. Among my responsibilities at a major digital content security provider was the creation of a backend server architecture that supported multitier distribution of software. As I learned the capabilities of the system, I also learned some of the lessons key members of the development team had learned over several years of working with major software publishers.

One of the most interesting lessons lay in the management of serial numbers. As it turns out, almost every major software vendor has a unique approach to managing serial numbers through its sales channel. Some create them in real time. Others create serial number blocks that are distributed according to predetermined algorithms to key channel participants. In this approach, the numbers are used not only for identification of shipped software but also for backend reporting and analysis. Other vendors use variants of these approaches.

Supporting every variant requires close interaction between marketects and tarchitects. In the case of this company, it was also vital to involve professional services, since they were the group that made everything "work" for a customer. It was clear to me that the only way the system could have evolved to support all of these demands was through the long-term relationships established with customers that enabled key team members to learn the problem domain.

component to her software. Any delays in shipping our software affected her customers. She readily acknowledged that the beta had many issues that needed to be resolved. However, its value was so compelling and her customer's need was so great that we eventually agreed to let her ship the beta subject to some strict terms and conditions regarding its use and a firm commitment to upgrade the released version when we thought it was ready.

Engineering (and especially quality assurance) needs to make certain that the risks associated with *good enough* choices are clearly understood. In the example I just mentioned, engineering provided the customer with a very clear assessment of how the product would fail outright under certain usage scenarios. This didn't change her mind—the beta still shipped—but it did enable her to equip her customer support organization with answers should these problems arisen in the field.

As described in Chapter 1, most development teams must make a variety of technical compromises in order to ship the product on time. Managing these compromises is difficult, as most compromises have their most pronounced negative effect in the release that follows the release in which they were introduced. This is another reason to demand that your tarchitect have the experience of two or more full release cycles. Only experience can help you gauge the potential severity of a technical compromise

Bug Severities, Priorities, and Normalization

One technique that I have found very effective in managing ilities is to classify bugs by severity and priority. Severity refers to the impact of the bug on the customer. Setting it to a value ranging from 1 to 5 works well, where 1 is a crash with no workaround and 5 is an enhancement request. Priority refers to the importance of fixing the problem. A five point priority scale also works well. A 1 is a bug that must be fixed as quickly as possible—such as one that breaks the build or that is required to satisfy an important customer. A 5 means "fix it when you can."

It is relatively easy to create consistency within your QA organization for severities, because they can be objectively verified. Priorities, on the other hand, are subjective. A misspelling in the user interface may get a 4 for severity, but different cultures will ascribe different priorities to fixing it. Americans and Europeans are happy to give these kinds of bugs low priorities. Japanese customers tend not to be as tolerant and give user interface bugs high priorities. Because of their subjective nature, setting priorities consistently across various members of the team can be difficult.

Fortunately, I learned how to improve prioritization consistency from one of the very best QA managers I know, James Bach. When a code freeze occurred, James would hold a bug review meeting that included all of the major stakeholders involved with the release. In this meeting he would review a sample of bugs (or all of them) to set initial priorities. Because all of the major stakeholders were represented, we could learn when and why support might prioritize a bug higher than product management would or why a senior developer would be so concerned if a certain kind of bug appeared. Although the meetings were rather long in the early part of the QA cycle, they had the advantage of "calibrating" the QA team so that they could more effectively prioritize bugs based on their collective perceptions.

These meetings worked well for our organization because we could quickly come together, review the issues, and move on. They don't work for every team, and when they go poorly a lot of time can be wasted. If you try this approach and find it isn't working, consider an alternative approach that my colleague Elisabeth Hendrickson has used: preset quality criteria.

Preset quality criteria act both as exit criteria and as a prioritization guide. Suppose you define them as MUST, SHOULD, and MAY. Any violation of a MUST is an automatic P1 bug, SHOULD violations became P2s, and so forth. You then have to define the various criteria. You might define MUST as follows:

- The system MUST support 100 concurrent users.
- The system MUST retain all data created in a previous version throughout an upgrade.

- The system MUST present an error dialog only if the dialog contains directions on how the user can fix the problem.

One advantage to this approach is that you get people thinking about priorities (both market and product) long before quality assurance is initiated. Bugs are also managed by exception, with the review committee meeting to handle those that for some reason don't seem to match the preset quality criteria.

and only a long term commitment to the integrity of the product will make absolutely certain such compromises are removed.

The technology base dimension includes the full suite of possible technologies available to the development team. These include the languages and compilers, databases, middleware, messaging, as well as any "uber-tarchitecture" associated with the system—a technical architecture that prescribes the basic structure of many classes of application and that is delivered with an extensive array of development tools to make it easy for developers to create applications within it. Examples of uber-tarchitectures include J2EE, CORBA, Sun ONE and Microsoft .NET (all of which are also marketectures, depending on your point of view).

Choices made within the technology base must support the tarchitecture as motivated by the problem domain. This can be challenging, as developers are people with their own hopes, desires, preferences, and ambitions. Unfortunately, "resumé-driven design," in which developers choose a technology because they think it's cool, is a common malady afflicting many would-be architects and a major contributor to inappropriate architectures. Marketects are also people, and "airplane magazine market research" becomes a poor substitute for the hard and often mundane but necessary market research and decision making that lead to winning solutions.

I have intentionally simplified my discussion of the early forces that shape a winning solution. If you were to ask me about a discrete force not already discussed, such as competitive pressures or requirements imposed by a regulatory agency, I would lump its effect with one associated with the problem domain, the ilities, or the underlying technology. This process is not intended to diminish the effect of this force in your specific situation. To consciously do this would be dangerous and would certainly miss the point. It is imperative that you remain vigilant in identifying the *most important* forces affecting both your marketecture and your tarchitecture.

Creating Results in the Short Run while Working in the Long Run

World-class marketects approach their task from a perspective of time that easily distinguishes them from those less skilled. Instead of listening to what customers want now (easy), they extrapolate multiple streams of data, including current requests, to envision what customers will want 18 to 24 months in the future (hard). To them, the current release is ancient history, and they often use past tense to refer to the features for the next release that are supported in the current tarchitecture as these requirements stabilize—even though this next release may be ten or more months in the future. World-class marketects know that when a feature motivates a new capability or other fundamental change to the tarchitecture they must watch it carefully, for a mistake here may not only hurt their ability to secure future customers but also harm their ability to support existing customers. Envisioning the future on behalf of customers, even when they can't articulate what they want, is the world-class marketect's key distinguishing feature.

Like their marketect counterparts, world-class tarchitects also extrapolate multiple streams of data and envision a technological future that provides superior value to their customers. One of the key reasons certain tarchitectures, such as the IP addressing scheme or the 5ESS phone switch, have provided enduring value is simply that the key tarchitects behind them envisioned a future and built for it.

Projecting the Future

If the marketect and the tarchitect pursue different visions of a future, the total system will fail. You can minimize this risk through a variety of simple diagrams that capture how you want to create your future. I will refer to these diagrams as "maps," even though they do not map what currently exists but what you want to create. These maps are reviewed briefly here and presented in greater detail as patterns in Appendix B.

The first map is the *market map*. It shows the target markets you've identified and the order in which you will create offers for them. (An offering is a bundle of one or more products or services). To make certain you can properly compete for these markets it is helpful to create a *feature/benefits map*, which shows the key features required for each identified market segment and their associated benefits. Variants of these maps are common in product development organizations. A *market events and rhythms map* helps to ensure that the timing of your product releases matches the market timing. Maintained by the marketect, but open to sharing and upgrades by all, these maps are key communication vehicles for the marketecture.

The *tarchitecture map* is the necessary equivalent of the market-related maps. It shows the natural evolution of the tarchitecture in service to the market segments, features, and benefits identified by the marketing team. Essential features that may not be supportable within the existing tarchitecture must be noted as discontinuities so that the changes needed to resolve them can be managed. Alternative, emerging technologies that hold promise for substantially improving the product and/or for opening a new market are shown so that marketects can prepare for these futures.

Examples of discontinuities abound. Consider an application originally designed for a single language. If this language becomes successful, the marketect may include internationalization in her map, but the corresponding entry in the tarchitecture map is often a major discontinuity, especially if the team is not experienced in building such applications. Another example is new business models envisioned by the marketecture. It is doubtful that the original tarchitecture was planned with them in mind, so they should be noted as tarchitectural discontinuities. In a similar vein, known problems with the tarchitecture that grate against developer sensibilities should be identified so that they can be addressed in future revisions.

Although teams can improve their performance by creating any of these maps, the best results are obtained when all are created so that they work together, as shown in Appendix B.

Harnessing Feedback

Marketects typically use the following formal and informal, external and internal feedback loops to ensure that they receive the data they need to make sound decisions:

- Organizing and/or attending user conferences (their own and competitors)
- Reviewing first- and second-line technical or product support logs
- Reviewing feature requests generated by customers
- Interviewing salespeople for features they believe would significantly improve the salability of the product (often referred to as a “win/loss analysis”)
- Meeting with key customers or advisory groups
- Meeting with industry or market analysts

Tarchitects employ similar feedback loops to stay abreast of technological trends. Conferences, magazines, mailing lists, home computers, and insatiable curiosity all provide them with data.

Drawing from different data sources results in divergence between the tarchitecture and marketecture maps described in the previous section.

Fortunately, the creation and ongoing maintenance (e.g., quarterly updates) of these maps are the best ways to prevent divergence and to share data. Other helpful techniques include making the raw data that informs these maps available to both marketects and tarchitects. For example, marketects usually obtain primary market data via user conferences or focus groups. Inviting marketects to these events is a great way of reaching consensus on key issues. Marketects, in turn, should be open to reading the key technical articles that are shaping their industry or tarchitecture, and the tarchitect is a key source for such articles. Note that my goal isn’t to change the naturally different information-seeking and -processing methods of marketects and tarchitects but to make certain that the subset of data used as a source for key decisions are available to everyone on the project.

What if They Say Something They Shouldn't?

One simple and effective strategy for leveraging primary feedback is to ask your developers to work directly with customers. Several of my clients have been Silicon Valley startups. One created a marketplace for intellectual property licensing and for several years ran a user conference to actively seek feedback from customers on current and proposed product offerings. What made this conference unique was that nearly the entire development staff was present to make presentations, conduct demos, and work with key customers. This direct involvement was a key element of the company's ability to build products that its customers truly wanted.

Of course, you might be thinking, "I'm not going to let my developers talk with customers. What if they say something they shouldn't?" This fear may be real—sometimes developers do say things that they shouldn't—but in practice it isn't that big a risk. If you're really concerned, give your developers the following guidelines:

- Don't make any promises on priorities.
- Don't make any commitments.
- Don't talk negatively about our product or our competitors' products.
- Don't say, "That should be easy." It sets expectations too high and can kill any negotiation to have the customer pay for the modification.
- Don't say, "That's too hard." It can prematurely stop conversation about what the customer really wants and ways to achieve this.
- Listen nonjudgmentally. They are your customers, and they're not stupid. They might be ignorant, but they're not lazy. They might have priorities you don't know about. They're neither your fan nor your adversary.

Generating Clarity

A marketect has among his or her primary objectives and responsibilities the generation of a sufficiently precise understanding of what the development team is charged with building so that the team can actually build it. The specific approach for achieving this varies and is heavily influenced by the structures, processes, and outcomes the *total* development organization has chosen in building the system.

There are a variety of source processes and materials to select from. Marketects can choose simple paper-and-pencil prototypes or more formally defined marketing requirements documents (MRDs). In response, development organizations can create models using UML, entity-relationship models, dataflow diagrams, and so forth. Communication between the teams can take place in regular meetings that formally review progress or in daily meetings that track incremental improvements.

Chief among the variables that determine appropriate structures, processes, and outcomes are the size of the team and the number of external interactions it must support. (See [Hohmann 96] for an in-depth description of these variables). Larger projects require a level of formality and detail that would suffocate smaller ones. Other variables, including team culture, are vitally important.

The marketect has similar objectives and responsibilities but for a very different audience. He must make certain the prospective client is clear on how the system will impact its environment. If the system can be extended, such as in a browser with a plug-in architecture, the API must be made available to the appropriate developers. Data sheets outline the broad requirements, while detailed deployment requirements enable customers to prepare for the introduction of the system within their idiosyncratic IT environment. Performance and scalability whitepapers are common for any software that has a server component.

Managing Cultural Differences in Software Development

In the course of my career I've managed several groups of technical and marketing personnel, including those from Russia, Germany, India, Israel, China, Japan, Korea, Poland, Canada, and Mexico. At times I've had to manage a worldwide team focused on the same deliverable.

There are, of course, several challenges in managing a worldwide development organization, and many of them are logistical. For example, it is nearly impossible to schedule a simple meeting without inconveniencing some members of the team—08:00 U.S. PST is 16:00 in Israel. Some development teams have it even harder—12-hour time differences are common in Silicon Valley. Other examples exist in creating or adopting international standards for naming conventions, coding standards, source code management systems, and so forth. Most of these logistical challenges are relatively easy to overcome given a sufficiently motivated workforce.

A bigger challenge, and one that I've found exhibits no ethnically based pattern, is the relationship that a given group of developers have to their software process. These relationships actually form a culture although not the kind we commonly associate with the word. My own bias is heavily weighted toward processes and practices promoted by the Agile Alliance (*www.agilealliance.org*). However, at times I need to subordinate my own preferences to accommodate the dominant culture of the team I'm managing. Thus, while I firmly believe that in most cases iterative/incremental development practices are most effective, sometimes waterfall models are more appropriate, not because they inherently produce a better result but because the culture of the team *wants* them. Marketects and tarchitects both must pay close attention to these potential cultural differences and choose approaches and processes that work for a given culture.

The marketect is critically dependent on the flow of appropriate information from the tarchitect. An unfortunate, but all too common, situation occurs when last-minute changes must be made to customer-facing documentation and sales collateral because the tarchitect realizes that they contain some grave error resulting from a misunderstanding by the marketect. Even more common is when the tarchitect sheepishly informs the marketect that some key feature won't make the release. Printed material must be created weeks and sometimes even months in advance of a product launch, salespeople must be educated on the product, existing customers must prepare for the upgrade, and so forth. The marketect is partially responsible for making certain all of these happen on time and with accuracy.

Working in Unison

I reject the images perpetuated by Dilbert that marketing departments are buffoons and that engineering departments must bear the pain they incur. Instead, marketects and tarchitects should work together to ensure that the total system achieves its objectives. Lest I be misunderstood, I will try to be much more explicit: There is much for each side to gain from a strong, collaborative relationship. While this sounds good, learning to work in unison takes time and effort. Are the potential benefits worth the effort?

Let's first consider this question from the perspective of the marketect. Over the years I've found that marketects routinely underestimate or fail to understand the true capabilities of the system created by the development team. Working with tarchitects or other developers can expose marketects to unexpected, and often delightful, system capabilities. Think about systems that can be extended via plug-ins or APIs. I was delighted when a member of the professional services team of an enterprise-class software company I worked for elegantly solved a thorny customer problem by hooking up Excel directly to the system through the client-side COM API. We had never intended the API to be used in this manner, but who cares? One of the primary goals of creating extensible systems is that you believe in a future that *you can't envision* (extensibility is explored in greater detail in Chapter 8).

Now consider features that can be offered because of choices the development team made when implementing one or more key requirements. In one project I managed, the development team had to build a functional replacement of an existing server. The old architecture had a way of specifying pre- and postprocessing hooks to server messages. Unfortunately, the old architecture's solution was difficult to use and was not widely adopted, so the development team implemented an elegant solution that was very easy to use. Among other things, they generalized the pre- and postprocessing hook message handlers so that an arbitrary number of hooks could be created and chained together. The generalization was not a requirement, but it created new features that the marketect could tap.

A final set of examples illustrates marketing's ability to exploit development tools for customer gain. I've co-opted and subsequently productized developer-created regression test suites for customers so that the operational health of the system could be assessed by the customer onsite. I've converted log files originally created by developers so they could be used as sources of data for performance analysis tools. I'm not advocating goldplating, which is wasteful. But marketects who fail to understand the capabilities of the system from the perspective of its creators lose a valuable opening for leveraging latent opportunities. By establishing strong relationships with tarchitects, marketects can quickly capitalize on their fertile imaginations.

Reflexively, a tarchitect's creative energy is most enjoyably directed toward solving the real problems of real customers. By maintaining a close relationship with marketects, tarchitects learn of these problems and work to solve them. I'm not referring to the problems that the tarchitect would like to solve, that would be cool to solve, or that would help them learn a new technology. I'm talking about the deep problems that don't lend themselves to an immediate solution and are captured on the maps described earlier. Working on these problems provides a clear outlet for the tarchitect's strategic energy.

The following sections describe activities that have proven effective in fostering a healthy working relationship between the marketect and the tarchitect.

Reaching Agreements

Agreement on the project management principles and resultant practices driving the project. A variety of principles can drive any given project. Project leaders select the specific techniques for managing the project from them. Differences on principles and resulting techniques can cause unnecessary friction between marketects and tarchitects which will be felt throughout the entire project organization.

To illustrate, many software projects are driven by a "good enough" approach to quality assurance, but some, especially those dealing with human safety, require much more rigor. These goals motivate marketects and tarchitects to utilize different principles. These different principles motivate different project management practices. Not better or worse, just different.

Identifying and agreeing to the set of principles that drive the project, from the "style" of documentation (informal versus formal) to the project management tools used (MS Project or sticky notes on a shared wall), are an important step toward marketects and tarchitects working in unison. As described earlier, this agreement is also vital to meeting the cultural requirements of the development team.

Making Data Available

Visibility to maps and features is crucial. None of the approaches I've described for capturing and planning for the future are much good if the data are hidden. Get this information into a forum where everyone can share it. Some teams accomplish this through an intranet or a Lotus Notes database. Other teams are experimenting with Swikis, Twikis, or CoWebs with good results, although my own experience with these tools has been mixed and is heavily influenced by team culture. Other teams simply make lots of posters available to everyone. Visibility, in turn, is built on top of a corporate culture founded on trust and openness. Putting up posters merely to look good won't fool anyone. Making a real commitment to visibility—and dealing with the inevitable issues your project team members will raise—is a powerful tool to ensure marketect and tarchitect cooperation.

Context Diagrams and Target Products

Context diagrams are a great tool for keeping the marketect and the tarchitect in step. A context diagram shows your system in context with other systems or objects with which it interacts. It typically shows your system as a "single box" and other systems/objects as boxes or stylized icons, with interactions between systems shown using any number of notations. Avoid formal notations in context diagrams and instead focus on simple descriptions that capture the important aspects of the relationships between the items contained within the context diagram. Context diagrams are not a formal picture of the architecture but a "higher level" shot that shows the *system in the context of its normal use.*

Context diagrams are useful for a number of reasons.

- They identify the technologies your customers use so that you can make certain you're creating something that "works" for their environment. This can range from making certain you're delivering your application using a platform that makes the most sense to ensuring that the right standards are used to promote interoperability among various components.
- They identify potential partnerships and market synergies. One of the most important applications of the whole-product concept is identifying partnerships that create a compelling augmented product and defining a map to a potential product.
- They clarify your value proposition. Clearly understanding your value proposition is the foundation of a winning business model.
- They identify the integration and extension options you need to support in the underlying architecture. A good context diagram will help you determine if you need to provide integration and/or extension capabilities at the database, logic, or even user interface levels of your application. They are a guide to the design of useful integration and extension approaches. (See Chapter 8 for more details.)
- They help you understand what deployment and target platform options make the most sense for your target customer. If you're selling to a target market that is generally running all other applications in house, it doesn't make sense to

offer your part of the solution as an ASP. If your context diagram indicates that all of your partners use a specific technology to integrate their applications, it is probably best if you use it, too.

The marketect must take primary responsibility for the context diagram, although, of course, other members of the team can and should provide input to it. For example, the tarchitect can identify key standards, salespeople may suggest new entries based on how customers use the product, and so forth.

❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑

Chapter Summary

- The marketect (marketing architect) is responsible for the marketecture (marketing architecture).
- The tarchitect (technical architect) is responsible for the tarchitecture (technical architecture).
- Marketecture and tarchitecture are distinct but related.
- Three forces that are particularly influential in the early stages of solution development are the *ilities*, the problem domain, and the technology base.
- To become an architect you have to have extensive experience in the problem space and have worked on systems in this space for a relatively long period of time.
- You should classify bugs along two dimensions: severity and priority. Severity refers to the impact of the bug on the customer. Priority refers to the importance of fixing it.
- Use the patterns in Appendix B to create a strategic view of your product and its evolution.
- Winning solutions are much more likely when marketects and tarchitects work together.
- Context diagrams are an essential tool for creating winning solutions. Learn to use them.

Check This

❑ We have a marketect.

❑ We have a tarchitect.

❑ We have a bug database that classifies each bug according to severity and priority.

- ❑ We have followed the patterns in the Appendix and have created a market map, a feature/benefit map, a market events and rhythms map, and a tarchitecture map. These are in a place that is easily accessible for every member of the team.
- ❑ Developers who meet with customers have been properly trained on what they can and cannot say.
- ❑ The marketect has created a context diagram for our system.

Try This

1. What is the *natural* tarchitecture of your application domain? Do you have the requisite skills and experience to work effectively in this application domain?
2. What are the *specific* responsibilities of marketect? tarchitect?
3. How do the *ilities* match between the engineering/development team and the customer? Are there significant differences?
4. How do you obtain feedback from your customers?

Index