# Beyond Software Architecture

*Creating and Sustaining Winning Solutions*

Luke Hohmann

## Acknowledgments

[need to add more]

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, Ron Lunde, Scott Ambler for their detailed review of the entire book. Some of you were especially critical, and the book is better for your criticisms. Steve Sweeting, Craig Larman, Sandra Carrico, Adam Jackson, Tony Navarette, Chris Reavis, Elisabeth Hendrickson, and Alan Shalloway all provided detailed reviews of one or more chapters. A very special thanks goes to Bob Glass for working with me to create the new title.

Steve Dodds, Lee Sigler, and a whole host of students and colleagues have provided me with inspiration and friendship as I undertook this project. Special thanks to my good friend and publisher Paul Becker, who patiently waited for me to write this book. Paul, thanks for your willingness to keep waiting until I had something to say.

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. Tell me, and I'll correct the error in the next edition!

*"It is not the critic who counts, not the man who points out how the strongman stumbled, or where the doer of deeds could have done them better. The credit belongs to the man who is actually in the arena; whose face is marred by dust and sweat and blood; who strives valiantly; who errs and comes short again and again; who knows the great enthusiasms, the great devotions; who spends himself in a worthy cause; who, at the best, knows in the end the triumph of high achievement, and who, at the worst, if he fails, at least fails while daring greatly; so that his place shall never be with those cold and timid souls who know neither victory nor defeat.*

*— Theodore Roosevelt*

## Contents

# 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 towards 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 one of your customers is going to have a problem with your software. When they contact you for assistance, 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 meet their needs. *Beyond Software Architecture* helps you move beyond software architecture and towards creating winning solutions by discussing a wide range of business issues and their inter-relationship with architectural choices.

*Beyond Software Architecture* presents a unique perspective that is motivated and informed from my experiences in creating everything from single-user programs costing less than $50, software systems used in academic research, utilities used to diagnose and fix problems associated with internally developed systems, and distributed, enterprise-class platforms costing multiple 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 lead engineering, product marketing/product 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 that ties all of this software together is that all of 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, on the other hand, is designed to serve the needs of a well-defined set of users and the businesses who 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. The presentation and discussion of these issues occurs most often in the context of enterprise application software, where I have spent most of professional career. While there is no universally accepted definition of an enterprise application, 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+);
- They have complex deployment and operational requirements;
- While they can be operated as an independent application, they 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, and is certainly not limited to the domain of enterprise applications!

Although I will often refer to software architecture, and discuss things that are technical in nature, my discussions won't be focused 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, there is almost *too* many books on these topics, with the unfortunate side effect that many technical people become so focused on technical details that they lose sight of the business value they're trying to provide.

Instead of focusing 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 how you should integrate branding elements into your solution, reduces the often artificial barriers that can exist between developers and the business/marketing people with whom they work. These barriers prevent both sides from creating winning solutions. I cringe when engineers profess to only taking a "technology" point of view, without due consideration for "business" issues, or when marketers make demands to "get me this feature", without due consideration of the underlying technical ramifications of the same. When either side takes a position without due consideration of the impact of this position, the liklihood 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 be somehow separated from business issues, or that business issues can be somehow separated from technical issues. At best this is simplistically 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 inter-relationship 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 whithers 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 doesn't deliver any sustained 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 and sustaining winning solutions.

*Luke Hohmann*

*luke@lukehohmann.com*

# 1. SOFTWARE ARCHITECTURE

The foundation of a winning solution lies in the architecture that creates and sustains it. To see why this is so, we need a working definition of architecture and an understanding of the impact that architectural choices can have on success. Following this, I'll discuss other aspects of software architecture, including how software architectures are created and how they evolve through use.

## 1.1 DEFINING SOFTWARE ARCHITECTURE

"Software architecture" is a complex topic. Because of this complexity, the concept of software architecture has produced a variety of definitions, each more or less useful depending on your point of view. Here is a definition of software architecture that I used in my first book, *Journey of the Software Professional.*

> "A system architecture defines the basic "structure" of the system (e.g., the high-level modules comprising the major functions of the system, the management and distribution of data, the kind and style of its user interface, what platform(s) will it run on, and so forth)."

This definition of software architecture is pretty consistent with many others[1]. However it does lack some important elements, such as specific technology choices and the required capabilities of the desired system. A colleague of mine, Myron Ahn, created this definition of software architecture. It is a bit more expansive, and covers a bit more ground than my original definition.

> "Software architecture is the sum of the nontrivial modules, processes, and data of the system, their structure and exact relationships to each other, how they can be and are expected to be extended and modified, and on which technologies they depend, from

[1] See, for example, definitions of architecture provided by [Bass], [Larman], [POSA] and others in the references.

which one can deduce the exact capabilities and flexibilities of the system, and from which one can form a plan for the implementation or modification of the system."

We could go on extending these definitions from the technical point of view, but this wouldn't provide a lot of value. More than any other aspect of the system, "architecture" deals with the "big picture". The real key to understanding architecture is to adopt this "big picture" point of view.

## 1.2 ALTERNATIVE THOUGHTS OF SOFTWARE ARCHITECTURE

While the previous definitions of software architecture are useful, they are far too simplistic to take into account the full set of forces that shape, and are shaped by, an architecture. In truth, I doubt that any single definition of software architecture will ever capture all of what we believe to be important. To illustrate, this section raises some issues that aren't often covered by traditional definitions of software architecture but are nonetheless quite important. Unlike the previous definitions, which focus on the "technical" aspects of architecture, consider that these focus on some of the most important "human" and "business" issues – all part of the "big picture".

- *Subsystems are designed to manage dependencies.* Having managed distributed teams that have (literally) spanned the globe, I've found that an important criteria in decomposing subsystems is to create the simplest possible dependencies among different development organizations. By simple, I mean "manageable" based on the people who are creating the system. In my work with my consulting clients, I've found that contrary to "technical" justifications, many architectural choices regarding subsystem design are based on creating easily managed dependencies among groups of developers. The practical effect of these choices is that subsystems are rarely split across development organizations.

- *Subsystems are designed according to human motivations and desires.* Many books on architecture remove far too much of the human element from the architectural design process. For example, architectural patterns provide a wonderful place to begin the architectural design process. But the process of creating an architecture isn't just taking some form of starting structure, such as a pattern, and tailoring it to the needs of the business. Because of this, it is important is to understand the team creating the system. Just as architecture shapes the team, the team shapes the architecture. The hopes, confidences, experiences, dreams, fears, aspirations, preferences, and desires of the team all serve to shape the architecture.

To check this, ask yourself if you've ever worked on a project where you wanted to work on a particular aspect of the architecture because you knew you could do a better job than anyone else (confidence based on experience); you wanted to learn the underlying technology (desire); you thought that doing a good job might earn you a bonus or promotion (aspiration); or, you were concerned that no one else on the team had the requisite skill or experience to solve the problem "the right way" (fear).

**Designing Subsystems to Create a Sense of Wholeness**

Few developers want to be so specialized in their work that all they do is analysis, or design, or code, or fix bugs. More often than not, developers want to be associated with the full range of development activities: working with customers or product managers to clarify requirements, developing analysis and design artifacts, implementing them, fixing bugs, tuning performance, and so forth. I think of this as a deep desire to achieve a sense of "wholeness" or "completeness" relative to our work. This desire has deep implications, and good designs are often chosen so that the teams that are building them can achieve this sense of wholeness.

The concept of wholeness means different things to different groups, and it is important for managers to understand how a given team interprets wholeness. In one application, we considered several alternative designs for the client. I'll use two of the alternatives to illustrate how this team interpreted "wholeness" and how it influenced our choice.

In the first design, one of the teams would be given responsibility for working on the "customer facing" issues and the other team would be given responsibility for creating "infrastructure" components. In the second design, the architecture was structured a bit differently, so that each team would have both customer facing and backend infrastructure components.

While the first design may have been a bit easier to manage on paper, it left the infrastructure team feeling demoralized. They felt that they would not be full participants in the release process. Specifically, they wouldn't be working directly with product management, technical publications, or potential customers. They had done this in prior releases and wanted to keep doing this in the current release and in future releases. As a result, we chose the second design, as it was the only choice that allowed both teams to achieve the sense of “wholeness” that they felt was important. For them, this included interaction with product management and to full participate in the release process.

- *We “give in” to great architectures.* I use the phrase “giving in” to architectural design when an architect or development team subordinates, to the extent that they can, their experiences and expectations about what is “right”, and instead lets the forces of the problem domain guide them in the realization of the architecture. Some people claim this is not a problem, and that they or their team always creates an architecture that is soley based on an objective understanding of the customers problems and how best to structure a technical solution. The operative word, of course, is “best”. Your opinion of “best” may not match mine, and is probably more heavily influenced by my experiences than the problem domain – unless my experie nces are borne from this problem domain. One aspect of “giving in” to a great architecture is continually assessing if the decisions we’re making are designed with the customer and their needs first and foremost in our minds.

- *Beauty is in the eye of the beholder!* We all have many definitions of successful software architectures. While a company may feel that an architecture is successful because it is powering its more profitable product, the developers who are asked to maintain this system may cringe because of its antiquated technical architecture. Alternatively, many truly elegant technical solutions fail for any number of reasons. In addition, we all bring our own opinion on architectural elegance. Two of my most skilled developers had very different philosophies regarding stored procedures in databases. While I’m confident that both of these developers could create a good solution to just about any database related problem you could throw at them, I’m equally confident that the designs would be different and that they could both justify their own design while validly criticizing the other. Few developers can escape the aesthetics of the choices associated with their design decisions.

## 1.3 WHY SOFTWARE ARCHITECTURE MATTERS

Software architecture matters because a good software architecture is a key element of your long term success. To illustrate, here are some of the ways that architecture influences your success. Not all of the following are equally important. What is most important is dependent on your context. All of them are related to your architecture.

- *Longevity*. Most architectures live far longer than the teams who created them. Estimates of system or architectural longevity range from 12 – 30+ years, while developer longevity ranges from 2 – 4 years.

- *Stability*. There are many benefits that accrue from the longevity of a well-designed architecture. One of the biggest is stability. Architectural stability helps ensure that there is a minimal amount of fundamental rework as the system's functionality is extended over multiple release cycles, reducing total implementation costs. Architectural stability provides an important foundation for the development team. Instead of working on something that is constantly changing, the team can concentrate their efforts on making those changes that provide the greatest value.

- *Degree/nature of change.* Architecture determines the nature of change within the system. Some changes are perceived as "easy" while others are perceived as "hard". When "easy" correlates strongly with the "desired" set of changes that improve customer satisfaction or allow us to add features that attract new customers we usually refer to the architecture as a "good" architecture.

  To illustrate, in one application I worked on the development team had created a plug-in architecture that was designed to extend the analytic tools that manipulated various data managed by the system. Adding a new tool was relatively easy, which was a good thing, because it was a major goal of product management to add as many tools as possible.

- *Profitability.* A good architecture is a profitable architecture. By "profitable", I mean that the company that has created the architecture can sustain it with an acceptable cost structure. If the costs associated with sustaining the architecture become too great, it will be abandoned.

This does not mean that a profitable architecture must be considered an "elegant" or "beautiful" architecture. One of the most profitable architectures of all time was and is the Microsoft Windows family of operating systems – even though many people have decried this architecture as being inelegant.

It is important to recognize that the profitability of a given technical architecture often has little to do with the architecture itself. Microsoft has also enjoyed tremendous advantages over its competitors in areas such as marketing, distribution, branding, and so forth. All of these things have contributed to the extraordinary profitability of Microsoft.

This is *not* an argument for poorly created, inelegant architectures that cost more money than necessary to create and sustain! Every care should be taken to create simple, elegant solutions, because in the long run, such solutions cost far less to sustain.

## How Many People Will It Cost To Replace This Thing?

One of the most difficult decisions faced by senior members of the product development team is choosing when an existing architecture should be replaced with a new one. Many factors play into this decision, including the costs associated with sustaining the old architecture, the demands of existing customers, and the moves of your competitors. When you feel that your current architecture could be replaced by a development team of half the size of the existing team in less than one year you should seriously consider replacing the old architecture with a new one. Be careful, for replacing an existing system is often harder than you realize! Don't forget to include the total cost of replacement, including development, QA, technical publications, training, and upgrades. Economically, I've found that a good rule of thumb is that it costs at least 10% of the *total* investment in the old architecture to create a functionally equivalent new architecture. If you're on your fourth iteration of a system that has been created over five years with a total investment of $23M, you're probably fooling yourself if you think that you can create the first version of the new system for less than $2M!

- *Social Structure.* A good architecture works for the team that created it. It leverages their strengths, and can, at times, minimize their weaknesses. For example, many development teams are simply not skilled enough to properly use the programming languages C or C++. The most common mistake is mismanaging memory, resulting in applications that fail for no apparent reason. For those teams who do not absolutely require the unique capabilities of C or C++, a better choice would be a "safer" language such as Java, Visual Basic, Perl, or C#, in which memory is managed on behalf of the developer.

Once created, the architecture, in turn, exhibits a strong influence on the team. No matter what language you've chosen, you have to mold the development team around this language. It affects such as things as your hiring and training policies, to name a few. Because architectures outlast their teams, these effects can last for decades (consider the incredible spike in demand in 1997-1999 for COBOL programmers as the industry retrofitted applications to handle the Y2K crisis).

- *Boundaries are defined through architecture.* During the architectural design process the team makes numerous decisions about what is "in" or "out" of their system. These boundaries, and the manner in which they are created and managed, are vital to the ultimate success of the architecture. Boundary questions are innumerable: Should the team write their own database access layer or license one? Should the team use an open-source web server or license one? Which subteam should be responsible for the user interface? Winning solutions create technical boundaries that support the specific needs of the business. This doesn't always happen, especially in emerging markets, when there is often little support for what the team wants to accomplish and they have to create much of their architecture from scratch, and poor choices lead the development team down "rat holes" from which they never recover.

- *Sustainable, unfair advantage.* This point, which summarizes the previous points, is clearly the most important. A *great* architecture provides a technically sophisticated, hard to duplicate, sustainable competitive advantage. If the technical aspect of the architecture and/or competitive advantage is relatively easy to duplicate, a good architecture will help maintain or extend whatever other factors that may exist to give the company an advantage over its competitors. So, if it is technically easy to duplicate the functionality of your system, you may find yourself pouring more effort into improving performance or usability, as a way to strengthen existing customer relationships and win over new customers.

## 1.4 CREATING AN ARCHITECTURE

A “software architecture” begins as the result of the collective set of design choices made by the earliest team associated with creating the very first version of a system. These early beginnings, which might be sketches on a whiteboard to diagrams created in a notebook – or on a napkin – represent the intentions of this first development team. When their finished with the system, these sketches may not represent reality, and oftentimes the only way the architecture can be explained is through a retrospective analysis that updates these initial designs to the delivered reality.

The initial version of a software architecture is often like a young child: Whole and complete, but immature, and perhaps a bit unsteady. Over time, and through continued use and multiple release cycles, the architecture matures and solidifies, as both its users and its creators gain confidence in its capabilities while they understand and manage its limitations. The process is characterized by a commitment to obtaining honest feedback and to responding to this feedback by making the changes that are necessary for success. Of course, an immature architecture can remain immature and/or stagnate altogether without feedback and the processing of this feedback. The biggest determinant is usually the market.

I’ve been fortunate enough to have been involved in creating a several new product architectures from scratch, and I’ve learned that the way architecture is created in practice is a bit different than the recommendations of many books. These books tell us that when the system is completely new, and the problem terrain somewhat unfamiliar, we, as designers, should do such things as “explore alternative architectures to see what works best”. This is sensible advice. It is also virtually useless.

There are many reasons this advice is virtually useless, but three of them dominate. Although any one of them can be strong enough to prevent the exploration of alternatives, all three are usually present at the same time, each working in its own way, and sometimes in conjunction with the others, to motivate a different and more pragmatic approach to creating a software architecture. Note that these forces are not “bad” – they just *are*.

The first force, is that there is rarely enough time in a commercial endeavor to truly explore alternatives. Time really is money, and unless the first version is an utter disaster, the team will almost certainly need to ship the result. They won’t be able to wait! The strength of this force usually results in hiring practices that demand that the ‘architect’ of the initial system have sufficient experience to make the right decisions without having to explore alternatives. In other words, hire someone who has already explored alternatives in a different job!

The second force, and one that is considerably stronger than the first, is that the nature of the problem and its surrounding context will dictate a relatively small set of sensible architectural choices. If you're building a high-volume, high-speed, web-site, you're going to build it according to a relatively standard architecture. Similarly, if you're building an operating system, you're going to build it using a slightly different, but no less "standard" architecture. There probably isn't a lot of value in exploring radically different high-level architectures, although there is often some value in exploring alternatives for smaller, more focused, portions of the architecture.

The third, and strongest force, is that architectural "goodness" can only be explored through actual use of the system. Until you put the architecture into the situation it was nominally designed to handle, you don't really know if it is the right architecture. This process takes time, usually many years.

## When You Know It Is Going To Fail

As you gain experience in working with different kinds of architectures, you can sometimes spot disastrous architectural choices before they are implemented. Once identified, you can either choose to kill the proposed architecture and send the design team back to the drawing board, or kill the project. Killing the project may seem a bit drastic, but sometimes this is required, especially when the poor choices will substantially change the economic projections associated with a new system.

In one assignment, the engineering team decided that a full J2EE implementation was needed for a problem that a few dozen Perl scripts could have easily handled – provided another part of the application was also modified. In a classic case of "resume driven design", the technical lead, who fancied himself an architect, managed to convince several key people in both development and product management that the J2EE implementation was the best choice. The architecture was also beset with a number of tragically flawed assumptions. My personal favorite was the original design choice of emailing CD-ROM images of up to 600Mb of data as attachments! In the end I became convinced that there was nothing to salvage and I ultimately killed both the architecture and the project. We simply could create and release the desired product within the necessary market window (that period of time when a product is most likely to find a receptive customer).

## 1.5 Patterns and Architecture

The net result of these forces presented in the previous section is that the process of creating the initial architecture must be grounded in a thoroughly pragmatic approach. The emerging discipline of software patterns provides the foundation for such a pragmatic approach. Software patterns capture existing proven and refined solutions to recurring problems in ways that enable you to apply that knowledge to new situations. By "proven", I simply mean that a given pattern has been used effectively in at least a few real-world systems. The best patterns have been used in dozens or hundreds. By "refined", I simply mean that somebody has taken the time to research, understand, codify, and document the pattern so that it can be communicated to other people. It is "digested" knowledge that you can use.

Architectural patterns capture fundamental structural organizations of software systems. Primarily addressing the technical aspects of architecture presented earlier in this chapter, architectural patterns provide descriptions of subsystems, define their responsibilities, and clarify how these subsystems interact to solve a particular problem. By focusing on a specific class of problems, an architectural pattern helps you decide if that kind or style of architecture is right for your system.

The pragmatic approach when creating a software architecture is to explore the various architectural patterns that have been documented and choose one that is reasonably thought to address your particular situation. From there, you must tailor the architecture to meet your needs, ultimately realizing this architecture in a working system. The tailoring process, and the design choices made, is guided by the design principles described later in this chapter. If your familiar with architectural patterns, you might want to read the remainder of this book from the perspective that it can "fill in the gaps" associated with each of them. For example, every software architecture can benefit from a well designed installation and upgrade process or sensibly constructed configuration files. The goal is to use all of these approaches to create and sustain winning solutions.

## 1.6 Architectural Evolution & Maturation: Features vs. Capabilities

Although much emphasis is placed on the initial creation and early versions of an architecture, the fact is that most of us will be spending the majority of time working within an existing architecture. How a given architecture evolves can be more fascinating and interesting than simply creating the initial version of an architecture. It is through evolution that we know where we have success, especially when the evolution is based on direct customer feedback. On a personal note, I’ve faced my greatest managerial challenges, and my most satisfying accomplishments, when I’ve been able to work with software teams in modifying their existing architecture to more effectively meet the needs of the marketplace and position their product for greater success.

This process, which involves both evolution and maturation, is driven by actual use of the system by customers. While many companies claim to be continually requesting customer feedback, the reality is that customer feedback is most actively sought, and most thoroughly processed, when planning for the next release of the system. The next release, in turn, is defined by specifying its required functionality as expressed in the *features* that are marketed to customers. Whether or not these features can be created is dependent on the capabilities of the underlying architecture. The interplay of requested or desired features and the underlying capabilities required to support these features is how architectures evolve over time.

What makes this interplay so especially interesting is that it takes place in the context of continual technological evolution. In other words, customer feedback isn’t always based on some aspect of the existing system. It can be based on reading some announcement on some technology that could help the customer. In fact, the source of the data on new technology doesn’t have to be a customer. Quite often, the source of technology oriented enhancements is the development team. Regardless of the source or context, it is useful to think of architectural evolution and maturation in terms of features and capabilities.

A *feature* (or function; I use the terms synonymously, but prefer feature to function because features are more closely related to what you market to a customer) defines something that a product does or should do. Collectively, the entire set of requested features defines the requirements of the product. Desired features can be elicited, documented/captured, and managed through any number of techniques. In case you're wondering what a feature “is”, here is a definition based on the definition of a function from the book *Exploring Requirements Quality Before Design* by Weinberg and Gause. The authors suggest that “to test whether a requirement is actually a [feature], put the phrase ‘We want the product to ...’ in front of it. Alternatively, you can say, ‘The product should ...’”. This approach demonstrates that a wide variety of requirements qualify as features. Here are a few examples:

- *Supported platforms* ("We want the product to run on Solaris 2.8 and Windows XP").
- *Use cases* ("The product should allow registering a new user").
- *Performance* ("We want the product to provide a dial-tone within 100ms of receiving the off hook signal").

One thing to note is that as descriptions of features use cases have an advantage over other forms of documentation because they can put the desired feature into a specific context. Skillfully written use cases capture both the goals of the actors and the manner in which actors derive value from achieving these goals. A further advantage of use cases is that they are one of the best tools from capturing desired behavior from the perspective of the user. These can then be analyzed by product management to capture the system's essential value propositions use in sales and business development.

Features are most easily managed when clearly prioritized by marketing, and are best implemented when the technical dependencies between features are made clear by the development team. This is because features are usually related, in that one feature often requires another for its operation. As stated earlier, because the system architecture determines how "easy" or "hard" it is to implement a given feature, "good" architectures are those in which it is considered easy to create the desired features.

"Capability" refers to the underlying architecture's ability to support a related set of features. The importance of a capability emerges when marketing is repeatedly told that a class of related features, or a set of features that appear to be unrelated on the surface but are related due to technical implementation details, is difficult or "impossible" to implement within the given architecture. Here are a few examples to illustrate the point.

**Architectural Capability or Feature Agility?**

In one system I managed, users could search and collaboratively organize documents into folders. New documents also arrived from external sources on a regular basis. One set of requirements that emerged concerned storing and executing pre-defined queries against new documents. Another set of requirements that emerged was the notion that when one user modified the contents of a folder another user would receive an email detailing the modification. Both sets of features required the architectural capability of a *notification mechanism*. Until this capability was implemented none of the desired features could be realized.

The primary target market of this system was Fortune 2000 companies, and the original business model was based on a combination of perpetual or annual licenses accessed by named or concurrent user. While these business models are standard in enterprise class software, they prevented us from selling to law firms, who needed to track and bill usage on a metered basis for cost-recovery purposes. Working with my product managers, we realized that we could address a new market segment if we provided support for a metered business models. Unfortunately, it was far easier to define the new business models than implement them, for they required substantial changes to the underlying architectural capabilities. Instead of simply counting the number of named or concurrent users, the development team had to create several new capabilities, such as logging discrete events into secured log files for post-processing by a billing system. Until this was done we were unable to address this newly defined market segment.

A different system I managed was responsible for creating trial versions of software. Trialware is a powerful marketing tool, created by taking special tools and applying them to software after it has been written to add persistent protection to the software. Customers of this system asked for increased flexibility in setting trial parameters. Unfortunately, these seemingly simple requests for increased functionality were not supported by the underlying architecture, and I had to initiate a major design effort to ensure the right capabilities were added to the system.

A common example of the difference between features and capabilities occurs when product marketing requests that the user interface be localized in multiple languages. Each desired language can be captured as a separate feature, and is often marketed as such, but unless the underlying architecture has the necessary capability for internationalization, supporting these languages can quickly become a nightmare for the entire organization. There are many other examples, especially in enterprise applications: workflow, flexible validation rules, increasing demands for customizations, to name but a few.

The interplay between architectural maturation and evolution is a function of time and release cycle. It is uncommon for the development team to implement most or all of the desired features in the very first release, especially if the product manager specifies requested features in a well-defined, disciplined manner. If all of the desired features aren't implemented in the first release they often come along shortly thereafter. Because there aren't a lot of customers to provide feedback, the team is likely to continue working on the leftover features from the first release and complete them in the second. Because the focus of the team is spent completing the total set of desired features, there is usually little time/energy spent on changing the architecture. Instead, the initial architecture matures. Certainly some changes are made to the architecture, but they are usually fairly tame.

After the system has been in continuous operation for three or more release cycles, or for two or more years, the initial set of features envisioned by its creators have typically been exhausted, and product managers must begin to incorporate increasing amounts of direct customer feedback into the plans for future releases. This feedback, in the form of newly desired features, or substantial modifications to existing features, is likely to mark the beginning of architectural evolution, as the development team works to create the necessary capabilities that provide for these features.

Of course, not all architectural evolution is driven by customer demands. Companies who proactively manage their products will look for new technologies or techniques that can give them a competitive edge. Incorporating key new technologies into your evolving architecture can give you a sustained competitive advantage. Chapter four discusses this in greater detail, and the Appendix includes a pattern language that shows you one way to organize this process.

The process of architectural maturation and evolution is cyclic, with each phase of the cycle building on the next. New capabilities are rarely introduced in a mature state, as it is only through actual experience can we know their true utility. Through feedback, these capabilities mature. In extremely well-tended architectures, capabilities are removed.

There is a special situation in which the maturation/evolution cycle must be broken, and where capabilities dominate the discussion of the development team. This is when you must undertake a complete redesign/rewrite of an existing system. Although many factors motivate a redesign/rewrite of an existing system, one universal motivator is the fact that the existing system does not have the right capabilities, and it is perceived that adding them is too costly, either in terms of time or in terms of development resources. In this situation, make certain your architect (or architecture team) clearly captures the missing capabilities so that everyone can be certain you're going to start with a solid new foundation.

The motivation for redesigns/rewrites of existing systems (the lack of underlying capabilities) often finds its roots in the fast paced world of delivering new features to a growing customer base as quickly as possible. Quite simply, success can breed failure when success is not properly managed.

Architectural degradation beings simply enough. When market pressures for key features are high and the needed capabilities to implement them are missing, an otherwise sensible engineering manager may be tempted to coerce the development team into implementing the requested features without the requisite architectural capabilities. There are usually many justifications provided by everyone involved: competitors will beat the team to an important customer, or an important customer won't upgrade to the next version without these features, delaying critically needed revenue. Making these decisions is never easy.

The almost certain outcome of this well-intentioned act is that the future cost of maintaining or enhancing these features will be quite high. Additionally, because the underlying architectural capabilities have still not been added to the system, any other new features depending on those missing capabilities will find themselves in the same predicament. Worse yet, the cost to morale will likely reduce the effectiveness of the team, further compounding the problem - especially if the alleged "market pressures" turn out not to be so obviously pressing in retrospect.

The result is that implementing these features results in the team taking on a technical "debt" that must be repaid in the future. The principal of this debt is the cost associated with creating the right underlying capability. It simply won't go away. The interest of this debt is the additional burden of sustaining a feature in an architecture that can't support it. This interest increases with every release, and increases again as customers continue to demand new features without the underlying capabilities. If things get bad enough – and they can, and do – the team might eventually have to scrap the entire architecture and start from scratch. Servicing the debt has become too high.

While it is sometimes true that there really is a critical market window and a shortcut must be taken, or that you simply must implement a given feature for an important customer, more often than not this is just an illusion fostered by impatient and harried executives looking for a supposed quick win.

As people mature, they often became wary of taking on unnecessary debt. You cannot wish away debt. You have to pay it back. When a given set of desired functions require a significant new or modified architectural capability, avoid incurring debt whenever you can. When it comes to implementing needed capabilities, the phrase "Pay me now, or pay me later, with interest and penalties" truly applies.

### Entropy Reduction: Paying Off Technical Debt After Every Release

In the heat of the battle to ship a product a development team will usually need to make some compromises on the "quality" of the code and/implementation to get the product finished. It doesn't matter if this team is "agile" or not – to get the product shipped you have to focus on shipping the product. This usually means that the team is going to have to put in compromises ("hacks", "ugly code" – you get the picture!).

No matter how they get into the code base, unless these compromises are removed the quality of the source base will degrade. After the product is shipped some reasonable amount of time needs to be devoted improving the source code. I call this process "entropy reduction". A development organization realizes several crucial benefits from the entropy reduction process. A few of these are itemized below, in no particular priority order.

- Maintains a focus on source level quality. Developers know that a very fundamental level source code quality matters. And the company reduces risk associated with an unmaintainable code base.
- Reinforces feelings of good will in the development team. Great developers wince when they compromise the source code in order to ship the product. They'll do it, but they will wince. If you don't let them go back and clean up they will become "dead" to quality. And if their sense of quality dies, so does your products.
- Substantially improves the maintainability and extensibility of the system.
- Allows the team to clearly understand the specific sets of behavior that exist within the system and gets their mind present to the possibilities of really changing it.
- Ensures that any inconsistencies relative to things like the coding standard are addressed.

Entropy reduction is NOT about making substantial changes to the architecture or adding lots of new features. The goal of the entropy reduction process is to hold the external behavior of the system constant while improving the internal structure of the system.

Entropy reduction episodes are usually scheduled after major releases, roughly every 9 – 12 months for most products. Within these release cycles your team should be refactoring their source code as needed. Entropy reduction is often about handling refactorings that were postponed so that you could release your system or avoided because they were deemed too big or complex.

It is important to establish appropriate rules or guidelines when initiating an entropy reduction episode. The biggest, and most important, is the requirement that no new features or capabilities should be added during an ER. To help enforce this rule, try to keep product management from becoming involved. Keep ER as a pure development activity.

Functional tests, such as they may exist, of course need to run. Do *not* ship the product after ER without a full and complete regression test.

Teams vary in their willingness to engage an entropy reduction episode. Teams that embrace it are easy to manage. Teams that enthusiastically embrace it may be trying to use the process to change their architecture—be wary! Teams that have had their sense of quality and their desire to tend their architecture beaten out of them by repeated poor management may have to required to do it ("I know that this ER thing sounds weird, but this is just the way that I do things, and since what we've done in the past isn't producing the results we want we're going to try and start something new").

Before engaging ER the team should create and peer review a plan that details the specific modules and/or functions they're going to improve. This is important because sometimes a team will propose changes that are a bit too big to accomplish in a 1-2 week period of time, which is the amount of time I normally allow for an ER episode. To help ER episodes happen smoothly here are some "best practices" that have worked well for me.

1. Code tags. Code tags are some way of identifying sections of the source code that should be fixed in a future ER episode. I've had teams use "TODO", "REDO", or even "HACK" to alert readers of the source that something should be fixed. Finding the tags are easy.

2. Establish a rhythm. The rhythm of ongoing releases is very important to me. Successful teams develop a nice rhythm. The work at a "good" pace during the majority of the development, like a brisk walk – sustainable but fun. Near the end, they work as needed, including 80+ hour work weeks, if necessary, to hit the release date. The sprint to the finish. After the release date,

the team recovers. We rest, work on ER (which *should not* be mentally challenging). The recovery race.

3. Time box the ER activity. 1-2 weeks works best. I once tried 3 weeks, but the process was too hard to control -- new features were being added. If you need 4 weeks or more to engage a productive ER session look in the mirror and say “The architecture is dead”.

4. Don't ship the result. I really didn't have to say this, right? I mean, you could be making changes all over your code base. Just because your automated unit tests pass doesn't really mean you can ship it without a full regression test.

5. Choose your language carefully. I've the terms “technical debt” and “entropy reduction” to generally mean the same thing. Different people hear these words in different ways, so consider if the phrase “entropy reduction” will work for your team before you use it. If it won't, consider something else, like “technical debt payback”, “architectural improvement”, or whatever else will work best for your team.

6. Use ER to reinforce other values you deem important. I define a value as something deemed worthy or important. You can use ER to instill a whole host of values, including source code quality, the importance of finishing on time, responsible management (“incur a little technical debt now and you'll get the time to pay it back after the release”). You can associate cultural artifacts with the first ER episode, like a “chaos” game or a toy (a symbol).

8. Don't commit unless you can deliver. If you promise an ER episode, make certain you can deliver. I almost lost all credibility in a turn around situation when the CEO of the company told me that I couldn't do an ER episode because he promised a new feature to a key customer. It took some stringent negotiation, including a few heated meetings, but in the end we had the ER episode before we attempted to deliver the promised new feature. Avoid this pain and make certain you have appropriate senior executive buy-in before engaging an ER episode.

## 1.7 Architectural Care and Feeding

In addition to architectural maturation and evolution, which are driven by features and the capabilities that support them, the development team must also “care and feed” their architecture. An architecture is like a carefully designed garden. Unless you tend to its care and feeding, it will soon become unruly, overgrown with the wasteful vestiges of dead code. Ignore it long enough and you'll find that your only recourse is to engage in massive – and very costly – changes to correct the neglect. Architectural care and feeding isn't about adding new features or capabilities; it is about keeping the features and capabilities that you've got in good shape.

To illustrate, consider the following forces that shape the architecture. Many of these are not strictly correlated with architectural maturation and evolution. Instead, these forces are often motivated by other needs. Many of them are discussed in greater detail in subsequent chapters.

- *Technological Currency*. Every complex application interacts with a wide variety of fundamental technologies. Staying current with advances in these technologies as your product evolves ensures that you won't have to engage in expensive redesign efforts. Technological advances are often the key to providing additional benefits to your users. The result? A double-win. It doesn't hurt your marketing department either, as the phrase "new and improved" will actually mean something.

- *Addressing "Technological Debt"*. Developers are constantly struggling to release the system on time while creating appropriate long-term solutions. Successful software teams know when and how to compromise on technical issues in order to hit the ship date. More bluntly, successful software teams know when they need a potentially ugly hack to get the system to a shippable state. The problem with such hacks is not found in the current release (which needed them to ship) but in subsequent releases, when the so-called "compromise" makes itself known, often exacting a heavy toll to fix it.

  These "compromises" are another kind of technical debt, similar to the debt that is incurred when you implement a feature without the underlying capability. Part of architectural care and feeding is paying down this kind of technical debt.

- *Addressing Known Bugs*. Every complex application ships with bugs. Think of them as pestiferous weeds. Leaving them in your garden, especially when they are big enough to be seen, can cause an unnecessary toll on the sensibilities of your development staff. Give them some time to fix the bugs that they know about, and you'll end up with happier developers – and a better architecture. You also foster a cycle of positive improvement, in which every change leaves the system in a better state.

- *License Compliance.* Complex applications license critical components from a variety of vendors. In general, as described above, as they upgrade their technology, you'll respond in kind to keep pace. Of course, sometimes you may not need their new technology and are better off directing your development efforts elsewhere. But watch out: Wait too long and you risk falling out of compliance with your component vendors. Remember to review each upgrade from your component vendors. Know when you must upgrade your architecture to maintain compliance. License compliance is discussed in greater detail in chapter five.

This list is just a starting point, and I invite you to add your own categories for architectural care and feeding. Finally, be careful not to confuse radical changes in feature sets that require similarly radical changes in architectural design with the kinds of changes described above. Scaling an departmental application designed for 100 users to an enterprise application that can handle 1,000 users or converting an application whose business model is based on an annual license to one that is based on transaction fees are *not* the kind of architectural change I'm talking about! Such changes require fundamentally new capabilities and quite substantial changes to the architecture.

## 1.8 PRINCIPLES FIRST, SECOND, AND THIRD

Creating, maturing, evolving, and sustaining an architecture is guided by the set of design choices made by architects and developers. All design choices are not equal. Some are downright trivial, and have no bearing on the architecture. Others are quite profound. All software developers, and architects in particular, must hone their ability to recognize better design alternatives. This means, of course, that we must have a set of criteria for evaluating design alternatives and that we must apply them as best we can. The following principles for good architectural design have stood the test of time.

- *The Principle of Encapsulation.* The architecture is organized around separate and relatively independent pieces that hide internal implementation details from each other. A good example occurs in telephone billing system. One part of the system, such as the switch, is responsible for creating, managing, and tracking phone calls. Ultimately, detailed transaction data are created for every phone call (who called whom, how long, and so forth). These data are fed into separate billing systems, which manage the complexities associated with calculating a precise fee based on the contract that exists between the telephone company and the customer. Within each of these broadly encapsulated systems, there are other encapsulations that make sense. For example, the billing system is likely to have separate modules for calculating the basic bill, another module for calculating any discounts, another module for calculating appropriate taxes, and so forth.

- *The Principle of Interfaces.* The ways that subsystems within a larger design interact are clearly defined. Ideally, these interactions are specified in such a way that they can remain relatively stable over the lifetime of the system. One way to accomplish this is create abstractions over the concrete implementation. Programming to the abstraction allows greater variability as implementation needs change.

  Consider a program originally designed to write output to a file. Instead of programming directly to the interface provided by a file, program to the abstract interface an output mechanism, which in many languages is referred to as an output stream. This allows the program to direct output to a file stream, the standard output stream, an in-memory string stream, or any other target that can be implemented consistent with this interface, even, and especially, those not yet envisioned by the original developer. Of course, what is most important is defining the initial interface, something that can almost only be done well through experience.

  Another area in which the principle of interfaces influences system design is the practice of carefully isolating those aspects of the system that are likely to experience the greatest amount of change behind stable interfaces. The ODBC and related APIs provide an example of this principle in action. By programming to ODBC, a developer insulates the system from a common dimension of change: the specific selection of a database. Using ODBC a developer can relatively easily switch from one SQL-based database to another. And yet, even this flexibility comes with its own special cost – you give up the "goodies" that are present in vendor-specific APIs.

- *The Principle of Loose Coupling*. Coupling refers to the degree of interconnectedness among different pieces in a system. In general, loosely coupled pieces are easier to understand, test, reuse, and maintain, because these operations can be done in isolation from other pieces of the system. Loose coupling also promotes parallelism in the implementation schedule. Note that application of the first two principles aides loose coupling.

- *The Principle of Appropriate Granularity*. One of the key challenges associated with loose coupling concerns the granularity of components. By "granularity", I mean the level of work performed by a component. While loosely components may be easy to understand, test, reuse, and maintain in isolation, when they are created with too fine of a granularity creating solutions using these components can be *harder* because you have to stitch together so many of them to accomplish a meaningful piece of work. Appropriate granularity is determined by the task(s) associated with the component.

- *The Principle of High Cohesion*. Cohesion describes how closely the activities within a single piece (component) or among a group of pieces are related to each other. A highly cohesive component means that its elements strongly relate to each other. We give the highest marks for cohesion to those components whose elements contribute to one and only one task.

- *The Principle of Parameterization*. Although components are encapsulated, this does not mean that they perform their work without some kind of parameterization or instrumentation. The most effective components perform an appropriate amount of work with the right number and kind of parameters that enable its user to adjust its operation. A sophisticated form of parameterization referred to as "inversion of control" occurs within frameworks and plugin architectures. This is where the component hands over processing control to another component.

- *The Principle of Deferral.* Many times the development team is faced with a tough decision that cannot be made with certainty. Sometimes the decision cannot be made with certainty because of technical reasons, as when the team is trying to choose a library to perform a specific function and they are not certain which vendor can provide the best performance. Sometimes the decision cannot be made with certainty because of business reasons, as when the product management team is negotiating with two or more technology providers for the best possible terms.

By deferring these decisions as long as possible, the overall development team gives themselves the best possible chance to make a good choice. While you can't defer a decision forever, you can quarantine its effects by using the principles presented earlier in this section.

## 1.9 CREATING ARCHITECTURAL UNDERSTANDING

Let's return, for a moment, to the definitions of software architecture provided earlier in this chapter. An important criteria for the development team is that they have *some* way to identify, describe, communicate, and modify their understanding of the architecture, without resorting to the source code. One or more models must exist in order for the team to actually operate at the level of the "big picture".

There are a variety of models for doing this, and I've had good success with several of them. Recently, I've adopted the Rational "4+1" model (the references will identify others that you may wish to consider), which captures several of the most useful models in one convenient approach.

Based primarily on resolving the needs of key participants in the software process, the Rational "4+1" view of architecture recommends creating four primary views of the architecture as shown in in Figure 1-1.

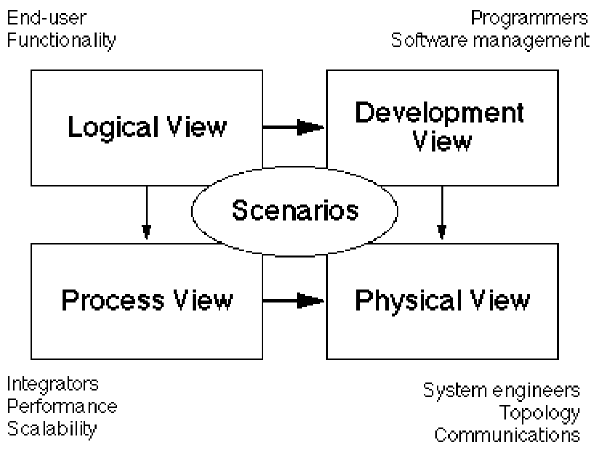


**Figure 1-1**

Philippe Kruchten, the creator of the 4+1 View concept, defines these views as follows:

- *Logical View.* The logical view provides the “static” snapshot of the relationships that exist among objects or entities in the development of the system. This view may actually have two or more representations, one representing the “conceptual” model and the other representing the realization of that conceptual model in a database schema.
- *Process View.* The process view describes the design's concurrency and synchronization aspects.
- *Physical View.* The physical view describes the mapping of the software onto the hardware, including any choices related to distribution of processing components created to achieve such goals as high-availability, reliability, fault-tolerance, and performance.
- *Development View.* The development view describes the software's static organization in its development environment.

Software designers can organize the description of their architectural decisions around these four views. To make the system easier to understand, each of these views can be described in the context of few key use cases (all of the use cases constitutes the “+1” view of the architecture).

In truth, this approach should be referred to as the “n+1” view, because nothing in the approach restricts the architect to just four views of the architecture. Indeed, all of the major disciplines for representing software architectures explicitly recognize the need for multiple views to correctly model the architecture. As near as I can tell, *no one* dogmatically requires that you use exactly *three* or *five* views. Instead, they encourage you to understand the communication properties of a given view and then use this view as needed.

This shouldn’t be surprising, as the concept of multiple models of complex systems is quite common. I have both built and remodeled homes, and in the process have learned to carefully read the many models of the home produced by the architect, from the site plan, to the electrical plan, to the lighting plan, to the elevations, to the plumbing plan. The specialized and skilled workers creating the home need all of these models, and to the extent possible, I want to make certain that these plans will result in a home that I will like.

The value of these plans, like the various viewpoints of architecture promoted by Rational, and others, is that they provide artifacts of relatively enduring value to the team. More generally, the parts of the architecture that are worthy of investing time, energy, and money in capturing deal with the relative stable aspects of the problem, the problem domain, the business model (see chapter [business model]) and the idiosyncratic aspects of the technical solution that exists between all of these things.

## 1.10 The Team

The architectural creation, maturation, and evolution process cannot be separated from the team associated with each release. In this section I will share some thoughts on the interplay between the architecture and the team. A more complete discussion can be found in *Journey of the Software Professional.*

It is natural to first consider the initial development team. As the creators of the first system, it is important to remember that these people are making design choices based primarily on their experience. While subsequent teams must always deal with, in some way or another, the choices made by this first team, the first team has no such constraints (this is one powerful reason why so many developers are drawn to working on new systems). Because of this any new architecture, even one based on an architectural pattern, will reflect the idiosyncratic experience of the team who created it. The number of people involved in creating the architecture, and their relationship with each other, will also profoundly affect the initial architecture. Simply put, there is no way to isolate the "team" from the "architecture".

Experience has shown that creating the initial architecture should be done by as small a team as possible, to minimize communication overhead and to maximize the cohesion of the team. As subsystems are defined, planned, or retrospectively justified, the team will naturally allocate responsibilities to reflect the skills of the team members. Suggestions for the initial team range from as few as three to a maximum of ten. My recommendation is to shoot for three to five people, with one identified as the architect, primarily to keep the team moving (see chapter four for a more complete description of the role of the architect).

With the release of a successful initial architecture, the team often grows to accommodate the demands associated with additional requests for features and capabilities. It is natural to grow the team within the boundaries defined by the initial architecture. Perhaps the user interface, which was once handled by one developer, is expanded to have three. Or, perhaps the database, which originally could be managed by a single developer, now needs two. In this manner subteams spontaneously emerge in a way that reinforces the initial architecture. The advantage to this model is that the original team member, who is now part of the subteam, can be a carrier of the overall design and share it with the new members of the group.

The process of growing the team continues until the team and the architecture have stabilized or until some management-induced limit has been reached. I've read reports that some companies limit the size of any given team to a specific number of people in order to maintain a fluid, open, and collaborative approach to communication. Other companies allow teams to grow as large as needed to meet the needs of the system. I consulted on one defense department contract in which the development team consisted of more than 150 C++ developers, organized into about one dozen subsystems. Although this was a fairly large team, they were extremely effective, primarily because they allowed the demands of their problem dictate the size and structure of their team.

The most important thing to remember is that the team and the architecture of the system they are creating are intertwined. They affect, and are affected by, each other.

## CHAPTER SUMMARY

- Software architecture is focused on the "big picture".
- The structure of the team and the structure of the system are inevitably intertwined.
- The typical software architecture will outlast the team that created it.

## CHECK THIS

- ❑ The dependencies that exist between subsystem components are clearly identified.
- ❑ Each person on the team is working on a subsystem that they find personally interesting.
- ❑ Each person on the team is working in a way that is believed by all to improve productivity.
- ❑ Our architecture is profitable.
- ❑ We know if our current release is focusing on issues of evolution or issues of maturation.

- ❑ We understand the degree of technical debt we've incurred in our system. We can identify such debt (e.g., we have placed special comments into our source code identifying areas to fix).
- ❑ We are in proper compliance with all in-licensed components (see also chapter [licensing]).
- ❑ The architect has articulated the principles that are driving architectural choices.
- ❑ Our team is about the right size to accomplish our objectives, neither too large nor too small

## TRY THIS

1. One way to summarize my position on software architecture is that software architecture is *organic*. This position informs a whole host of management practices, including such things as spending time in each release caring and feeding the architecture. What is your position on software architecture? How does this position affect your own management practices?

2. Here is an exercise that I've found especially helpful when working with troubled development teams, because it helps me identify potential sources of confusion and/or conflict. Ask each member of your team to take a plain sheet of paper and draw the architecture of the system. They have to draw it, meaning they can't just fish out some diagram and print it. The drawing must be done individually, with no talking between different members of the team. When finished, tape each of the drawings to a wall to create an "architecture gallery". What occurs for you, and each member of your team, as you review these diagrams? Do you find congruence? If not, why?

3. Do you have a visual representation of your architecture posted in an easily viewed space? If not, why not? If yes, when was the last time it was updated?

4. Can you enumerate the capabilities of your current architecture?

5. Check out the definitions of software architecture posted at http://www.sei.cmu.edu/. Which of the posted definitions work best for you? Why?

6. What are the specific artifacts, or views, that you have created to describe your architecture? Why have you created these artifacts.

# 2. PRODUCT DEVELOPMENT PRIMER

Many software developers are at a decided disadvantage in creating successful products because they don't understand many basic and important concepts in product management. This chapter outlines many crucial concepts and processes in product management, including defining the role of the product manager. If you're extremely comfortable with basic principles of product management you may find it sufficient to skim this chapter. If not, you should read it carefully, for it is the best

## 2.1 WHAT IS "PRODUCT MANAGEMENT"?

Like software architecture, the concept of product management has motivated a number of people to create a variety of definitions. One definition that I've found useful is as follows:

"Broadly speaking, the product manager has two responsibilities. First, the product manager is responsible for planning activities related to the product or product line. Thus, the product manager's job involves analyzing the market, including customers, competitors, and the external environment, and turning this information into marketing objectives and strategies for the product. Second, the product manager must get the organization to support the marketing programs recommended in the plan."

[Lehmann & Winer, *Product Management*, 3rd ed.]

My own definition is quite a bit simpler: *Product management* is the comprehensive set of activities required to create and sustain winning solutions. Product management, then, is like software architecture in that it is also about "the big picture". In reality, though, the "big picture" of product management is an even bigger picture than the "big picture" of software architecture. Product managers deal with everything from pricing the product to creating data sheets, from defining and educating the sales channel to monitoring the behavior of competitors. The job is no less difficult than that of being an architect, and is some ways considerably harder.

## 2.2 WHY PRODUCT MANAGEMENT MATTERS

Perhaps I've convinced you that architecture matters. Now, a potentially more difficult task is convincing you that product management matters. I'm hopeful that I can do this, because product management is as at least as important as software architecture, if not more so.

Let me start with the blunt approach: technology alone doesn't win. Just because you're first, fastest, best, coolest, or "obvious" does not mean that you will win in the market. Product management matters because simply building the right product isn't enough. A whole host of complex and inter-related activities, from establishing pricing models to building useful partnerships is needed to ensure the total success of a product. If any one of these vital activities are weak the product is at risk of failure.

Successful product management provides the same kinds of benefits as a successful software architecture. Chief among these is profitability, in that successful product management will create truly profitable products and companies. Unlike software, where terminated projects are considered failures, successful product managers routinely subject projects to rigorous examination and terminate those that are no longer likely to meet business objectives. What I find most appealing in this approach is that product managers are proactive in terminating projects. In contrast, I find that many software project managers and architects tend to take a reactive approach – perfectly willing to continue working on a poor project until someone else kills it. It doesn't have to be this way, a topic I will explore later in the book.

Perhaps the most important reason that product management matters, especially from the perspective of the architect and development team, is that product management should represent the voice of the customer. Good product managers engage a whole host of activities, including direct and indirect market research, that put them in the position to guide the development organization. When you ask them what the customer needs, they can answer. And because their answer is rooted in data and experience, you can believe them, and act on their advice in the building of the system. In practice, these data are captured and expressed in such things as business plans and marketing requirements documents (MRDs). However, before the MRD is created or the system built, the successful product manager will have already completed several steps in a successful product development process. Before considering some of the important documents associated with successful product management, let's review a product development process.

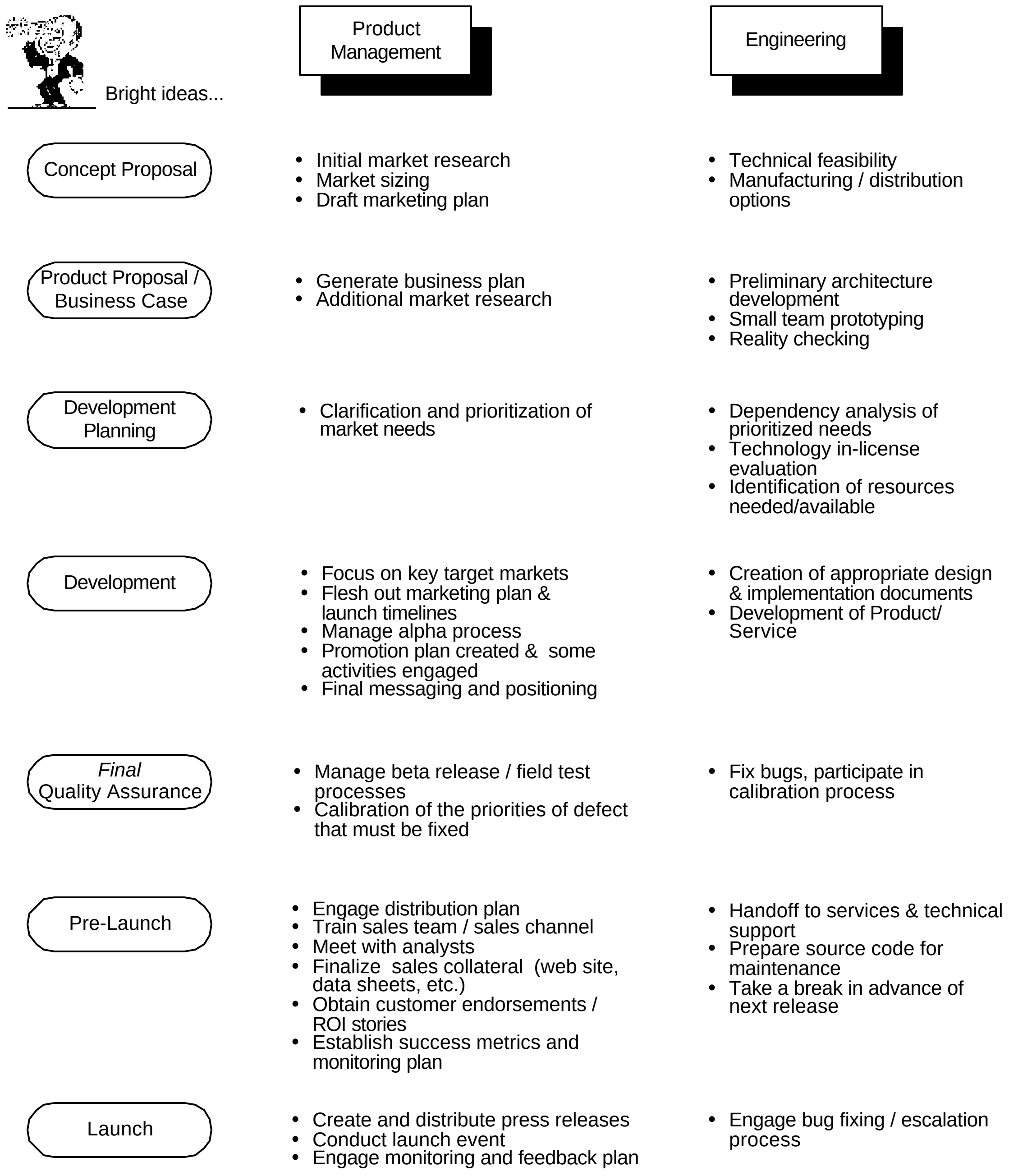
## 2.3 PRODUCT DEVELOPMENT PROCESSES: CREATING RELEASE 1.0

The product development process is the biggest "big picture" perspective that can be taken with respect to the product. My experience is that software development processes, from Waterfall to Spiral to XP, are each more effective when considered and conducted in the context of the more comprehensive product development process. In this section, I will present the conceptual foundation of a successful product development process associated with creating a first release of a product. In the next section, I will show how this foundation can be used to deal with subsequent releases.

The process presented below organizes product development as a series of stages. These stages, and some of the activities of product management and engineering associated with each stage, are presented to the right. I'm choosing to focus only on product management and engineering. Other groups that are associated with the development of a successful product, such as finance, legal, quality assurance, technical publications, are not shown. While these groups are clearly important to the overall product development process, a detailed description of their involvement is beyond the scope of this book.

As you review these steps, keep in mind that the actual size of the product management and engineering team is quite different. A product management team is usually very small in compared with the total engineering/development team. I've worked on products where a team of 2-4 product managers were easily able to keep a team of 30-40 developers busy. Some of my friends report even greater variability! In one example, a friend told me about a *really terrific* product manager that was able to keep a team of 50 developers satisfied.

Another key difference concerns the distribution of work. In the early phases of the project, product management may be perceived to be doing "more" work than engineering. This is partly because at this stage the product management team is producing more tangible results. It is mostly true because in a well run product development process, the early documents created by the product management team are "high stakes" documents. Many future decisions, including decisions that will affect the product for many years, are being made during the development of these early documents, most notably the business plan. Taking care to create good ones is worth the results.



| Bright ideas... | Product Management | Engineering |
|---|---|---|
| Concept Proposal | • Initial market research<br>• Market sizing<br>• Draft marketing plan | • Technical feasibility<br>• Manufacturing / distribution options |
| Product Proposal / Business Case | • Generate business plan<br>• Additional market research | • Preliminary architecture development<br>• Small team prototyping<br>• Reality checking |
| Development Planning | • Clarification and prioritization of market needs | • Dependency analysis of prioritized needs<br>• Technology in-license evaluation<br>• Identification of resources needed/available |
| Development | • Focus on key target markets<br>• Flesh out marketing plan & launch timelines<br>• Manage alpha process<br>• Promotion plan created & some activities engaged<br>• Final messaging and positioning | • Creation of appropriate design & implementation documents<br>• Development of Product/ Service |
| *Final* Quality Assurance | • Manage beta release / field test processes<br>• Calibration of the priorities of defect that must be fixed | • Fix bugs, participate in calibration process |
| Pre-Launch | • Engage distribution plan<br>• Train sales team / sales channel<br>• Meet with analysts<br>• Finalize sales collateral (web site, data sheets, etc.)<br>• Obtain customer endorsements / ROI stories<br>• Establish success metrics and monitoring plan | • Handoff to services & technical support<br>• Prepare source code for maintenance<br>• Take a break in advance of next release |
| Launch | • Create and distribute press releases<br>• Conduct launch event<br>• Engage monitoring and feedback plan | • Engage bug fixing / escalation process |

**Figure 2-1: Product Management and Engineering Processes**

- *Bright Ideas*. Clever ideas form the foundation of the initial product. The best of these fill us with enough passion to take the considerable leap necessary to try and create a product. Some companies actually formalize this aspect of the overall process in a sub process called *ideation.*

- *Concept Proposal.* The purpose of the concept proposal is to establish sufficient motivation for creating the product. Usually created by a small team, the concept proposal includes enough business data to justify the market and enough technical data to justify feasibility. If neither of these conditions are met, the project is gracefully terminated, a process that I will describe in greater detail later in this chapter.

- *Product Proposal / Business Plan.* The product proposal / business plan is the *key* document that is created by the team to justify the product. It outlines a number of important elements so that the business can be certain the project is justified. It is so important that I will include a detailed discussion of this document later in this chapter. Note that during this phase of the product development process it is not uncommon for engineering to be doing literally nothing while marketing is creating the business plan.

- *Development Planning.* Upon approval of the business plan, the two teams enter the highly collaborative phase of development planning. In this phase, the primary responsibility of marketing is to clarify and prioritize market needs, expressing them as desired functions of the target product. In turn, the primary job of engineering is to analysis the dependencies that exist within these functions, identify necessary architectural capabilities, create some crude estimates as to time required to complete various tasks, evaluate technology solutions, and begin to identify needed resources. Marketing should also take primary responsibility for including other teams, as needed, in the development of the product.

  The results of this process can range from a formal Marketing Requirements Document (MRD) to an informal set of user stories written on 5x8 notes and taped to a wall. Indeed, in one system I managed, we generated requirements by simply directing questions to a very experienced member of the team, secure in our knowledge that this person would answer each question with unfailing consistency. While this was quite an unorthodox manner of obtaining and managing requirements, it worked great. This example should illustrate that the development team should be comfortable in letting the size of the project, the size of the team, and the complexity of the product that they are creating dictate the formality of the development planning process. Regardless of their ultimate form, requirements only need to specified to a sufficient level of clarity that the development team can engage in useful work.

In this stage the development team will also create whatever analysis, design, or other pre-development artifacts they feel are appropriate given their project and their chosen development process. The challenge is, of course, in determining those artifacts that are needed for success. Sometimes informal processes and artifacts work great, as when a development team sketches the design for a new user interface on whiteboard using an existing system as a model. At other times formal processes and artifacts are required, as when a development team is working on a hardware device and must create a solution within a narrow set of weight, power consumption, and size constraints. Just about every development team should be conducting some experiments based on the continued advancement of various web-based collaborative technologies. It is quite surprising to realize how easy it is to improve communication within geographically distributed teams with just a web site and a digital camera!

A useful description of a design document, and the value it provides to the team, is found in Alistair Cockburn's book *Agile Software Development*. In this book, Alistair refers to design documents as the "residue" that is created by the development team in their pursuit of actually creating the system. The goal is to create the minimum amount of residue needed for current and future product successes. I've found this perspective useful, as it helps me determine if a given design document is worth creating. That said, you must take care in your definition of development team. My own definition explicitly *includes* technical publications and QA. These teams can both contribute to and benefit from their inclusion in the development planning process.

Disciplined development organizations usually engage one final review meeting at this point before development is engaged. The purpose of this review is to assess the likelihood that the product can be created within time and cost estimates and to systematically kill the project if it cannot. While it is relatively rare to find organizations killing projects after this stage (most are killed after the concept proposal, many are killed after the business plan), a well-run product management organization will stop work when it becomes clear that the product cannot be created in a way that provides for a sustainably profitable product.

- *Development*. The next phase of the process deals with development, or actually building the system. I find this stage quite interesting because product management now focuses their attention on what happens *after* the product is released, before it is even built. Engineering, on the other hand, actually creating the project. There are a wide variety of options, here, including traditional development methods and the newer Agile methods, such as XP, SCRUM, and Crystal. Many authors have written a great deal about development and development processes, so I won't go into the details. What I will say is that you should adjust your development process according to a variety of factors, including the size of the team, its geographical distribution, and the nature of the product you're building. A complete discussion of how these, and other, variables, can affect the team and its processes, can be found in my book *Journey of the Software Professional.*

Modern development practices mandate that the delivery of working systems should happen in reasonably well-defined pieces that cascade or overlap. Specifically, development *does not* happen "all at once". Instead, development is staggered or overlapped so that working versions of the system can be delivered to QA, technical publications, alpha testers, key customers, and other important stakeholders. One example of such an "overlap" is when a working system is given to QA to perform initial testing and to refine final test plans while the development team continues to work on the next set of deliverables.

- *Final Quality Assurance.* The next key phase of the development process is final quality assurance. One of the functions of the quality assurance process is testing: Checking that the system created reasonably matches the requirements and providing data to the product manager to help him or her determine if the system is "good enough" to ship to the customer. In this capacity, quality assurance provides "risk assessment" data to product management, who ultimately makes the decision of whether or not to ship the product. This decision cannot be made unilaterally, and is best conducted through as many meeting as are needed, each meeting being chaired by product management in which the priority of bugs are considered in light of overall product objectives.

  World class quality assurance is associated with more than just testing. Quality assurance is a collection of activities that includes, among others testing, process improvement and enforcement, metrics collection, and root cause problem analysis. If you're quality assurance group is *only* doing testing, be careful of their advice, for they may not be skilled enough to usefully participate in business risk assessment.

  Since modern development practices mandate QA involvement during development, you might argue against the need for final quality assurance. While this sounds good in theory, in practice every complex system that I've worked on has required a final quality assurance pass before delivery to a customer.

- *Pre-Launch*. On or near the successful completion of the system the development process moves into the pre-launch phase. Depending on the estimated time of the final quality assurance phase, this can happen in parallel with this phase or after it has been engaged. One way to characterize this phase is that the intensity of work associated with the engineering team begins to decrease while those associated with product management begin to increase. The engineering team prepares to hand off their work to services and support, prepares the source code for maintenance and future development, and often takes a short break to catch up on sleep. Product management, on the other hand, is often a whirlwind of activity, doing a variety of things to ensure that the product is successful in the marketplace. These include everything from preparing sales collateral to meeting with key analysts to make certain they understand the new product.

- *Launch*. While engineering often has a celebration party to coincide with the creation of the "golden master", the product management team usually waits until there is a launch event. This may be as simple as issuing the official press release, or as complex as staging a special event managed by a professional public relations firm to promote the new release. A final, but critical step in the launch phase is to engage some kind of customer focused monitoring and feedback. When engineering returns from the launch party, they should make certain that they are prepared to address issues and escalations coming from the field.

## 2.4 IT ISN'T LIKE THAT

Experienced readers may feel that this presentation of the product development process has several flaws. Rather than allowing any of these flaws to prevent or distract you from reading further, let me try to address them now.

- *It is a Waterfall process. And we know that these don't work*. While the process presented above are indeed reminiscent of a "Waterfall" development process, there are several substantial differences. The most important of which is that the traditional waterfall process was described in the context of the development phase and related primarily to engineering, not product management. The traditional stages associated with the waterfall ("requirements", "analysis", "design", "implementation", and "test") rarely had sufficiently tough evaluation criteria applied to the work products to properly fix or address them if they were found lacking in some key measure.

Companies that have implemented effective product development practices are most well known for the manner in which the results of each stage are subjected to strictly defined “go/kill” criteria. The overall process is referred to as a “stage gate” process: After each stage, there is a gate. If you do not meet the criteria established by the gate, the project is killed. These criteria are usually precisely defined financial criteria, which differ dramatically from the very loosely defined criteria associated with results of a waterfall process.

The process of carefully examining and then terminating a project is what most strongly distinguishes a stage gate process from a waterfall process. Software developers often mourn the termination of a project. Product managers who are using a stage gate process celebrate it, because it means that the overall process is working. The proof of this is that extremely well run product-focused companies may even kill a project during the beta testing process if it finds that despite prior market research and acceptable development costs the product still does not meet the needs of the market! Examples of this abound in the retail sector, when countless thousands of new food products and packaging ideas are introduced into small, target markets and then subsequently killed before being launched into larger markets.

- *It presents each stage as if it were of equal importance*. This is not my intention. While all of the stages are important, the two most important stages are the concept proposal and the product proposal / business plan. To a large extent, the results of these stages drive the rest of the process. It is product management’s job to do the hard work associated with these stages to ensure lasting success. Ultimately, this means creating a sober, credible concept proposal and business plan that demonstrates that the proposed product can be a profitable endeavor.

- *It doesn’t detail any time*. The amount of time associated with each stage is far too variable to make any estimates as to how long any one stage should take.

- *Where is the iteration?* When software developers think of iteration, they tend to think of iterative construction practices, like XP or SCRUM. When product managers think of iteration, they tend to think of a variety of techniques that enable them to sharply define the product before construction begins. These are things like primary and secondary market research, focus groups, and test marketing on prototypes. The differences and their effect on the project are profound.

The single biggest differentiator of success in new product development is the degree of homework conducted before construction is initiated. A successful product will have a full and complete business plan, with clear and sharp definitions for the product, its core feature sets, its target market, the planned-for business model, and so forth. Of course, any of these may change during construction based on new data. But, before construction begins, the product (what is being built, who it is being built for, the essential features it will contain, and so forth) must be defined.

I am *not* advocating a return to Waterfall construction practices. My own experience with these is that they don't work. Perhaps the following analogy will help me make my point. Imagine that you are an explorer and you've just crossed a mountain pass and have entered a strange new land. At your disposal you have a variety of gear that can help you navigate mountains, valleys, deserts, rain forests, deep lakes, fast-moving streams, and so forth. You have also brought along a small, single person helicopter for reconnaissance. Do you just jump in and start exploring or do you fire up the helicopter, take a look around, and plot your course?

Good explorers fire up the helicopter and plot their course. They know that to the North lies mountains, in the East is a lake, the West has a desert, and the South a rich and deep forest. Armed with this knowledge, they can plot a course: first South, then West, then East, and then finally, North. Along the way they can prepare for the gear that they will need, handle detailed issues associated with the terrain as they are navigating it, and build detailed knowledge through exploration.

The same goes for product managers. The foundational definition of a product could be 48 high-level uses cases, organized in 6 groups. While none of these use cases are defined in a detailed manner, their collective structure provides a coherent definition of the desired product. Before any single use case or group of use cases is constructed, the product development team (product management and engineering) detail them (generate more detailed requirements, create or update necessary design documents, and so forth) so that they can do a good job creating the system.

- *It doesn't prescribe a development process*. That's correct: The product development process described above does not prescribe any particular development process. Personally, I prefer to use development processes based on agile development approaches. As a manager, I prefer to use the development processes that my team believes will help them be successful. From the perspective of the product manager, the overall process associated with the product development function, such as whether or not he or she is required to create a concept proposal and the manner in which it will be evaluated, is usually more important than the idiosyncratic development process followed by the development team.

- *It does not identify the level of collaboration between groups within stages*. So what? You know that collaboration between product management and engineering/development is essential to creating winning solutions. There should be *constant* communication among these groups. Figure 2-1 shouldn't have to show this.

## 2.5 THE BUSINESS PLAN

The business plan distinguishes itself from all other documents associated with the product. It is the document that forms the foundation for the entire work associated with the product. It does not guide a specific release, but an overall family or set of releases.

Product managers have the responsibility to prepare a business plan that justifies the development of the system. Developers have the right to receive such a plan, so that they know their efforts are not in vain. Note that the creation of the business plan usually precedes the creation of the first architecture. In this sense, then, you can expect a business plan associated with the *first* release, but not necessarily *every* release.

A well written business plan is rarely more than 15 to 20 pages and includes the following kind of information.

| **Topic / Section** | **Description** |
|---|---|
| Executive Overview | Brief (two paragraph) overview the entire plan. A good rule of thumb is that a harried senior executive should be able to read and understand the basic contents of the business plan within 5 minutes. |
| Strategic Fit | Describe what is to be accomplished. For new products, show the proposed products into the current strategy of the company. For product line extensions (or new releases of an existing system) show how the strategy is further enhanced. |

| | |
|---|---|
| Market Analysis | Who will buy, why will they buy, how will they use this product and what is the anticipated size of the market? Describe customers by segments (e.g. researchers, lawyers). Specify target companies (Fortune 500 etc.), specify target market. Market segmentation and market analysis are so important that I will discuss them several times over the course of the book. |
| Financial Analysis | A simple financial analysis should be included that summarizes detailed financial information found later in the plan. The financial analysis must address, in format and content, the set of criteria required by the company building the product. |
| Product Description | Briefly describe the product, with an emphasis on target market benefits. If this is a new release describe the most essential new functions. This is not an MRD, which details requirements, but instead more of an overview that demonstrates key criteria from the perspective of the marketplace. |
| Competitive Analysis and Product Differentiation | Provide an overview of the competitive landscape. I find it helpful to categorize competitors and then perform a SWOT analysis of both the category and key competitors within the category. The end result is specific statements that outline how your product will compete. (A SWOT analysis details the perceived Strengths of a given competitor, their perceived Weaknesses, Opportunities that your company / product can exploit, and Threats that your competitors can exploit against you). |
| Product Positioning | Describe the relationship of this product, relative to both existing products and the target market. Positioning is especially important because it guides what will be conveyed to the target market through various promotional activities (discussed further, later in this chapter). |
| Marketing Strategy | How will this product be promoted and sold? In the initial development of a business plan it is usually sufficient to merely state if the release is going to be a big, noisy event going to all customers, or a low key, managed event going to a specific subset of customers (as in a patch or maintenance release). |
| Channel Strategy | The channel is how the product reaches customers. This section of the business plan should detail such things as how the product will be sold (direct or indirect, via the web, and so forth). |

| | |
|---|---|
| Support Model | How will this product be supported? How will the customer come up to speed on the use of the product? |
| Impact Analysis | What impact will this product have on other products? |
| Business Model | The proposed business model, including pricing, should be provided. |
| Revenue Forecast | A simple but believable forecast of revenue should be provided. Trust your instincts on this, and learn to cut projections in half. Most marketing plans are overly optimistic on revenue and growth. |
| Cost Analysis | Provide an estimate of engineering, support and marketing costs. Estimate both non-recurring and recurring costs. |
| Critical Risks | List any risks that may prevent you from completing your objective. Include dependencies with other projects, critically needed but not as yet allocated resources, and so forth. |
| Product Extensions and Futures | Detail key extensions to this product. Show that it has a future. I recommend using a variety of roadmaps, discussed in chapter 4 and again in Appendix A. |

## 2.6 PRODUCT DEVELOPMENT PROCESSES: CREATING RELEASE N.N.N

Like software architecture, we spend a lot of time and energy in creating the first release, when in reality most products spend the bulk of their time in subsequent releases. The primary changes to the product development process presented in section 2.3 are as follows.

- *The concept and product proposal are likely to be skipped.* The value of the product proposal is substantially less as a product matures. This is because there is less work associated with identifying and targeting a given market and more work associated with identifying how to acquire the next adopter segment (see below) or the next niche within a defined market. Rigorous companies may spend some time writing a product proposal, but this is usually foregone in favor of the MRD.

- *The business plan is, at best, updated to reflect the new release.* The business plan that justified the initial development is often only marginally updated to reflect the new release. This is not a flaw, but a reflection of the higher level, broader perspective of the business plan.

- *The MRD becomes the central document of the release.* The MRD, which in the first release captured only those features essential to enter the market, becomes the central document in a subsequent release. To understand why this is so, think of the product in its full context. This context, which can be likened to an ecosystem, includes internal and external stakeholders (developers, QA, marketing communications, partners, customers, and so forth) as well as influencers (trade press, industry analysts, competitors, and so forth). The ecosystem of Release 1.0 is likely to be relatively simple, especially for a technology product. The ecosystem of a successful product matures with the product, and the ecosystem associa ted with Release *n.n.n* is usually considerably more complex than the ecosystem associated with Release 1.0. Furthermore, the needs of the constituent stakeholders are different, as described further, below.

- *The Pre-Launch and Launch phase will vary significantly depending on the release.* Depending on the nature of the release, the pre-launch and launch phases may be more important for Release *n.n.n* than Release 1.0! Consider the case of Microsoft promoting Windows XP, Apple promoting OSX, or Sun promoting Solaris 2.8. The launch of each of these operating systems were substantially larger and “more important” than the launch of their predecessors.

## 2.7 AUGMENTING THE PRODUCT DEVELOPMENT PROCESS

A common concern, and one that I share, is that a sensible product development process will become bogged down in useless detail or busy work. To prevent this, I’ve found it helpful to augment the development process described with a few key concepts and processes, as described in this section.

- *Successive Freezing.* In the early stages of development I’ve found it helpful to give the responsible team *maximum* flexibility in managing their work products. However, when the time comes for others to make *committed* decisions on these work products, they must be *frozen.* As a result, the concept of *successive freezing* various deliverables becomes important.

  In this approach, various deliverables become frozen over the lifecycle of the product.

While product managers may examine a wide variety of markets for a proposed idea in the concept phase, they must usually narrow this down to a specific target market in the business plan. For this release, then, the target market is *frozen*. The target market may be then further stratified into niche markets, with one niche focused as the target for the launch activities.

On the development side, you may choose to freeze the requirements, then the user interface, then the database schema, the APIs that exist among subsystems, and finally the code itself. It is important to note that the concept of freezing is not designed to prevent change. Instead, the term "frozen" is used to describe an decision that can only change through a relatively formal change-control process, described next.

- *Change Management Protocols.* Change protocols refer to the degree of formality associated with changing a given outcome. The most lenient kind of change protocol is *no* change protocol. An example of this is when a developer is free to check out source code, make changes, and check this source code back into the source management system. A formal change management protocol exists when a proposed change must be approved by a given committee.

  I've found that many developers become uncomfortable with change management protocols. This is because they often misunderstand the fundamental purpose of a change management protocol. The goal of a change management protocol *is not* to stifle creativity (as expressed in desired changes to the product). Instead, the fundamental goal of a change management protocol is to ensure that the right set of people are informed of the change before it is made so that they can properly prepare for the same.

  Suppose you want to change the layout of the user interface. Before the user interface is frozen, you're free to change it as you see fit. Once the user interface is frozen, changes need to be managed, if for no other reason than you need to coordinate changes to the help system, the end-user documentation, and the training materials that describe the system. And there might also be changes required in the automated unit tests created by QA, and perhaps updated reference materials for technical support. That's *several* different groups that must be notified of a potentially simple change to the user interface!

  More generally, the change management process must include those stakeholders affected by the change to make certain that they understand, approve, and can correctly process the proposed change. Candidates include technical publications, QA, product management, technical support, and release engineering. In product-centric organizations change management meetings should be organized and chaired by product management. In other organizations change management meetings should be organized and chaired by the project or program manager running the overall project.

- *Recycle Bin.* At several times during the project the team may find that they've bitten off more than they can chew. Rather than throw out potentially useful ideas, I recommend creating a "recycle bin" where these ideas can be stored for future use. A given idea may resurface in a future release, or a future iteration. In a sense, the recycle bin also functions as a "pressure valve", reducing (at times) the pressure felt by the development team to get every requested feature in the release.

### Documentation Balance

One of the most important things that every manager must learn for him or herself, in their own way, is that there is simply no universal formula for creating the right set of documents needed for a given project. I've created very successful systems using agile methods and a minimum of documentation. I've also had projects fail for lack of key documentation. At times I've taken great care to build precise requirements reminiscent of a Waterfall process, with the kind of success that makes you wonder why *everyone* doesn't simply adopt Waterfall and be done with it. Finally, I've also produced carefully written and researched documents only to find them producing a system that no one needed or wanted.

Lest you think there is no hope and you should just randomly pick a process, here are two guidelines that have worked well for me. Early releases should be created using a process that is as agile as possible. While you and your team may have a lot of experience in the proble m domain, you don't have experience with your new product, how your target market and competitors will respond to your offering, or whether or not the business model you've chosen is the best business model possible. Agile processes are often the best way to maximize feedback from the market. Over time, as the product solidifies and the market matures, your going to be spending more time responding to the needs of the market, including the markets need for reliability and predictability in deliverables. Thus, development organizations of mature products often find a transition from agile methods to more formal methods to be appropriate. This transition may sound easy, but it isn't, and many organizations face severe, and sometimes fatal, challenges in making this transition.

It is vitally important is to make certain you understand how your team wishes to develop their product. Surprisingly, I've worked with teams that completely reject Agile methods. These teams consider Agile processes the domain of "hacks" who "don't know how to specify and build reliable software systems". These teams demand MRDs and related documents; they have a process; and, they follow it. As you can guess, I adjust my own preference for agile processes to meet their needs. Forcing a given team to adopt an approach, either in their development process, or the language they're using to create the system, that they just don't believe in, is a certain recipe for failure.

## 2.8 CRUCIAL PRODUCT MANAGEMENT CONCEPTS

This section addresses some vitally important concepts in product management and product marketing. Some of them will be referenced in future chapters. All of them are useful to know, because all of them affect, in very serious ways, your system and its architecture.

### 2.8.1 THE "FOUR PS" OF MARKETING

The activities involved in product management is often summarized by the "Four P's of Marketing". Because you may hear these terms when working with product management, it is useful to understand what they mean.

- *Product (Offering).* What you're offering to your customer. It could be a product, it could be a service, or, most likely, it is some combination of a product and service. Later on in this chapter, and in chapter four, we'll discuss the concept of a product in greater depth.

- *Price and the Business Model.* Your *business model* is the manner in which you charge customers for your products or services – the way you make money. A *pricing model* defines how much you will charge. Selecting a business model and defining its associated pricing model are among the most challenging areas of product management.

  The best business models charge customers in a way that is congruent with the value they perceive with the product. The best pricing models maximize the revenue for the company without leaving the customer feeling they've paid too much. Charge too little and you're leaving money on the table. Charge too much and you're product won't grow and you're leaving yourself vulnerable to a competitor. Pricing can also be counter-intuitive. Pricing is not correlated to technical difficulty. Features that may be difficult to implement, like a sophisticated user interface or report generator, are not always features that you can charge for. Pricing is not easily correlated to cost. It may cost you a lot of money to license a search engine, embedded database, or real-time operating system, but chances are you won't be able to pass these costs directly to your customer. Instead, they will become buried in your pricing model.

Effective pricing is related to the perceived value of your product. The implications of this are profound, and deal more with psychology than technology. A complete discussion of pricing, which may or may not affect your architecture, is beyond the scope of this book. Business models, which form the foundation of pricing, are intimately associated with your architecture, are discussed extensively in chapter 6.

- *Place (Distribution Channel).* The manner in which you're product or offering is delivered to your customer. There are several perspectives on the channel for software products. One perspective concerns how the bits are delivered to the customer (e.g., via the web, or perhaps on a DVD). Another perspective concerns who is authorized to offer the product/service to a customer. Consider an enterprise application that can deployed at a customer's site or through a third party service provider. In this case, the service provider is acting as a channel for the product. All of these choices affect your system architecture. `

- *Promotion (Advertising, Marketing Communication).* Promotion refers to the full set of activities associated with increasing awareness of your product within your target market. It is often easiest to think of promotion as just advertising, because advertising is arguably the most exciting promotional activities. When you put aside sheer excitement, other promotional activities are often more important.

### 2.8.2 Total Available Market, Total Addressable Market, and Market Segmentation

The *total available market* is *all* of the customers who could possibly use your good or service. The *total addressable market* is that subset of the total available market that you can reach. An effective marketing program will further divide the total addressable market into well-defined market segments. A *market segment* is a group of customers that share specific characteristics, chief of which is that they must communicate with each other. A very effective marketing program may further divide market segments into *niche markets*, which allow for very targeted promotion and sales activities.

Here is an example of these concepts in action. Let's suppose that you have created a web-based self-service benefits management system that allows employees to manage their 401K accounts, health care provider, vacation days, and other benefits. Your total available market is all of those companies that could possibly use this system (presumably, a large market). These companies will range from extremely large Fortune 500 companies to very small (less than $5M in revenue) private companies.

The requirements associated with these companies are very different. As a result, you may choose to define a market segment of public companies with between $25M and $50M in annual revenue. This influences all aspects of the marketing mix, including the pricing, promotion, distribution, product, and sales models.

Unfortunately, this is still a very large market segment – probably too large to service in the early releases of the product. Narrowing further, you might define one or more niches within this segment. A common approach is to divide based on vertical industries: pharmaceuticals, automotive, chemical, technology-hardware, technology-software, and so forth. Another common approach is to divide based on the strengths of your product relative to your competitors. Suppose you're creating a new email client that has been extensively usability tested for novice computer users and that also has extensive anti-Spam controls. Chances are good that your competitors are targeting this market, and by virtue of their longevity, claim that they are the most user-friendly email client. By segmenting the market into people who care about Spam, you can differentiate yourself from your competitors by promoting your anti-Spam features, thereby creating and owning a niche market. Once you've achieved success in this niche, you can expand into others.

### 2.8.3 S-SHAPED CURVE OF ADOPTION

Countless numbers of studies have found that the adoption of new products, generally referred to as *innovations*, follows the S-shaped curve shown Figure 2-2. Note that an innovation doesn't just mean a "new" product. It also includes new releases or new versions of existing products. Parts of this curve have been labeled to reflect common characteristics of adopter categories. Understanding these categories can help product management and associated marketing functions change the shape of the curve, which varies tremendously by the type of innovation. Simply put, some innovations are adopted much more rapidly than others. The differences between categories are not perfectly defined, but they do represent generalizations that accurately characterize the willingness of a given person to adopt an innovation. As will be addressed later in the book, various adopter categories place quite different demands on your system and its associated architecture.

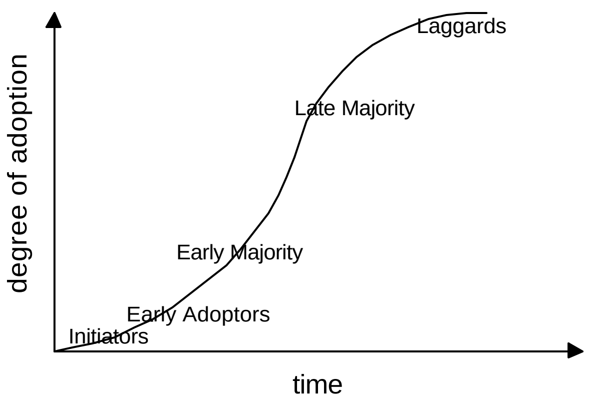


**Figure 2-2: The S-Shaped Curve of Adoption**

[Fix initiators]

I. Innovators. The very first individuals to adopt an innovation are known as *innovators*. These people are usually more technically curious and enjoy “pushing the boundary” with new products and services. They often have the necessary resources at their disposal to “make” an innovation work. To illustrate, innovators are tolerant of poor installation and integration features, and a lack of training, documentation, and help systems. Unfortunately, early wins with innovators may create a false sense of security that the product is “ready for prime time” when it really isn’t. The early majority category is likely to demand that the same system accepted by the innovators must have good installation procedures and some integration features.

II. Early Adopters. Following the innovators come the early adopters. While they also like pushing the envelope, they are more conservative in their approach. They expect a more “complete” product, and are more demanding. They are likely to be more respected by senior management than the innovators because they are a bit more conservative. Early adopters are often key predictors for overall success. If *they* can be convinced of the innovations benefits it will likely become a long term success. If they can't, the product may ultimately fail. Closing the gap between innovators and early adopters is referred to as *Crossing The Chasm*, which is also the title of the influential marketing book written by Geoffrey Moore.

III. Early Majority. Close on the heels of the early adopters are the early majority. As you can expect, these individuals are considerably more conservative than the innovators and still more conservative that the early adopters. This is not necessarily bad, for this portion of the curve is often when the price/performance associated with the new product becomes most favorable. The earliest of the early majority often finds the greatest competitive advantage: most of the major kinks have been worked out, and they should be getting a fairly reliable product. From the perspective of product management, arguably most important aspect of the early majority is that they will require references to key early adopters to be assured that the product works. This includes, but is not limited to, such things as customer success stories and ROI (return on investment) analyses.

IV. Late Majority. The late majority are reluctant to adopt a new product, preferring tried and true mechanisms for dealing with problems. In fact, they may only adopt under significant economic pressure. Unfortunately, adopting this late in the process often results in the late majority receiving far less economic advantages than the early majority.

V. Laggards. Individuals who wait the longest to adopt an innovation are characterized as laggards. In general, they have the lowest social and economic status of all adopter categories. By adopting so late they derive little, if any, economic benefit from the innovation.

These broad categories represent a relationship between an individual and a given innovation. They are not universal labels, but convenient tools for understanding the behavior of key market segments. It is important to remember that innovations are adopted primarily because of their perceived effects on current problems. You might be an innovator and quickly adopt a new release of your favorite compiler in the middle of a project, especially if this compiler provides a much-needed bug fix or new feature. Alternatively, if the new release fails to address any specific problems it is far safer and cheaper to stick with the current compiler, which would categorize you as a member of the late majority.

The categories defined not only define adopter categories, but also define the maturation process associated with entire markets. The CDMA-based cell phone market is a mature market, while the 3G-based cell phone market is just emerging. Within these markets the adopter categories defined above exist. In subsequent sections of this book I will refer to both markets and adopter categories in the context of creating and sustaining winning solutions.

### 2.8.4 Whole Product

The concept of a "whole product" helps product managers create the full range of solutions required for success in the marketplace. Consider a common product, such as a cell phone. The *generic product* is the cell phone, a device that enables us to make phone calls in a convenient fashion. But, as anyone who has a cell phone knows, the cell phone must be augmented in a variety of ways before it becomes truly useful. For example, if you're unable to receive a phone call the cell phone provider almost always provides you with voice mail. This is the *expected product*, which is commonly defined as the smallest configuration of products and services necessary to minimally meet customer expectations.

Moving beyond these minimal expectations is where real value and customer loyalty can be created. Consider, for example, a cell phone that allowed you to pay online with a credit card, or one that had a visible indicator of how many minutes you had remaining on your monthly plan. This is an example of an *augmented product*, or a product that has been designed with a variety of "extras" to provide the maximum value.

These small ideas are just a few of the things I'd like in my cell phone plan. If you're a cell phone user, I bet you could think of others. If we enumerated these ideas, we'd have a definition of the true *potential product* that would exist for cell phone users: The full set of creative ideas that can exist to continue our product's growth and expand its market.

These four concepts collectively represent the "whole product". When applied to the world of technology, they reveal some surprising insights. While the generic might be the application, we expect that it will be relatively easy to install, that it will operate in a manner that is consistent with other applications on our chosen platform, and that it has basic online reference materials. Augmenting this product with an extensible API or including sophisticated operation and analysis capabilities may make us very happy. All of these capabilities are supported, directly or indirectly, through the choices made in the overall architecture of the system.

Marketing types often use the term *whole product* when the should be using the terms *expected product* or *augmented product.* Unfortunately, this can cause confusion, because the perception of the generic, expected, augmented, or potential product changes over time as technology and markets mature. To make the distinction clear, I use the term *target product* to define the specific product that is being created and offered to a given target market. In the early stages of the market, the target product may consist primarily of the generic product. As the market matures, the target product must evolve, because the majority of customers are unwilling to accept anything less than the expected or augmented product. The target product is related to a specific release, but the terms are not synonymous. The target product may include, for example, specific choices made regarding business models or support that do not directly relate to the system as it is delivered to your customer. Chapter [target product] is devoted to exploring the concept of the target product in greater detail.

### 2.8.5 TECHNICAL VS. MARKET SUPERIORITY

Product managers love the concept of “superiority”, because they can exploit superiority to create “unfair advantages” in a given market and thereby dominate the market. There are various kinds of superiority, not all of which translate into a winning product. Technical superiority is often easily duplicated, unless protected by patents, trade secrets, or other forms of intellectual property protection. Market superiority can be stronger, consisting of everything from a brand name to a distribution channel. A well-known example of market superiority is Microsoft’s OEM contracts, that have resulted in Microsoft operating systems being installed as the default operating system of the vast majority of personal computers.

### 2.8.6 POSITION AND POSITIONING

Your *position* is a complete sober, objective, and accurate assessment of how your customers currently categorize or perceive your product. It is their point of view, not yours. It is objective, which means if your customers think your product is hard to use, it is.

*Positioning* is a strategic, managed effort to create and defend a distinctive concept that your customer can care about and remember. Unlike “position”, which is about the present, “positioning” is about the future.

Marketing people care about position vs. positioning because the goal of positioning is to change the competitive playing field to one in which potential customers naturally think of your product / service before all others. Consider 7-Up. Common sense says it is a lemon-lime soft drink. Common sense might have tried to position 7-Up as the freshest lemon-lime soft drink, or the most sparkling, or the first, or the best, or the most lively, or the lemon-lime drink for the young at heart. Whatever. Uncommon sense decided to get 7-Up off that battlefield, and to hitch its star to the thriving cola category. “We’re not lemon-lime soda. We’re the Uncola.”

In technical markets positioning can matter more than position, because purchasing decisions are not controlled by purely rational thought processes. Instead, most purchasing decisions are driven first and foremost by emotions, and only later backed up with objective, rational “facts” that support the emotional decision. Consider your own behavior when making a purchase. Do you really go with Consumer Reports when they recommend a product or company you’ve never heard of? Or, like most of us, do you go with the product or company that has positioned itself as the leader?

It follows that positioning is rarely, if ever, feature-based. It is far better to own a meaningful and relevant benefit in the customer’s mind than own a temporary advantage in features. Positioning is long term, strategic, defensible, and ownable. Features are short term and can be duplicated. This also means that successful positioning can focus on only one concept – one that is narrow enough to be compelling and broad enough to allow for the future.

Once positioning has been set, you must continually create and recreate a position that moves you towards this goal. Without this, your positioning quickly becomes meaningless and loses all value in the mind of the customer.

### The Motivational Impact of Effective Positioning

I’ve found that there is a very strong correlation between a product development team that is having troubles and a product that has ineffective or nonexistent positioning. Positioning isn’t just about creating a future for your customers. Effective positioning is about creating a compelling future for *everyone* on the project, including the development team.

Whenever I am asked to work with an engineering team I inevitably end up working the product management team (and vice-versa!). My experience goes something like this. The development team can fairly quickly create a position, for they know the good and the bad of their product (although struggling teams spend a disproportionate amount of their time focusing on the bad). They can't tell me their positioning, because either they don't have one, or it isn't compelling.

Once we've created a compelling positioning statement I've found that it has a tremendous motivational impact on the development team. For perhaps the first time, they know they are going to be creating a future that their customers care about. A future that they can live in for quite some time, because it is strategic and defensible, and not based on a fleeting set of easily duplicated features.

It can be difficult to create a "motivational" positioning statement from scratch. If you find yourself stuck, try filling in the blanks in the "classic" positioning statement formula listed below to get yourself started.

| | |
|---|---|
| For | <target customer> |
| Who | <compelling reason to buy> |
| Our product is | <product category> |
| That | <key benefit> |
| Unlike | <main competitor> |
| Our product | <key differentiation> |

Applying this positioning formula to the fictitious benefits management system described earlier, we might create something like the following: *For hospitals with 250 – 1000 employees, who are finding benefits management costs to be increasing each year, our product self-service, web-based benefits management system, that, unlike traditional mainframe based benefits systems, provides easy, low-cost, and universal access to all benefits managements functions.*

### 2.8.7 BRAND

Your brand is the promise you make to a customer—It is why people care. *Everything* you and your company does is reflected in its brand. This includes partnerships, customer support, the nature and structure of the company's web site, the quality of the goods and services it provides to customers, the manner in which it manages profits – everything reflects on the brand. To illustrate the importance of brand, consider the differences that come to mind when I compare: Mercedes and

Hyundai, Coke vs. Pepsi, or Microsoft vs. Sun. This is among the reasons why brand management is a very important part of product and marketing managers job functions.

Brands are usually represented and communicated through a number of brand elements: terms, symbols, slogans, and names. Brand elements are often protected through various kinds of intellectual property, notably trademarks and copyrights. The impact of brand elements on creating a winning solution and their effects of software architecture are discussed in chapter [branding].

### 2.8.8 The Main Message

Your "main message" is a short (one or two phrase) statement that creatively captures the positioning. The main message should provide a continually useful bridge between your position and your positioning, reinforcing the positioning while making certain your customer "gets" what you're saying. The main message is important because it drives all creative marketing communication activities, from press releases to advertisements.

The main message should be customer-centric. Unfortunately, it is surprising the number of times this simple and pretty much obvious advice is not followed. Too often, messaging for technology products become shrouded in technical jargon, which often prevents the message from reaching its intended market. There are times when highly technical messaging does work, but like all good messaging this is based on what a customer "has ears to hear" to hear based on his/her needs. Great messages express an important benefit, telling the customer why they should care or how the product will solve their headache. Finally, any specific message that is delivered to a customer must be accurate and truly supported by the product.

## Chapter Summary

- Product management is the comprehensive set of activities required to create and sustain winning solutions.
- The processes associated with creating the first version of a software product begin with a concept proposal that justifies the product and ends with a launch plan. These processes are larger and more comprehensive than similar processes associated with product development. The processes associated with creating subsequent releases are lighter and build upon the materials created during the first release.

- Well run product development processes are characterized by a stringent "go/kill" decision at every stage of development. They are augmented and improved by successive freezing, change management protocols, and an idea recycle bin.
- The business plan is the central document that justifies the ongoing product development process.
- The "Four P's" of marketing are: Product (offering), Price and the business model, Place (distribution), and Promotion (advertising, marketing communications).
- The total available market is all of the customers who could possibly use your good or service.
- The total addressable market is that subset of the total available market that you can reach.
- A market segment is a group of customers that share specific characteristics, chief of which is that they must communicate with each other.
- The adoption of innovations, such as a new product or the new release of an existing product, follows the S-shaped curve of adoption. Key adopter categories within this curve are the innovators, the early adopters, the early majority, the late majority, and the laggards.
- The concept of the whole product comprises the generic product, the expected product, the augmented product, and the potential product.
- Your *position* is a complete sober, objective, and accurate assessment of how your customers currently categorize or perceive your product. *Positioning* is a strategic, managed effort to create and defend a distinctive concept that your customer can care about and remember. Your "main message" is a short (one or two phrase) statement that creatively captures the positioning.
- Your brand is the promise you make to a customer—It is why people care. *Everything* you and your company does is reflected in its brand.

## CHECK THIS

- ❑ We have defined some mechanism for capturing requirements.
- ❑ Product management represents the "voice of the customer".
- ❑ There is a way to kill a project or product in our company.
- ❑ We understand what our competitors are doing. We know what we need to do to win.
- ❑ We understand whom we are building our product for and to whom we are selling.

- ❑ We have a useful change management protocol

## TRY THIS

1. Who is the person that is playing the role of the product manager? When was the last time you ate lunch with this person?

2. Who is the person that is playing the role of the program manager? When was the last time you ate lunch with this person?

3. It might seem convenient to think that a given person is *always* an "innovator" or "member of the early majority". Fortunately, this is simply not true. Each of us has a unique response to an "innovation" (some new product or service). Sometimes we're innovators, sometimes a member of the late majority, and sometimes we don't adopt an innovation *at all* even if we're a member of the target market. Name at least one innovation in which you were an innovator; an early adopter; a member of the early majority; a member of the late majority; a laggard; and did not yet adopt at all. If you're having trouble with this question, ask yourself any or all of the following questions:
   Have I purchased a DVD player? If yes, when?
   Have I purchased a cell phone? If yes, when?
   When was the last time I downloaded a file from a Peer-To-Peer web site?

4. Get a copy of your current position, positioning, and messaging statements/documents from product management. Critically examine each. Do you think your position statement describes your product with integrity and accuracy? Why or why not? Is your positioning statement credible, true, unique, ownable, simple, relevant, memorable, sustainable? Does it paint a future you'd like to create? Is your message short and direct? Does it creatively link your position and your positioning

5. The essence of a brand is captured in one word or a short phrase: Mercedes means "prestige" while BMW means "professional driving". What is your brand? How does it compare to what you think of your product?

# 3. The Difference Between Marketecture and Tarchitecture

In chapter one I presented an overview of software architecture. Chapter two followed with a discussion of product management. This chapter returns to concept of architecture and clarifies how the marketing and technical aspects of the system work together to accomplish business objectives.

## 3.1 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 systems architecture. For software systems it encompasses such concepts as subsystems, interfaces, distribution of processing responsibilities among different processing elements, threading models, and so forth. As discussed in chapter one, in recent years several authors have documented distinct styles or patterns of tarchitecture. Some of these styles include client/server, pipeline, embedded systems, and blackboards, to name a few. Some of these descriptions include examples of where these systems are most appropriately applied.

Marketecture is the business perspective of the system's architecture. It embodies such things as 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, such descriptions of functionality as are commonly included in Marketing Requirements Documents (MRDs), Use Cases, and so forth. Many times the word "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 a "modular" approach to extending system functionality. The primary objective of this modular approach was that each of these modules could be separately priced and licensed to customers. The business model was that for each desired option, customers would 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 of a "module" 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 they could “install” and “uninstall” the module in a manner consistent with their goals and objectives. 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 upgrades 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 in a very easy manner; 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 the API as an "integrated" part of the client.

Net? Maintaining a difference between marketecture and tarchitecture gives both teams the flexibility to choose what they think is the best approach.

## 3.2 Early Forces in Solution Development

A variety of technical and market driven forces shape the creation of a winning solution. These range from the underlying technology to the competitive landscape to the mature of the target market. What makes these process so interesting is that these forces are always changing: Technology changes, the competitive landscape changes, a market matures and a new market emerges.

Three forces that are particularly influential in the early stages of development are the “ilities”, the problem domain, and the technology base (see Figure 3-1). Driving, and being driven by, these forces include the target customer, shown in the upper right hand corner, and the development organization, shown in the upper left. Product management is shown in the center of Figure 3-1 to emphasis their collaborative, leadership role in trying to resolve these forces.

The strength of the affinity that the customers 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.

[editor: can we get better representations of a customer, development team, and product mgr?]

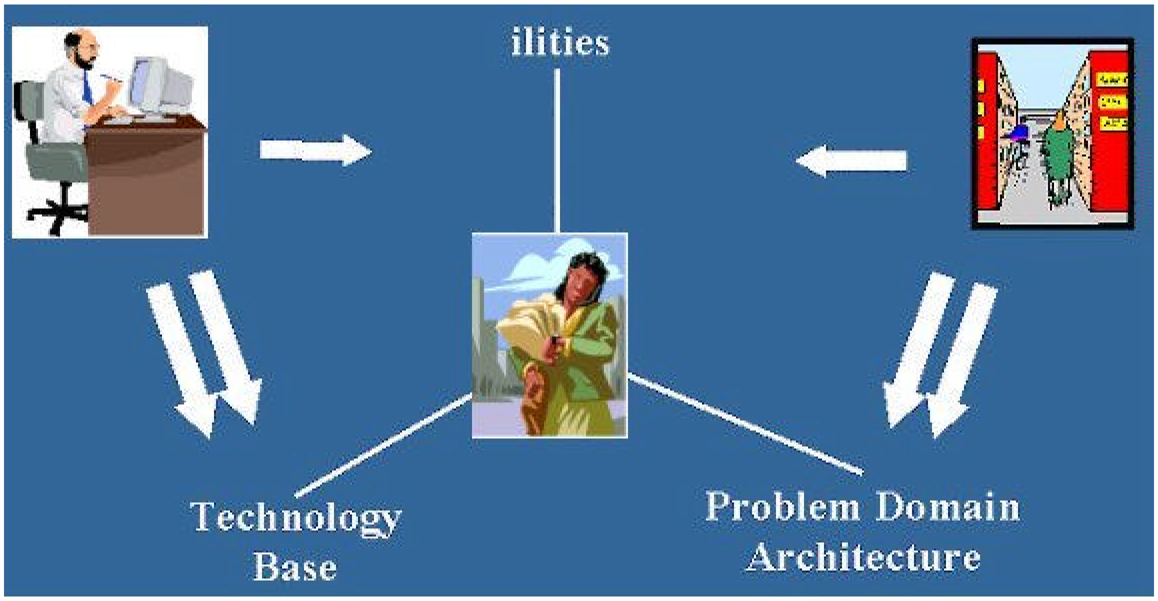


**Figure 3-1:Forces Shaping Software Architectures**

The *problem domain* is the central force guiding 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 uniquely different set of rules, nomenclature, procedures, workflows, and so forth. Included in my definition of the *problem domain* is the ecosystem in which the whole product exists, including customers, suppliers, competitors, regulatory entities, and so forth. 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.

The interplay between the marketect and the tarchitect in this process is quite interesting. Recall from chapter two that the marketect's primary job is in clarifying and prioritizing market needs while the tarchitect's primary job is to create a technical solution that will meet these needs. If the marketect is convinced that speed is of paramount importance, 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. To create a successful tarchitecture, the tarchitect must also bring their own domain experience to the tarchitectural design task.

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

- have 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 release after the initial release; or,
- have been a key member of a team that has made major architectural changes to an existing system and has experienced the effects of these changes through at least two full release cycles after these changes were introduced.

You're not an architect in your very first job. There is simply no substitute for sticking with a problem long enough to receive and process the feedback that is generated through customer use of your system.

**Sometimes, Learning "The Hard Way" Is The Only Way**

There are times when 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 supports multi-tier distribution of software. As I learned the capabilities of the system, I also learned about some of the lessons the team that created it learned over several years of working with major software publishers.

One of the most interesting lies in the management of serial numbers. As it turns out, almost every major software vendor has a unique approach to managing serial numbers through their sales channel. Some create serial numbers in real-time. Others create “blocks” of serial numbers that are distributed according to pre-determined algorithms to key channel participants. In this approach, serial numbers are used not only for identification of shipped software but for backend reporting and analysis.

Supporting all of these approaches requires close interaction between both marketects and tarchitects. In the case of this company, it is also vital to have professional services involved, since they are the group that makes everything “work” for a customer. It is 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 members of the team to learn the problem domain.

“ilities” are the various quality and product attributes ascribed to the architecture. As [Bass-Clements-Kazman] point out, these ilities fall within two broad dimensions: those discerned by observing the system at run-time, and those *not* observed by observing the system at run-time. 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 are critic ally 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 fixing the bug, 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 make certain their points of view are heard. Marketing needs to present arguments as to why a particular choice is "good enough" for the target customer. I’ve found it especially helpful to have customers participate in these forums. I vividly remember one customer demanding that we allowed them to ship a beta version of our software three months before the scheduled delivery date. The customer readily acknowledged that the beta version had many issues that needed to be resolved before shipping. However, the valu e of the beta was so compelling, and the customer need so great, that we eventually agreed to let the customer ship the beta subject to some strict terms and conditions regarding its use and a firm commitment to upgrade to 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 outright fail under certain usage scenarios. This data didn’t change their mind – they still shipped the beta – but it did enable the customer to equip their customer support organization with answers should these problems arise in the field.

As described in chapter one, engineering must make a variety of compromises in order to ship the product on time. Managing these compromises is difficult, for technical choices that compromise the product often have the most pronounced effect in the *next* version of the product, and not in the version that is about to ship. This is another reason to demand that your tarchitect have the experience of two or more full release cycles.

### Bug Severities, Priorities, and Normalization

One technique that I have found very effective in managing the “ility” perspective is to classify bugs by severity and priority. Severity refers to the impact of the bug on the customer. I’ve found that setting severity to a value ranging from 1 to 5 works well, where “1” means a crash with no work-around and a “5” is an enhancement request. Priority refers to the importance of fixing the problem. For consistency, I also prefer that values ranging from “1” to “5”. Here a “1” means a bug that must be fixed as quickly as possible – such as a bug that breaks the build or a bug that is required to satisfy an important customer. A “5” is a bug that 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 such a bug. Americans and Europeans are more tolerant, and are happy to give these kinds of bugs correspondingly low priorities. Japanese customers tend not to be as tolerant and give user interface bugs high priorities. Because of the subjective nature of bug priorities, setting priorities in a consistent manner across various members of the team can be difficult.

Fortunately, I learned a technique for improving prioritization consistency from one of the very best QA managers I know, James Bach. When code freeze occurred, James would organize bug review meetings that included all of the major stakeholders involved with the release. In this meeting James would review a sample of bugs (or all of them) to set initial priorities. Because representatives of all the major stakeholders were present, we could learn when and why support might prioritize a bug higher than product management 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 the collective perceptions of the team.

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

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

- The system MUST support 100 concurrent users;
- The system MUST retain all data created in a previous version throughout an upgrade;
- The system MUST only present an error dialog 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 bugs that for some reason don't seem to match the preset quality criteria.

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

Choices made within the technology base must be supportive of the tarchitecture as motivated by the problem domain. This can be challenging, as developers are people, replete with their own hopes, desires, preferences, and ambitions. Unfortunately, "resume driven design", in which developers choose a technology because they think it is cool, is a common malady that afflicts many would-be architects and is a major contributor to inappropriate architectures. In the same manner, marketects are also people, and "airplane magazine market research" becomes a poor substitute to the hard and often mundane work of performing the necessary market research and informed decision making that forms the foundation of winning solutions.

I have intentionally simplified my discussion of the early forces that shape the development of 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 tend to "lump" its effect into one associated with the problem domain, the ilities, or the underlying technology. This process is not intended to diminish, in any way, 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 your efforts to identify the *most important* forces affecting both your marketecture and your tarchitecture.

## 3.3 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 their lesser-skilled peers. Instead of listening to what customers want now (easy) they extrapolate multiple streams of data, including current requests, to envision what customers want 18-24 months in the future (hard). To them, the current release is ancient history, and they often refer to the requirements for the next release that are supported in the current architecture in the "past tense" 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 requirement motivates a 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 may also harm their ability to support existing customers. Envisioning the future on behalf of the customer, even when these customers can't articulate what they would like, is the key distinguishing feature of a world-class marketect.

Like their world-class 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 because the key tarchitects behind these systems envisioned a future and built for the same.

## 3.4 PROJECTING THE FUTURE

If the marketect and the tarchitect pursue different visions of a future the total system will fail. To minimize this risk I recommend creating a variety of simple diagrams to capture your understanding of how you want to create your future. I will refer to these diagrams as "maps", even though they are not "maps" of what currently exists, but "maps" of the future you're trying to create. These maps are reviewed briefly here and presented in greater detail in pattern format in the appendix.

The first map is the *Market Map*. It shows the target markets you've identified and the order in which you're going to create offers for these targets. (An offering is a bundle of one or more products or services). In order to make certain you can properly compete for these market segments it is helpful to create a *Feature/Benefits Map,* which shows the key features and their associated benefits required for each identified market segment. Some variant 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 timing of the market. 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 roadmaps. It shows the natural evolution of the tarchitecture in service of 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 appropriate consideration can be given to managing the changes needed to resolve the discontinuity. Alternative, emerging new technologies that hold promise for substantially improving the product and/or opening a new market are shown so that marketects can prepare for these futures.

Examples of such discontinuities abound. Consider an application originally designed for a single language. If this language becomes successful the marketect may include internationalization in their road map. The corresponding entry in the tarchitecture road map is often a major discontinuity, especially if the team is not experienced in building such applications. Another example occurs when the marketecture envisions new business models. Since it is doubtful that the original tarchitecture was planned with these alternative billing or licensing models, 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. The appendix shows how these maps work together to produce the best result.

## 3.5 HARNESSING FEEDBACK

Marketects typically use the following kinds of formal and informal, external and internal, feedback loops to ensure they are receiving 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 sales people for those features the sales team believes 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 data to tarchitects.

Drawing from different sources of data is a source for divergence between the tarchitecture and marketecture road maps described in the previous section.

Fortunately, the creation and ongoing maintenance (quarterly updates work well) of these roadmaps is the best way to prevent divergence and share data. Other techniques that can help include making raw data that informs these roadmaps available to both marketects and tarchitects. For example, tarchitects usually obtain primary market data via user conference or a focus group. Inviting the tarchitect to attend 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; the tarchitect is a key source for such articles. Note that my goal isn't to change the naturally different information seeking and processing tendencies that exist between marketects and tarchitects, but to make certain that the 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 as many of your developers to work directly with customers as you can. Several of my clients have been silicon valley startups. One was a company that created a marketplace for intellectual property licensing. For several years, this company ran a users conference to actively seek feedback from their customers on their current and proposed future product offerings. What made their user conference unique was that nearly the entire development staff was present to make presentations, conduct demos, and work with key customers. This direct involvement formed a key element of this company's ability to build products that their customers really wanted.

Of course, you might be thinking: "I'm not going to let my developers talk with customers. What if they say something that they shouldn't?" While this fear may be real, in that sometimes developers do say things that they shouldn't, 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 competitor's products.
- Don't say "That should be easy". It sets expectations too high and can kill any negotiation opportunities to have the customer pay for the modification.
- Don't say "That is too hard". It can prematurely stop conversation about what the customer really wants and ways to achieve this.
- Listen nonjudgementally. They are your customers. They're not stupid. They might be ignorant. They're not lazy. They might have priorities you don't know about. They're not your fan, nor your adversary.

## 3.6 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 development team can actually build it. The specific approach for achieving this varies, and is heavily influenced on 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 from simple, paper-and-pencil prototypes to more formally defined Marketing Requirements Documents. In response, development organizations can create models using such things as UML, entity-relationship models, data flow diagrams, and so forth. Communication and feedback between the teams can take place in regular meetings that formally review progress or daily meetings that track incremental improvements.

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

### Managing Cultural Differences in Software Development

In the course of my career I've managed several diverse groups of developers, including developers from Russia, Germany, India, Israel, China, Japan, Korea, Poland, Canada, and Mexico. At times, I've had the task of managing a world-wide team focused on the same deliverable.

There are, of course, several challenges in managing a world-wide development organization. Many challenges are logistical. For example, it is nearly impossible to schedule a simple meeting without inconveniencing some members of the team—08:00 US PST is 16:00 in Israel. Some development teams have it even harder – 12 hour time differences are common in the valley! Other examples exist in areas such as creating or adopting international standards for such things as 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 to overcome, 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 – but not the kind of culture we commonly associate with the word "culture". My own bias is heavily weighted towards those processes and practices promoted by the Agile Alliance (www.agilealliance.org). However, I've learned that 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, there are times when waterfall models are more appropriate, not because they will inherently produce a better result, but because the culture of the team *wants* to use a waterfall process. Marketects and tarchitects must both pay close attention to these potential cultural differences and choose approaches and processes that work for a given culture.

The marketect has similar objectives and responsibilities but for a very different set of audiences. The marketect must make certain the prospective client is clear on every aspect of how the system will impact their environment. If the system can be extended, such as in a browser with a plug-in architecture, the API must be made 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 white papers are common for any software that has a server component.

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 realized 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; sales people must be educated on the product; existing customers must begin preparing for the upgrade; and so forth. The tarchitect is partially responsible for making certain all of these happen timely and accurately.

## 3.7 WORKING IN UNISON

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

Let's first consider this question from the perspective of the marketect. Over the years I've found that marketects routinely under-estimate 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 the unexpected, and often delightful, capabilities of the system. One common example of this concerns 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. While we had never intended the API to be used in this manner, 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 10).

Another example concerns features that can be offered to customers because of choices made by the development team when implementing one of more key requirements. In one project I managed, the development team was charged with building a functional replacement of an existing server. The old architecture had a way of specifying pre- and post-processing hooks to server messages. Unfortunately, the old architecture's solution was difficult to use and was not widely adopted. The development team took the requirement to support pre- and post-processing hooks and implemented a very elegant solution that was very easy to use. Among other things, they generalized the pre- and post-processing hook message handlers so that an arbitrary number of hooks could be created and chained together. While the generalization was not a requirement, it created new functions that the marketect could tap.

A final set of examples illustrates marketing's ability to exploit development tools to customer gain. I've co-opted and subsequently productized developer created regression test suites for customers so that the operational health of the system can be assessed by the customer at their site.

I've converted log files originally created by developers to aide them in development into sources of data for performance analysis tools. I'm not advocating gold-plating, for this is a wasteful practice. But marketects who fail to understand the capabilities of the system from the perspective of its creators lose valuable opportunities to leverage 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 towards 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 roadmaps described earlier. Working away at these problems provides a clear outlet of the strategic energy expenditures of the tarchitect.

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

*Reach agreement on the set of project management principles and resultant practices driving the project.* There are a variety of principles that can drive any given project. From these principles project leaders select the specific techniques for managing the project. Differences on principles and resulting techniques can cause unnecessary friction between marketects and tarchitects. This friction will be reflected in the entire organization associated with the project.

To illustrate, much software is driven by a "good enough" approach to quality assurance. Other projects, especially those dealing with human safety, require much more rigor in their approach to quality assurance. These goals motivate marketects and tarchitects to utilize different principles. These principles motivate a set of project management practices that are different. Not better or worse. Different.

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

*Visibility to roadmaps and features.* All of the approaches I've described for capturing and planning for the future aren't much good if the data is hidden. Get this information into a forum where everyone can share it. Some teams accomplish this through the use of an intranet or a Lotus Notes database. Other teams are experimenting with Swiki's, Twiki's or CoWebs, with good results. My own experience with these tools has been mixed and is again heavily influenced by the culture of the team. 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 members of your project team will raise -- is a powerful tool to ensure marketects and tarchitects are working in unison.

## 3.8 CONTEXT DIAGRAMS AND TARGET PRODUCTS

Context diagrams are a great tool for creating a diagram that can help both that marketect and the tarchitect work in unison. A context diagram shows your system in context with the other systems or objects with which it interacts. A context diagram typically shows your system as a "single box" and other systems as boxes or stylized icons. Interactions between systems are 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. Instead, the context diagram is a "higher level" diagram that shows the *system in the context of its normal use.*

Context diagrams useful for a number of reasons.

- They help 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 help identify potential partnerships and market synergies. One of the most important applications of the whole product concept is in identifying partnerships that create a compelling augmented product and in defining a roadmap to a wonderful potential product.

- They help clarify your value proposition. Clearly understanding your value proposition is the foundation for creating a winning business model.
- They help 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 aspects of your application. They are a guide to the design of useful integration and extension approaches. See chapter eight for more details.
- They help you understand what kinds of 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 much sense to try and 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 the primary responsibility for creating and maintaining the context diagram. Of course, other members of the team can, and should, provide input to this diagram. The tarchitect, for example, can identify key standards; salespeople may suggest new entries based on how their customer uses their product; and so forth. Nonetheless, the marketect must be responsible for creating and maintaining the context diagram.

## 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 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 the problem.
- Use the patterns in the Appendix to create a strategic view of your product and its evolution.

- Winning solutions are much more likely when marketects and tarchitects work together.
- A context diagram is 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 have been placed in an easily accessible place for every member of the team.
- ❑ Deve lopers 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 the 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?

# 4. Business and License Model Symbiosis

Your *business model* is the manner in which you charge customers for your products or services – the way you make money. Every software business model is associated with a license model. A *license model* is the terms and conditions (or rights and restrictions) that you grant to a user and/or customer of your software as defined by your business model. To capture the

The symbiosis between these concepts is reflected in how easily we have created short hand terms that describe both of them in the same phrase. "An annual license for 10 concurrent users with the right to receive upgrades and bug fixes" is both a business model based on metered access to the software (by concurrent user) and a license model that defines some of the terms and conditions of its use—the duration, the right to use the software, and the right to receive upgrades and patches. Although business models and license models are symbiotically related, they are *not* the same thing.

Many software companies practice "Model-T" business model and licensing strategies. They create very few business models (often only one!) and expect each of their target markets to adapt to this model. Moreover, they define relatively rigid licensing models, failing to realize that within a business model market segments will pay to obtain certain kinds of rights (like the "right to upgrade") or remove certain kinds of restrictions. Rigidly defined business and licensing models may work for a new product in an emerging market, but mature markets need flexibility to capture the maximum revenue and market share from each segment. Just like we want choice in the color of our cars, so too do customers want choice in how they license their software.

Instead of forcing each target market to adopt a single business and licensing model, it is better to examine each target market to determine the combination that will provide you with the greatest market share and revenue. Doing this well requires that your product marketing organization understand the basics of each business and licensing model and how these work in a given target market. Given the plethora of possible models, this can be a challenge!

Considering all of the work involved in creating and supporting a business model, you may wonder why software is licensed and not sold (like a pen or a coffee mug). Perhaps the biggest reason is control. If someone sells you the software like a physical good then they lose control over what you can do with it. So, if someone sold you software, they couldn't prevent you from modifying it, reselling it, reverse engineering the software to determine how it works – and using that information to make competing products, or copying the software and loading it on multiple computers or making it available over a network. A physical object does not suffer from these problems, which is why all software today – even the software you're using to create your next program – is distributed via a license.

As one of the key embodiments of the marketecture, the business model and its licensing models may have a *tremendous* impact on your tarchitecture, imposing requirements throughout the system. More sobering is the fact that if your business model is the way you make money, creating the right one, licensing it the right way, and supporting it technically are the steps to a truly winning solution. Missing any one of these things, for any given target market, means you might lose it all.

This chapter explores business and license models and the effects that they can have on software architecture. I'll review how to make certain your business models are properly supported, discuss key license models often associated with business models, and discuss a few emerging techniques for making certain you're getting the most from your business model. When we're finished, you'll have the background you need to create effective business and licensing models.

**Business Model Nirvana**

A key objective of the marketect is to ensure that the company is profitably rewarded for the value it is providing to its customers. This often requires multiple business models to co-exist, because customers often perceive different kinds of value with different functions of a software system. In one application I worked on the main server was licensed to customers based on transaction fees while optional modules needed for large installations were annually licensed. This combination of business models matched how our customer received value from the system and allowed us to sell to a larger total market. Transaction fees have the virtue of scaling with size, in that small customers, who have few transactions, pay less, while larger customers, who have many transactions, pay more. Only larger customers, who need them, license the optional modules. Careful design of tarchitecture to support these two different business models was key to creating a winning solution.

## 4.1 COMMON SOFTWARE BUSINESS MODELS

The most common software related business models make money by:

- providing unfettered *access* to or *use* of the application for a defined period of time;
- charging a percentage of the *revenue obtained* or *costs saved* from using the application;
- charging for a *transaction*, a defined and measurable unit of work;
- *metering* access to or use of the application, or something the application processes;
- charging for the *hardware* the application runs on, and not the application itself;
- providing one or more *services* that are intimately related with application operation and/or use.

Traditional software publishers or independent software vendors (ISVs) that create their own software for commercial rely heavily on the first two business models. The other business models may be associated with an ISV or may be based on an open source licensing model. For example, while you can't "sell" Apache, you can certainly base your business model on installing, configuring, and operating Apache-based servers.

Before you read further, go back to the beginning of this list and replace the word "application" with "feature". Applications are collections of features, and each of these features can be offered under a different business and licensing model. For example, you could provide unfettered access to the base application for an annual license fee but charge a transaction fee every time the user invokes a special feature that produces a defined and measurable unit of work.

**Will that be an Application, Suite, Bundle, or Feature?**

Companies that offer more than one product often try to create synergies around their various product lines. One way of doing this is to associate a set of products into a "suite" or "studio". To be effective, the products that comprise the suite should work together and should be address some aspect of a common problem domain. Organizationally, you it is best if there is a separate marketect in charge of the suite, whose primary job is to work with the marketects of each product within the suite to make certain the suite is a good value for their mutual customers.

A bundle is a simpler concept than a suite, and is usually based on a set of products that are offered together as a unit. The products within the bundle may have no special relationship to each other, may not inter-operate, and may even be normally targeted at different market segments.

There may be a separate marketect in charge of a bundle. Ultimately, the marketect must answer the questions associated with creating a suite, a bundle, or marketing a feature or an application.

The more sophisticated business models are usually used with enterprise applications, although this is continually changing as the technology to enforce license models matures. Metering is a good example. Many enterprise applications meter by concurrent user. In consumer applications, you might wish to meter by the amount of time the user has been accessing your application, but this requires sophisticated technologies, including a reliable way to capture usage data. As the technologies required to do this mature, you will find each of these business models offered wherever they can allow the marketect to increase their business!

Let's consider each of these business models in greater detail, keeping in mind the following points:

- A complex system may have multiple business models. The "base" system business model might be based on transaction fees while additional "optional" modules might be annually licensed. The central business model should be congruent with the value received from the customer by the generic and expected product. The augmented product may benefit from different licensing models.
- Business models are often coupled with one or more field of use license restrictions. A *field of use* restriction constrains where, when, and how the software the can be used. Common field of use restrictions include constraining the software to be used on a single computer, a single site (called a *site license*), or to one or more geographies (which export restrictions may require).

  Field of use restrictions work in conjunction with the core business model to create more total revenue for a company. Consider an annual license that is restricted to a single computer. If the licensor wishes to use the software on another computer, they must obtain an additional license, thereby driving more revenue. Field of use restrictions are another aspect of the licensing model, much like the "right to upgrade" or the "right to receive technical support". For consistency in contract language and in the organizing your approach to enforcing your licensing model, it may help to convert a field of use restriction to a right – you have the right to use this software on a designated computer or you have the right to use this software on any computer in your company.

- All of the business models have some concept of time associated with their use. This period of time is defined in the license model. Pre-defined periods of time are common because our business world is geared towards these periods of time (months, quarters, years).
- The licensing model may also define how and when a customer must make payments as a way of extending a given business model to a new market. For example, a service based on a monthly payment originally created for the Small to Medium Enterprises (SME) market and requiring a purchase order might have to be augmented with the ability to accept credit cards to reach the Small Office/Home Office (SOHO) market.

### 4.1.1 Time-Based Access or Usage

In this model, the licensee can use the software for a well-defined period of time, such as when you "purchase" an operating system. What is really happening is that the operating system publisher, or their agent, has granted you a right to use the operating system, typically on one computer, for a defined period of time. Other rights and restrictions will be defined in the license model, but the core business model is based on accessing or using the software (you *pay* to license the software so that you can *use* it).

Common periods of time for time-based access or usage include:

- *Perpetual.* The licensor is granted a license to use the software "in perpetuity" – forever! Upgrades or updates are not usually included and instead are separately priced. Bug fixes or patches may be included as part of the original fee or you may be required to pay a maintenance fee (these fees typically range from 15% to 30% of the initial license fee). Be very careful of perpetual licenses, as they often increase total support and maintenance costs (you may have to support *every* version—*forever!*).

  Despite the potential drawbacks of perpetual licenses, they are surprisingly common and often required in many markets. Consumer software, common productivity tools, such as word processors or presentation software, operating systems, as well as many enterprise applications, are based on a perpetual license. Perpetual licenses may be required if you’re providing a system or technology that may be embedded within other systems. Examples of this range from the run-time libraries provided by programming language vendors (e.g., the C runtime library) to systems that may become components of hardware devices or information appliances.

- *Annual.* The software can be accessed or used for one year from the effective date of the license. Annual licenses often include software updates or patches. They may or may not include other services, such as technical support or consulting fees. Annual licenses are usually renewable. A renewal may be automatic, and it may be priced differently than the original annual license.

  The difference between a perpetual license with an annual maintenance fee and an annual license is subtle, but important. In the case of a perpetual license, the licensee (user or customer) has obtained a license to the software in perpetuity. The maintenance fees are added on as a way of enticing the licensee to pay additional money for additional services, such as receiving bug fixes or new releases. In the case of an annual license, the licensee can only use the software if they have paid the annual license. If they haven’t paid, and they continue to use the software, they are in breach of the license. You will ultimately have to decide how you want to enforce license terms and conditions, a topic explored in greater detail later in this chapter. Annual licenses are common in enterprise class software systems.

Although perpetual and annual licenses are the most common pre-defined periods of time, there is nothing that prevents you from defining other periods of time (e.g., 9 months) or a collection of times when the software can be accessed (e.g., Monday through Friday from 8AM to 5PM). The hard part is defining a period of time that works best for your target market and making certain you can appropriately enforce the business model and license terms (which market segment would find a 9 month time period or only accessing software Monday through Friday from 8AM to 5PM sensible?).

Here are a few additional time-based usage models.

- *Rental.* A rental is a time-based model in which the amount of time allowed for the use is set when the license is purchased. In reality, a rental is just an annual license with a different term (or an annual license is a rental of one year). Rentals are becoming increasingly popular in certain industries, including the software test automation and the Electronic Design Automation (EDA) industries. As license management technology matures, I predict that rentals will reach all market segments, and that we will be able to rent just about every application available.

The business motivations for rentals are compelling: A rental allows you to reach a new market. Suppose, for example, that you market a high-end, professional video editing system for $50,000 for a single user, annual license. One way to reach the home user market is to create a simpler version of this system, with less features. Another way is to offer the system as a rental (say, $100/hr). For the user who has learned the system at work but wants to use it at home to edit a set of videos of the recent family reunion, the rental price might be a real bargain – they already know how to use the application, so why go with anything else?

Key issues that must be resolved with a rental model include specifying when the rental period begins (when you purchase the license, install the software, start using the software); determining how the software should respond when the rental is finished (will the program stop working entirely or will you give the user some kind of "grace" period so that they can save their work?); and pricing the rental (if you can obtain a perpetual license for a graphics design package for $295 what is the cost for a one hour rental? a one day rental?).

- *Subscriptions.* At the other end of the spectrum from a rental is a subscription, in which customers pay a periodic fee to access the software. A subscription is like a rental or an annual license—all that differs are the terms and rights defined in the licensing model. For example, a rental may not include the right to receive upgrades or patches to the rented software. Most subscriptions, and annual licenses, on the other hand, do. A rental and a subscription may include technical support, while in many cases you must purchase an additional support contract (which can define varying kinds or levels of support) with an annual license. Because subscriptions are often associated with some backend service, the license models that define the subscription are relatively easy to enforce. Simply cut off access to the backend service!

- *Pay after use.* An interesting question arises in any time-based usage model: What happens when the user accesses the application after the approved period of time? How you answer this question can make or break your relationship with your customer – and your bank account.

At one end of the spectrum is absolute enforcement of a business model with no grace period: when the term is finished, the software becomes inoperable. This very severe choice is simply inappropriate for most target markets.

An emerging business model is to charge the customer *after* they have used the software by keeping track of how long they have used the software. While this model places a host of complex demands on the tarchitecture (see below), it does help ensure that the company is receiving every dollar it can from the use of it's software.

As an analogy, consider a car rental. In a car rental, you're typically charged for each day that you rent the car, with strict definitions as to what constitutes a "day". If you go beyond the original contract, the car company continues to charge you (they have your credit card on file, and the rate that you'll be charged is pre-defined in the contract). The pay after use business model will become more popular as the relationship between business models, licensing models, and billing systems matures.

A special category of software that uses time-based access or usage software are the vast number of applications designed to work on a single personal computer, workstation, or handheld device. As a group, these are often referred to as *per-user* licenses, and are often applications that retail for less than $500. Per user licenses work great when the publisher is licensing one copy to one user, but not so well when a publisher is trying to sell 500 or 5,000 licenses to an enterprise.

To address this issue, most major software publishers have created *volume licensing* programs to more effectively reach enterprises (businesses, charitable organizations, academic institutions) that license more than one copy of the same software program. Volume licensing programs are not fundamentally new business models. Instead, they are sophisticated pricing models based on an access or usage based business model. They offer enterprises a way to acquire multiple licenses of a given program, usually at a discount. The more copies you license, the greater the discount.

Volume licensing programs are offered in two basic varieties. In a *transactional volume licensing* program, each product is associated with a certain number of *points*. A high-end, expensive product may be worth 5 points, while a low-end, inexpensive product may be worth one point. As an enterprise creates a purchase order (500 licenses to this software, 230 licenses to this other software) a point total is dynamically computed. Applicable discounts are calculated based on the total points as specified by the transaction.

In a *contractual volume licensing* program, the enterprise estimates the number of licenses that they will need for a projected period of time and commits to acquiring *at least* these licenses by the time specified in the contract. The commitment to purchase a minimum number of licenses enables the enterprise to obtain these licenses at a discount. Bonus discounts may be allowed if additional licenses are acquired, and penalties may be assessed if the customer does not license enough.

Both transactional and contractual licensing programs are highly regulated by the companies that offer them. They should be considered whenever you're likely to be selling multiple "copies" of the same software to an enterprise. Thus, they are more appropriate for applications that can be used in the enterprise context (games are probably not a good choice for a volume licensing program). You may not want to offer a program designed to manage personal finances as part of a volume licensing program, largely because the target market will be a single user on a single computer. That said, marketects should be aware of the benefits that a volume licensing program can provide to their customers and their product lines.

Another category in which time-based access or usage is the dominant business model is in the OEM or royalty market. The business model is access to the software. The pricing model is whatever makes sense (in the case of an OEM) or a fairly defined royalty.

### 4.1.2 TRANSACTION

A transaction is a defined and measurable unit of work. Business models based on transactions associate a specific fee with each transaction or a block of transactions. The definition of a transaction can be surprisingly simple or maddeningly complex. For example, a transaction can be defined as simply "executing" the software. This is common in a business model known as "Try and Die", in which you can execute the software a pre-defined number of times – say, five times – before it becomes inoperable. I've also worked on systems in which a transaction was distributed among multiple servers; the business model was based on charging the customer whose "root" server initiated the transaction.

Fees may be calculated in many different ways. I've worked on systems based on "flat" fees (a fixed cost per transaction); "percentage" fees (a percentage of some other calculated or defined amount); “sliding” fees, in which the cost per transaction decreases as certain volumes are realized; or “processing” fees, in which the work performed by to perform the transaction is measured and the customer is billed accordingly (e.g., a “simple” transaction that could be computed with few resources cost less than a “complex” transaction that required many resources).

There is nothing about a transaction-based model that requires the transactions to be the same “size”, “duration”, or “amount”. Indeed, the differences are the foundation of many business models, and you can use these differences to create different pricing structures. For example, in the telecommunications industry, a phone call is a transaction. The duration of the phone call determines the price.

Transaction-based business models are almost exclusively found within enterprise software. Creative marketects know how to define a transaction and construct a transaction fee that works best for their target market. It is imperative that the tarchitect understands both the legal and the business model definition of a transaction.

**Sounds Great, But What Do I Charge?**

A business model defines how you will charge a customer for your products and/or services – but not how much. A *pricing model* defines how much you will charge. Transaction fees, for example, can be just a few cents to multiple millions of dollars – depending on the nature of the transaction! Your business model may be based on access to the software; the associated pricing model could be based on a one time payment based on the specific modules or features licensed (a “Chinese menu” approach to pricing) or it could be a set fee based on the annual revenue of the enterprise.

Pricing a product or service is one of the hardest activities that a marketect will undertake. Charge too much and you leave yourself vulnerable to competition, create lower revenue (too few customers will afford your product), and slow growth. Charge too little and you’ll leave money on the table. While a detailed discussion of pricing is beyond the scope of this book, here are some principles that have worked well for me.

- Price should reflect value. If your customer is receiving hundreds of thousands of dollars of quantifiable benefits from your product, you should receive tens of thousands of dollars. From the perspective of your customer, price isn't affected by what it COSTS. Instead, price should be set by what it's WORTH to the customer.

- Price should reflect effort. If your application requires a total development team of 120 people, including developers, QA, technical publications, support, product management, marketing, and sales, then you need to charge enough to support them. From the perspective of your employer, if you’re not going to somehow charge enough to be profitable, they’ll shut down the product. And they should. Either it isn’t providing enough value or the value it provides costs too much to create.
- Price should support your positioning. If you’re positioning yourself as the “premium” offering, your price will be high. If you’re positioning yourself as the “low cost” alternative, your price will be low.
- Pricing must reflect the competitive landscape.
- Pricing models should not create confusion among your customers. I realize that this can be difficult to follow, especially when you’re creating a system with a lot of options. Find ways to reduce the complexity of the “Chinese menu” approach by offering bundles. Or just plain simplify your offerings.
- Pricing models should reflect market maturity. If you’re in an emerging market you may need to try several different pricing models before you choose the one that works best. Be careful with your experiments, because later customers will ask earlier customers about how much they’ve paid. Note that you can only do this if you have a method for tracking and evaluating the performance of your various models.  This can be a tarchitectural issue, in that you may need your software to report certain information to back office systems when it is installed or used.
- It is usually harder to raise prices than to lower them for the same product. If you start to low and you need to raise prices, you’re going to have to find a way to split and/or modularize your offering.
- Pricing models should reflect your target market. Selling essentially the same solution at different price points to different markets (e.g., the “student” or “Small Office” version) often makes good sense.

### 4.1.3 METERING

Metering is a business model based on constraining or consuming a well-defined resource or something that the application processes. A constraint model limits access to the system to a specific set of pre-defined resources. A consumptive model creates a “pool” of possible resources that are consumed. The consumption can be based on the concept of concurrency, as when two or more resources simultaneously access or use the system, or an absolute value that is consumed as the application is used. When all of the available resources are temporarily or permanently consumed, the software becomes inoperable. Many successful business models blend these concepts based on the target market. Here are some of the ways this is done.

- *Concurrent resource management.* This business model is based on metering the number of resources concurrently simultaneously accessing the system. The most common resource is either a “user” or a “session”. The business model is usually designed to constrain the resource (“a license for up to 10 concurrent users”). Both “user” and “session” must be defined, because in many systems a single user can have multiple sessions. The specific definition of a resource almost always has tarchitectural implications; managing concurrent users is quite different from managing concurrent sessions, and both are different from managing concurrent threads or processes.

  Like transaction fees, concurrent resource business models have a variety of pricing schemes. You may pay less for more resources, and you may pay a different amount for a different resource. Concurrent resource models are almost exclusively the domain of enterprise software.

- *Identified resource management.* In this model, specific resources are identified to the application and are allowed to access the system when they have been properly authenticated. The constraint is the defined resources. The most common resource is a *named user*, a specifically identified user that is allowed to access the application. Identified resource business models are often combined with concurrent (consumptive) resource business models for performance or business reasons. Thus, you may create a business model that is based on “any 10 out of 35 named users concurrently accessing the system” or “any 3 out of 5 plugins concurrently used to extend the application".

The concept of a "user" as the concurrent or identified resource is so prevalent that marketects should be alert to the ways in which they can create special ways organize their business model around users. One common way that this is done is to classify users into groups, or types, and separately define the functions and/or applications that can be accessed by each group.

The idea is analogous to how car manufacturers bundle optional / add-on features in a car and sell it as a complete package (the "Utility" model vs. the "Sport" model). As a result, it is common in concurrent or named user business models to find defined types of users ("bronze", "silver", or "gold", OR "standard" vs. "professional") with specifically defined sets of functionality associated with each user (e.g., a "concurrent gold user can access these features…").

The administrative burden of this approach can be overwhelming for corporate IT departments. To ease this burden, try to leverage existing directory or user management infrastructures, such as any AAA (Access Authentication Authorization) or LDAP (Lightweight Directory Access Protocol) servers that may be installed. These systems are designed to capture and manage these data on behalf of corporate IT.

- *Consumptive resource management.* In this model, you create a specified "amount" of a resource and "consume" the "amount" once when the application is invoked or continually while it is running. Unlike a concurrent model, in which the degree or amount of consumption varies based on the specific set of resources that are simultaneously accessing the system, a purely consumptive model expends resources that are not returned.

  To illustrate this model, consider time as a consumptive resource. In this approach, you define a period of time as a consumptive resource (e.g., "100 hours" or "30 days") and provide the user with a license with this amount of "time". As the software is used, it keeps track of the time, "consuming" the designated value from the licenses. When all of the allotted time has been used the software becomes inoperable. Key issues that must be resolved in this approach include the definition of "time" (actual CPU time or system elapsed time or other), the manner in which the software will track resource consumption (locally, remotely, or distributed), and the granularity of the time-based license (milliseconds, seconds, days, weeks, months, and so forth).

More generally, it is possible to define an abstract resource and base your business model on metering this resource. Suppose you have created an application for video publishing with two killer features: the ability to automatically correct background noise and the ability to automatically correct background lighting. You could define that anytime the user invokes the background noise correction feature they are consuming “one computing unit” while anytime they invoke the background lighting correction feature they are consuming “three computing units”. You could then provide a license for “20 computing units” that the user could “spend” as they deem appropriate.

Consumptive resource models can form the foundation for creating subscription based service models – each billing period (say, monthly), for a set fee, you get a pre-defined number of resources. When the resources are consumed, you can purchase more or stop using the application. Whatever resources are not consumed are either carried over to the next billing period (possibly with a maximum limit, like vacation days at many companies) or lost forever. This is similar to the access-based subscriptions, but differs in that you are metering and consuming a resource. The difference may be fine-grained, but it is worth it to explore the potential benefits of each possible model, because you may be able to access a new market or increase your market share in a given market with the right model. In addition, both models make unique demands on your tarchitecture, so the tarchitect will *have* to know the difference.

Consumptive models have another critical requirement that is often overlooked—reporting and replenishment. It must be extremely easy for a user/administrator to predict how much an operation will “cost” BEFORE they decide to “spend”, the rate at which “spending” is occurring, and alert the user when the rate of “spending” will exceed the allotment for the month, or a resource budget is nearing depletion. Because customers will often overspend, it should be painless for an overspent customer to buy more. They will not blame you if they run out of a critical resource on Friday afternoon at 6PM Eastern time, just before a critical “big push weekend” – especially if you warned them yesterday it would happen. But they will *never, ever* forgive you if they can't buy more till Monday at 9AM Pacific time.

### 4.1.4 HARDWARE

Hardware based business models associate the amount charged for the software with some element of hardware. In some cases the software is “free”, but is so intimately tied to the hardware that the hardware is effectively non-functional without the software. A more traditional approach, and one that has been common in business applications, is to associate the business model with the number of CPU's installed in the system ("Per-CPU licensing"). The motivation for Per-CPU business models, like all business models, is money.

Consider an application that has been licensed to run on a single machine. If the performance of the machine can be substantially improved simply by adding additional processors, the licensor (software publisher) stands to lose money, because the licensee will just add processors without paying any additional license fees! If this same application is licensed on a Per-CPU basis, then adding more processors may improve performance, but the licensor will still get more money.

Hardware based business models can be based on any aspect of the hardware that materially affects the performance of the system and can be enforced as required to meet the needs of your business. Per-CPU or "per expansion card" business models are the most common, but you can base the business model on memory, disk storage systems (e.g., redundantly mirrored disk drives might be charged twice), or any other aspect of hardware. That said, I wouldn't recommend basing the business model on the number of connected keyboards – but you could.

### 4.1.5 SERVICES

Service-based business models are focused on making money from one or more services, not the software that provides access to these services. My favorite example of a service-based business model is America Online. AOL doesn't charge for the software – they charge a monthly subscription fee that provides access to a wide range of services, including email, chat, and content aggregation.

Service-based business models are often used with open source licensing initiatives. Examples of services here include providing services to assist in the installation, configuration, and operations of an application or technology built on open source software or based on an open source license. Other services that can be provided include creating education programs (think "O'Reilly) as well as custom development or integration services.

Creating a servic e-based business model through the use of open source software (OSS) licensing is a creative approach to many licensing and business model questions. However, as of the writing of this book there have been no provably sustainable, long-term successful business models created solely around the creation of open source software. This is not an indictment of OSS! I'm aware that most of the Internet runs on OSS, and many companies have been started with very promising service based business models that are related to OSS. Instead, it is merely an acknowledgement that the market is immature, and that sustainable business models created around OSS have yet to be proven. As a result, any marketect that is approaching their business through the use of OSS should proceed with caution.

### 4.1.6 Revenue Obtained / Costs Saved

Another business model that is common in enterprise software is a percentage of revenue obtained or costs saved from using the application. Suppose you've created a new CRM (Customer Relationship Management) system that can increase sales to existing customers by an average of 15%. You may consider charging 10% of the incremental revenue, provided the total dollar amount is large enough to justify your costs.

Alternatively, let's say that you've created a new kind of inventory tracking and warehouse management system targeted towards small manufacturing companies ($5M - $50M in annual revenue). Your data indicates that your software will save these companies anywhere from $50K to $1M. A viable business model may be charging 15% of the savings, again, provided the total dollar amount was sufficiently large.

In choosing these models you have to make certain that you have a rock-solid analysis that clearly identifies the additional revenues or savings. If you don't, no degree of technical skill in the development team will help the architecture become successful. The fundamental problem is that the basis of these models are extremely subjective and easily manipulated. You might think your new CRM software generated Fred's Fish Fry $100,000 more business, but Fred thinks it's the spiffy new Marketing campaign that Fred, Jr. created. Result? You're not going to get paid the amount you think you've earned.

Enterprises also appear to be more resistant to models based on cost savings. From my own experience, I once tried to create a pricing model based on cost savings. Even though we had a very solid ROI, the model didn't work and we switched from costs savings to transaction fees. Percentage of revenue obtained or of costs saved are also unpopular because they make *customers* do work to track how much they should pay. It is usually must easier to determine how much a customer should pay in the other business models.

## 4.2 RIGHTS ASSOCIATED WITH BUSINESS MODELS

Marketects choose business models to maximize the benefits to their business. Associated with each of the business models described above are a set of rights and restrictions – "things you *can* do" or "things you *get*" and "things you *cannot* do" or "things you do *not* get". I've covered a few of these already, such as the "right to use" and the "right to upgrade" or a restriction like "you can only run this software on one computer".

Your business model should try and distinguish as many rights and restrictions as possible, because each "right" is a way of capturing and/or extracting value from your customer and each restriction is a way of protecting your interests. Many business models, for convenience, competitive advantage, or common practice, often create a "standard" licensing model that blends a variety of rights and restrictions into a single package – but remember that separating them can create greater value.

For example, an annual license to use the software (the business model is access) commonly includes the "right to use" and the "right to receive upgrades" but not the "right to receive technical support". A subscription to a software-based services model (such as America Online; the business model is service or metering, depending on customer choices) may include all of these rights. This section reviews some of the rights that are commonly associated with various business models.

Table 4-1 outlines some the rights and restrictions associated with the previously described business models. Columns are a specific right or restriction. Rows are the business model. A "✓" in a cell means that this right is commonly allowed by the business model. A "number" in the cell means that the right *may* be allowed by the business model and will be discussed in greater detail. I've added a column that addresses whether or not the fees paid by the licensor are one time or periodic. The timing of fees can affect the choice of a business model.

The focus of Table 4-1 is the subset of license model terms that are most closely correlated with various business models. It does not attempt to capture every possible legal issue that can be defined in a license, such as exclusivity, warranties, indemnifications, intellectual property rights, confidentiality, or any number of other issues that a savvy lawyer can put into an agreement. Most of these issues are not a function of the business model, but instead a function of larger corporate policy that governs every license model, regardless of the business model.

| **Business Model** | Right to upgrade (to latest version) | Right to receive bug fixes/patches | Right to return | Right to move to a different machine | Right to embed | Right to modify | Right to resell | Support options (e.g., phone, web, email) | Pre-defined installation/customization support | One time or periodic fee |
|---|---|---|---|---|---|---|---|---|---|---|
| Time-based Access | | | | 1 | 1 | 1 | 1 | 1 | 1 | |
| Perpetual license | | | | | | | | | | one time |
| Annual license | ✓ | ✓ | | | | | | | | periodic |
| Rental | 1 | 1 | | | | | | | | one time |
| Subscription | 1 | 1 | | | | | | | | periodic |
| Pay after use | | | | | | | | | | one time |
| Transaction | 2 | 2 | | 1 | 1 | 1 | 1 | 1 | 1 | periodic |
| Metering | | | | 1 | 1 | 1 | 1 | 1 | 1 | |
| Concurrent resource | ✓ | ✓ | | | | | | | | one time |
| Identified resource | ✓ | ✓ | | | | | | | | one time |
| Consumptive resource | 1 | 1 | | | | | | | | one time |
| Hardware | 3 | 3 | | 1 | 1 | 1 | 1 | 1 | 1 | one time |
| Service | ✓ | ✓ | | 1 | 1 | 1 | 1 | 1 | 1 | periodic |

**Table 4-1**

1. Associate this right with the business model if providing these rights will provide you with a competitive advantage, enable you to reduce technical or product support costs, or create a stronger tie between you and your customer. Avoid associa ting this right with your business model if your customers don't really care about this right or if doing so may cause them more bother or pain that it is worth. For example, in the anti-virus market, you pretty much have to provide upgrades and bug fixes. In enterprise class software, upgrades may not be as meaningful because of the large time delays that may exist between release cycles.
2. I'm usually surprised to find that transaction based business models don't automatically include these rights. In these models, your goal is to drive transactions. Improvements to your software,

and especially those improvements that can drive transactions, should always be made available to your customers.

3. Associate these rights if the benefits of point (1) above apply and your customers have a relatively easy means of applying the upgrade and/or patches. Note that in some hardware-based business models you *don't* want your customers to upgrade to new software. Instead, you want them to purchase entirely new hardware, which is another reason to withhold these rights.

## 4.3 TARCHITECTURAL SUPPORT FOR THE BUSINESS MODEL

This section addresses key tarchitectural issues associated with specific business models. If your total offering is based on a combination of business models, check for the key interactions that may exist between the various elements. In addition, review these factors every time the business model changes, because changes to the business model may invalidate prior assumptions and/or choices and motivate one more changes to the tarchitecture.

### 4.3.1 GENERAL ISSUES

The following general issues are present for every business model.

- *Capturing the necessary data*. Assess your business model to ensure that your system is capturing all of the data required to make the business model "work". Consider two primary categories of data capture:
  - *Direct*: The system captures and manages all data necessary to support the business model. It is "self-contained".
  - *Indirect*: The system must be integrated with one or more other systems to create a complete picture of the necessary data.

  Some business models may not have to deal with this in their software, but instead shift the responsibility for handling this data capture to other systems. To illustrate, a transaction or metered business model will either capture all of the necessary data or work with other systems to capture the full set of data needed for the application. A service based business model often has to be integrated with other systems to capture the full set of data needed to create a viable business model.

- *Reporting/Remittance requirements.* Somewhere along the line your accounting department is going to require that information be submitted for billing purposes. Your job is a lot easier if you can define format, security, and audibility requirements of these reports.

- *Business model enforcement.* What happens when the license to use the software is violated? Does it stop working? Should an entry be made in a log file? Is an email sent to a key person? Should a new billing and licensing cycle be automatically initiated?

- *Focusing "ility" efforts.* In the ideal world, the "ility" efforts (reliability, stability, scalability, supportability, usability, and so forth) of the tarchitect are congruent with the key objectives of the business model. Suppose your business model is based on processing various transactions, such as check clearing and/or check processing in the financial services arena or health care claims processing for large insurance carriers. In these cases, reliably and accurately computing a lot of transactions fast is critical to your success. Performance *matters*, and *faster really is better*. Who cares if the product a bit hard to install or upgrade as long as it the fastest possible solution! If your tarchitect and the development organization charged with building the system also cares about performance, you're doing great. If not, you may have some problems.

  Of course, performance is not the primary criteria of every business model. In a service-based business model, great performance with lousy customer support may not be good enough to succeed. Instead of focusing on performance, your development team may need to focus on making the product easy to install and easy to upgrade. A key goal in a service-based business model is to reduce and/or eliminate phone calls to technical support, so it is best to find a tarchitect and a development team who care about these issues. Tradeoffs are determined by how the software will typically be used – applications to be used infrequently would have a greater focus on ease of learning, while those to be part of a daily routine would benefit by being faster and easier to use.

- *Increase revenues or decrease costs*. Once you know your business model it is easy to focus tarchitectural efforts on those activities that increase revenues. Don't forget the *costs* your tarchitecture imposes on your customer. Anytime you can reduce the costs imposed by your solution, you have a potential for obtaining more revenue. Common areas where you can reduce costs include making installation and/or upgrade easier, or improving performance and/or backwards compatibility, so that your users won't have to purchase expensive new hardware.

- *Copy protection/anti-piracy protection*. All software is subject to illegal copying and distribution. Certain kinds of software, such as the software that comes inside a video game cartridge or your cell phone, is more resistant to piracy because of the hardware. But even hardware is not a strong deterrent—just check out the number of places where you can find replacement software that can boost the performance of your car! Most software, however, is trivially copied, as proven by the many online services that provide easy access to pirated software. Depending on the amount of money you may lose from piracy (some companies estimate several millions of dollars in losses) you should consider implementing a copy protection scheme, as described further below.

- *Verifying business model and license model parameters*. Many of the business and licensing models involve setting one or more parameters. An annual license has a definite end date. A concurrent user license has a specific number of concurrent users, possibly further divided by type. A consumptive, time-based license has the amount of time available for use. Whenever you're setting a value that is important to your business model, you're going to have to understand when, how, and where this value gets set, who can change it, and how its value can be verified. These are not trivial matters, a topic discussed later in this chapter under enforcing business and licensing models.

### 4.3.2 TIME-BASED ACCESS OR USAGE

Time based access or usage business models make few special demands on the tarchitecture, unless you're going to strictly enforce the business model. If this is the case, you're going to need a way of being able to disable the software when the time allotted for use has expired. Note that many subscriptions don't actually stop a program from working if you fail to remain current in your payments – you just don't get updates (which has the effect of converting a subscription to a perpetual license).

### 4.3.3 Transaction

Consider the following when dealing with a transaction based business model.

- *Define the transaction.* The definition of a transaction, or the unit of work that is the foundation of the business model, must be clearly and unambiguously defined. Once you've defined the transaction, make certain that the tarchitecture can support it and that it is clearly stated in the software license. This is not easy, but it is essential. As you define the transaction, consider the role that each component is playing with respect to the transaction. In distributed transaction systems, the manner in which element participates in the transaction must be defined, including which participants are the basis of the business model.

- *Define the relationship of the transaction to the business model.* I've heard some people say that the first step to starting a phone company is to purchase billing software. While this may not be true, the complexity of the plans offered by cell phones for completed transactions requires sophisticated billing plans. More generally, it is absolutely essential that you can map the completed transaction to the business model. In the case of the phone company, where the transaction is a phone call, the key data of who originated the phone call, who received, and how long it took is essential to the business model.

- *Audit trails.* Many transaction-based business models require audit trails that can be used to prove/disprove specific billing issues. This can be especially vital when attempting to reconcile the differences that can occur between various business entities that participate in the transaction. You may also need to cross-reference the transactions created by your system with those created by other systems.

- *Make certain transactions are uniquely identified.* Transactions must be uniquely identified. Be wary of simple database counters – these often won't work in today's distributed environments. Instead of relying on the database vendor, create a truly unique identifier through a reliable algorithm. I've had good luck with the built-in algorithms for creating Universally Unique Identifiers (UUID) in Unix-based systems or the Globally Unique Identifier (GUID) available on MS-Windows based systems.

- *Understand transaction states, lifecycle, and duration.* When I'm standing in line at the checkout counter about to make some purchase, my mind often leaps to images of mainframe computers processing hundreds of credit card transactions per second. There are, of course, myriads of other kinds of transactions. Many of them have complex states, lifecycles, and durations. Managing the duration of the transaction can be particularly complex, especially when a transaction can "live" over a system upgrade. The complete transaction lifecycle must be defined, because it impacts your back office systems that account for transactions.

  Suppose, for example, that a transaction can last as long as one year and you bill your customer monthly. Normally, you bill for a transaction when it has completed. However, in this case, you could be waiting a long time for money, negatively impacting your cash flow. If you know that 80% of transactions complete successfully, you could safely bill for the transaction when it is started, *provided* your system can recognize a failed or aborted transaction and your back office systems can properly adjust and/or otherwise credit your customers account.

### 4.3.4 METERING

Metering business models bring up a host of interesting challenges, especially when the metering is based on a concurrent or named user. Let's address some of the issues that are common to both models first.

- *How do you authenticate users?* Authentication is the process of attempting to answer the question "Are you who you say you are"? There are many authentication approaches, from simple user names and passwords to simple-to-use but hard-to-defeat tokens, to advanced (and costly) biometric systems. If your business model is derives a lot of money from uniquely identifying users, you should consider employing any number schemes beyond simple user names and passwords. More generally, you should work with other, established infrastructure technologies to authenticate users (such as LDAP based technologies).

- *How many users?* In a concurrent user system there is some limit as to the number of users that can concurrently access the system. The manner in which you specify the number of concurrent users is subject to considerable variation. I've worked on systems that span the security gamut of managing this value. On the low end, we specified the number of concurrent users as a plain text entry in an "INI" file. Completely insecure, but entirely acceptable for our requirements. On the high end, we specified the number of concurrent users in a signed license that was then stored on a hardware device using advanced cryptography. To the best of our knowledge, this approach has yet to be cracked!

- *How are you going to count concurrent users?* While you can, and often should, rely on other parts of the infrastructure to authenticate users, you may not be able to rely on them to count users. This may be a function of your application, or it may be obtained by integrating a license manager into your application.

- *Are they gone or inactive?* Session management is a key issue in concurrent and named user applications. Once a user is logged in, you can't assume that they will explicitly log out. Something might go wrong: their client might crash, or they might not remember to log out. The general solution is to associate a timeout parameter that forcibly logs off the user or drops their session after a set period of inactivity has expired.

  Unfortunately, setting this value isn't trivial. It must be tuned, based on how your application is actually used. Consider an interactive voice system that is accessed by a cell phone. Is a period of inactivity caused by the user pausing to consider their next action or has the line dropped because they drove into a "dead zone"? If they did drive into a "dead zone", how will you re-establish the session so that they don't have to re-enter a long string of menu commands? It is important to make certain that you provide plenty of configuration parameters on these values so that your application's administrator can properly tune your application.

Consumptive resource management places severe restrictions on your tarchitecture. The primary reason is that you have to maintain some kind of state. If your customer has purchased "100 hours" of total use, you need to record how long they've actually used the application. One approach is to store these data on a centralized server, which can work if you structure it so you can't be spoofed and you always have a reliable network connection. Another approach is to store usage locally. But be careful: it is often trivially easy to reset usage data by erasing previously stored values.

### 4.3.5 HARDWARE

The biggest issues with hardware based models are defining what constitutes the hardware and how you're going to manage it relative to the business model. What, exactly, is a Central Processing Unit (CPU)? The concept of a "CPU" is in many ways meaningless: Many personal computers and engineering workstations now with at least two "processing units" as part of their standard equipment, and some new chip designs have multiple processing units in a single package. Arbitrarily picking one as "central" seems a bit silly. That said, "per-CPU" business models are common. To support them, you will have to define a "CPU" and the how it will be enforced in your system.

## 4.4 ENFORCING LICENSING MODELS

Once you've defined your business and licensing model, and ensured your tarchitecture can support them, you have to decide how strongly you wish to enforce the licensing model. "Enforcement" typically means disabling some or all of the application when the license manager determines that the licensing scheme has been violated. You can avoid creating or using a license manager and rely on the honor system, create your own licensing manager or license one from a third party provider.

### 4.4.1 THE HONOR SYSTEM

The honor system is the simplest and easiest approach to enforcing a license model. You simply expect your customers to honor the terms of the license model! This means not changing license terms, illegally copying the software, changing configuration parameters to obtain more use of the software, and so forth. You're not giving up any rights, as you still have legal protection under contract law if you find your customer cheating. Instead, you're not putting anything into your application to explicitly prevent your customer from cheating.

I suspect that a large percentage of software publishers rely on the honor system. Whether or not this is a good choice should be based on a careful analysis of several factors, including the relationship you have or want to create with your customer, the potential for lost revenue, and your ability to track the use of your software via the honor system verses the cost of creating, maintaining, and supporting, a more advanced licensing system or licensing such a system from a third party. In consumer software, where you may have tens of thousands to tens of millions of copies, the honor system may not be a good choice. In enterprise class software, where you may have several dozen to a few hundred customers, the honor system may be a good choice, especially if you maintain a close relationship with your customers through support or service organizations.

### 4.4.2 HOME GROWN LICENSE MANAGERS

In this approach the developer team creates the infrastructure for license management. Such solutions are not industrial strength, in that they are often relatively easily defeated. However, they have the advantage in being low cost, lightweight, easy to implement, and completely in control of the development organization. Once you've made the choice to enforce the licensing models, the economics of what happens should the enforcement be defeated must be considered to determine if a home grown solution is sufficient or if a professional system is required. If you're software costs $700 per license, a home grown system may be acceptable. If you're software costs $70,000 per license, there is simply too much money to lose via home grown license manager, and you're better off switching to a professional license manager.

**It Doesn't Have To Be Unbreakable**

One enterprise class system I helped create provided for a combination of named or concurrent users: any number of users up to the concurrent user limit could log on at the same time, but only those users explicitly identified were allowed access. We implemented our own simple, lightweight, but extremely effective licensing manager. The number of concurrent and named users were simply stored in an INI file, with user ids and passwords stored in a database.

Changing either of these entries defeated the licensing scheme, in that if you licensed 20 concurrent users but changed this to 100 you had just cheated my employer out of 80 concurrent user licenses and tens of thousands of dollars in license fees. We didn't expect anyone to actually cheat, and to this day I know of no company that did. The net was that for a reasonable development effort we were able to devise a scheme that "kept honest people honest".

Note that some kind of home-grown license management is almost always required for session-based licensing schemes, because the development organization needs complete control over how the software responds to the system running out of the set of internal resources that the system needs to manage sessions. In other words, do you stop the application (extreme, not recommended) or simply refuse the session (common, but the issues associated with refusing a session are application specific).

### 4.4.3 Third-Party or Professional License Managers

Professional license managers are typically organized in three main components: the license generator, a client, or local license manager and a server, or remote license manager. The nature of the business model and the software being enforced will determine how you use these components.

- *The license generator.* The license generator is a component that generates a valid license for consumption by the client and/or server. Most license managers will generate digitally signed licenses that cannot be altered and that can be used to prove the rights delivered to the license manager are correct. License generators are typically located at the ISV or their agent and are integrated with other backend infrastructure systems, such as order fulfillment systems, which may initiate the generation of a license, and accounting systems, which may use the records maintained by the license generator to create billing records. Once a license is generated it can be distributed in a variety of ways, including fax, phone, email, or via direct connection from the server to the license generator via the internet.

- *Client.* The client license manager is used to manage the software running on an end users computer or workstation. In its simplest terms, the purpose of the client license manager is to either allow or prevent access to a given application. The client can typically be configured to operate in one or two ways. The first is a standalone, in which the client manages enforcement issues without working with the server. This mode of operation is common for single-user, consumer oriented software and works well for time-based access or usage models.

Here is an example that illustrates this mode of operation. You work for “SuperSoft” who markets “SuperDraw”. You want to distribute a 30-day trial of SuperDraw. This can be thought of as a free “rental”. When SuperDraw is installed the client-side license manager is also installed that enforces these rights along with the digitally signed trial license (e.g., the trial expires and the software cannot be used after 30 days).

The second mode of operation requires the client license manager to work with a server to enforce license rights. This mode of operation is common in business oriented software that uses metering, such as named or concurrent user. When the application is invoked, the client license manager checks with the server to determine if access is allowed. This mode of operation requires your customer to be connected to a network, and you will need to consider application behavior when your customer is not connected.

- *Server.* The server component interprets digitally signed licenses and provides a variety of enforcement services to client license managers. As described above, the server component is required (in one form or another) for the licensing models that support some kind of counting or metering of license terms.

  Here is an example to illustrate how a server works with a client to enforce license rights. You're a high-end CAD system that wants to sell concurrent user licenses at $8K/user. In this case the software can be installed on any available workstation, but each user using the software consumes a concurrent user. The function of the server is to work with the client to keep track of each concurrent user and ensure that the license terms are properly enforced. While this model may seem great, remember that you still must handle the unconnected user.

Although I’ve talked about enforcement, I haven’t addressed how your software interoperates with the client and/or server components of the license management system. “Normal” software, or the software that you create to solve your customers needs, has no concept of enforcing a business model. Something must be done to the software to prepare it for license management. The way that you may do this is quite varied for home grown systems. Third party license managers, on the other hand, generally employ two approaches to integrate the software with the license manager: Injection or APIs.

- *Injection.* Given a "normal" application, the "injection" approach analyses the object code and "injects" into the object code new code that typically obfuscates and/or encrypts the original code and adds the client license manager to enforce the license. In other words, injection works just like a virus, albeit a beneficial virus. Table 4-2 lists some of the advantages and disadvantages of the injection approach.

| **Advantages** | **Disadvantages** |
|---|---|
| • Requires little or no work by developers.<br>• Can result in more secure protection, because the injection approaches obfuscate and/or encrypts code. | • Increases size of code.<br>• Decreases execution performance.<br>• Can only be used with binaries; typically not suitable for interpreted languages. |

**Table 4-2**

- *API.* In this approach developers, write to an API or SDK provided by the license manager vendor. The API provides for such things as license registration and enforcement (e.g., it has calls for "checking the validity of the license" or "logging in a concurrent user". While the API approach is not strictly required for such things as concurrent licensing, it makes implementing such schemes vastly easier. The API approach can be used with interpreted languages, but most vendors feel that using an API in this manner does not provide very strong levels of security. More plainly, it is relatively easy to hack the API approach in Java/C#. Table 4-3 captures advantages and disadvantages of API-based approaches.

| **Advantages** | **Disadvantages** |
|---|---|
| • Provides maximum flexibility – you can control exactly how the license model | • If not used properly can be easy to defeat. |

| **Advantages** | **Disadvantages** |
|---|---|
| works.<br>• Can be used with interpreted languages, but don't be lulled into a false sense of security. | • Creates tremendous lock-in to an existing vendor. Once you integrate their API, you're probably going to be using this vendor for an awfully long time.<br>• Usually takes longer to implement a complete solution. |

**Table 4-3**

While professional third party license managers have many virtues, you need to evaluate any license manager very carefully. Consider the following issues when conducting this evaluation.

- *Business model support.* To the best of my knowledge, no licensing manager supports all of the business models listed earlier. For example, I don't know of any license managers that provide direct support for most hardware based business models (such as Per-CPU or per expansion card). Most work best if you alter your business model to work well with their specific technologies.

  Traditional license managers provide a fixed set of models with parameters that are filled in by the license manager. An example is a license for a time based usage scenario, in which you simply fill in the amount of time. More modern license managers provide a license scripting language, similar to languages like Visual Basic, that allow you create customized scripts for creative licensing models.

- *Platform and operating system support.* Make certain your license manager vendor can provide support for all required platforms and operating systems. While examining their supported platforms, also take the time to explore their development roadmap. When a new version of an operating system is released, you're stuck until your license management vendor supports it!

- *Check cracker web sites to determine the strength of their solution.* It is easy to create a simple license manager. It is *very hard* to create an industrial strength license manager – one that will consistently thwart crackers, maintain the security of your software, and ensure that you're getting the maximum revenue from your licensing model. If you don't have the skill to assess the strength of their solution, hire a consultant who can.

- *Check backend integration and volume capabilities.* As stated earlier, the license generator is almost always integrated with backend systems. Examine your potential license manager vendor's ability to integrate their system within *your* environment. While you're doing this, make certain they can also meet your performance, volume, scalability, and stability requirements.

- *Make certain their operational environment matches yours.* License managers create a whole host of operational issues. For example, customer service representatives may have to regenerate a license or create, on the fly, a temporary evaluation license, or cancel a previously generated license. You'll have to make certain that the integrations created between your license generator and other backend components are sufficiently scalable and reliable. Make certain that your license manager vendor can meet your operational requirements.

- *Check branding and user interface control.* When the code that is enforcing the license detects a violation, chances are good that some kind of error message will be displayed to the user. Assess the degree of control you have over the content and presentation of this error message. You want to make certain that it is meeting all of your usability requirements, especially internationalization. It stinks to discover your vendor has twelve dialogs they might bring up in obscure circumstances, and have never heard of DBCS or Unicode.

- *Examine license content and format.* The format and content of the license should be understandable and should match your business requirements. Anything that is generated by your system, even if this data is not used by the license manager, should be digitally signed. For example, you may wish to put a serial number inside the license to make integrating the license generator with other backend systems. Any custom or proprietary data that you may store in the license should be signed.

- *Examine license distribution capabilities.* Licenses can be distributed in a variety of ways. Make certain the vendor supports the approaches that are most important to you.

## 4.5 MARKET MATURITY INFLUENCES ON BUSINESS MODEL

The maturity of your target market is one of the strongest influences on the selection and management of a given business model. In the early phases of a given market, business models should be chosen so that they can be quickly and easily understood, primarily because you may not be certain of the best way to structure the business model. You may find that your customers prefer an annual license to a subscription, or that they expect discounts if they purchase in bulk. Moreover, despite the best intentions of the business plan, you might find that innovators and early adopters expect and/or demand special terms.

As the market matures, chances are good that your business model will need to become increasingly complex in order to serve the idiosyncratic needs of different market segments. I helped one client whose growth had stalled attack a new market segment with the same underlying system simply by defining a new business model. The original business model consisted of an annual license. The new business model was based on a pay-per-use model. The easy part was modifying the underlying architecture so that both business models could be supported. The hard part was creating the appropriate price points so that a given customer could choose the best model for them without harming the relationships between current customers.

The enforcement of business models also matches the maturity of the target market. In the early stages of a market, enforcement tends to be lax. As the market matures, or in cases where you suspect piracy, the need for enforcing the business model increases. My experience is that marketects and tarchitects take enforcement *far too lightly*. Software piracy is a serious problem. You've worked hard to create your system. Create a business model that identifies the real value provided to your customers, price it competitively, and enforce it accordingly. Just remember that onerous enforcement will lead to dissatisfaction among honest customers. Be careful.

## 4.6 CHOOSING A BUSINESS MODEL

Choosing a business model is one of the most challenging tasks faced by the marketect, as it requires incorporating everything that has been discussed in this chapter *and* several factors that are beyond the scope of this chapter, such as the business and licensing models offered by competitors (which may constrain you to existing market expectations) to corporate and/or environmental factors beyond your control (such as when another division does poorly and you need to find a way to increase short term revenue). To help you through the potential morass of choosing a business model, consider these questions.

1. Who is the target market? What do they value?

A crisp description of the target market and what they value is the first step in creating an appropriate business licensing model. If you're not certain of what they value, consider how they want to use what you're offering.

2. What are your objectives relative to this target market?

In an emerging market you may wish to capture market share, so create simpler models. In a mature market you may wish to protect market share, so create more complex models to provide flexibility.

3. What is your business model?

Pick one of the business models defined above and customize it to meet your needs.

4. What rights do you wish to convey?

Begin by asking your legal department for a "standard" contract, as it will contain a variety of non-negotiable rights and restrictions. See what you can do about everything that is left.

5. What is the affect of this business model on your software architecture?

Work with the tarchitect to make certain that any business model you propose is appropriately supported.

6. What is the pricing model?

The business model provides the framework for defining how you're going to make money. The pricing model sets the amount the customer will pay. You'll need to consider such things as volume discounts, sales and/or channel incentives, and so forth. Pricing choices may also affect your software architecture, so make them carefully.

As you develop the answers to these questions, you're likely to find that the best way to reach a given target market will require a variety of changes to your current business model, licensing model, and software architecture. You'll have to rank order the changes that these require in all areas of your product so that you can reach the largest target market. The benefits will be worth it, creating just the right business and licensing model is good for you and your customers.

## CHAPTER SUMMARY

- Your business model is how you make money.
- Business models are associated with, and to a large extent define, license models.
- Your license model are the terms and conditions you associate with the use of your software.
- The most common software related business models make money by:
    - providing unfettered *access* to or *use* of the application for a defined period of time;
    - charging a percentage of the *revenue obtained* or *costs saved* from using the application;
    - charging for a *transaction*, a defined and measurable unit of work;
    - *metering* access to or use of the application, or something the application processes;
    - charging for the *hardware* the application runs on, and not the application itself;
    - providing one or more *services* that are intimately related with application operation and/or use.
- Business models associated with users (such as concurrent user licensing) motivate integration with corporate systems that manage users (such as LDAP servers).
- Make certain you understand every right associated with your business mode. Separating rights may provide more opportunities to create revenue.

- License models may be enforced by home grown or third party professional license managers.

## CHECK THIS

- ❑ Each member of the development team can define the business models currently in use, or under serious consideration for the future.
- ❑ Our license agreements are congruent with our business model.
- ❑ Our license agreements define the specific set of rights provided to customers.
- ❑ We have chosen an appropriate mechanism for enforcing our business model.
- ❑ The costs of changing architecture to support alternative business models is understood and communicated to marketects.

## TRY THIS

1. What is your business model?
2. How well does your architecture support your business model? Why do you claim this?
3. Can you demonstrate support for a new kind of business model? For example, if your current system is sold on an annual license, could you easily add support for some kind of concurrent license, in such a way that you could open a new market for your software?
4. If you're using a license manager, have you allocated enough time in your project plan to properly integrate the license manager into your application?
5. Are your target customers likely to be innovators, early majority, majority, late majority, or laggards? How does this characterization affect your business model?

# 5. TECHNOLOGY IN-LICENSING

You can't build everything new. Every complex software system is part new code, part systems integration with previously written software—even if the software that you integrating with is nothing more than the C runtime library that comes with your favorite compiler. What you're integrating with is a product from another company. As described in the previous chapter, this product comes to you based on a business and license model. The process of licensing this technology and incorporating into your offerings is called technology in-licensing.

There are many motivations for licensing technology from others. Increasingly complex systems require us to license some technology because it can be cheaper and faster to license it than build it on our own. Sometimes, licensing isn't a choice, but a requirement, as a company may have obtained a key patent on technology essential for your success. An additional consideration include the skills and experience of key staff. You may not want them spending precious time and energy designing and building components that you could obtain via a license.

Any, or all, of these factors means that you are likely going to be licensing one or more key technologies from another party. As a result, understanding basic concepts associated with in-license agreements, and how they affect your tarchitecture, is a vital skill for everyone in the team.

*Disclaimer: I'm not a lawyer. Any contract that may affect your system, including any license agreement, should be thoroughly evaluated by a properly trained attorney. That said, an attorney can only do a good job when they understand the business objectives as well as the relationship between the licensed technology and the underlying tarchitecture. Thus, everyone benefits when the key members of the understand the core issues associated with technology in-licensing.*

## 5.1 LICENSING RISKS / REWARDS

The introduction outlined some of the motivations for licensing technology. While these motivations are quite real, there are also risks associated with licensing technology. The following table captures both motivations and risks that must be considered in every licensing situation.

| **Motivation / Reward** | **Risk** |
|---|---|
| You can reduce, manage, or otherwise eliminate complexity and/or risk by licensing technology from a 3rd party. The technology or provider is more of an "expert" in a given area that you deem important to your own success. By licensing their technology you gain their expertise. | You may be able to shift complexity from your team to the technology provider, but in doing so you're increasing risk by increasing your reliance on 3rd party technology. If their technology evolves in a way that fails to match your needs, you could be in serious trouble.<br><br>A supplier that changes their focus could leave you scrambling to find an equivalent replacement. This is *not* as easy at it may sound, and you may not even be able to find a truly plug-compatible supplier. The effects can be devastating.<br><br>This almost happened to one of my teams when a search engine vendor whose technology we had licensed decided to reposition themselves as a portal vendor. They initially stopped development of their core technology (text indexing and searching). Fortunately, several customers, including us, managed to convince them to maintain their investment in their core technology. Everything worked out, but the situation was very tense for several weeks as we explored several undesirable replacement scenarios.<br><br>Finally, make certain that you assess the viability of the technology provider. Given today's volatile corporate market, there is always the risk that a small technology provider may go out of business. |

| **Motivation / Reward** | **Risk** |
|---|---|
| In-licensing technology promotes the creation of component-based software systems, and we know that this is a "good thing" because component-based software systems can be easily changed. For example, you can easily replace one component with another! | While component-based systems are a worthy goal of a good tarchitecture, in-licensed technologies often become far more inter-twined into a solution than anyone realizes it is too late to make changes to support a different implementation.<br><br>To use a trivial example, suppose you in-license a reporting component. Chances are good that you could only replace it with a different implementation *iff* that implementation supported the same level of functionality. If you expose or rely on any vendor-specific functionality you've probably locked yourself to this vendor. If this was a conscious choice, great. If not, you might be stuck. |
| In-licensing technology makes our system easier to construct. We can focus our efforts on creating our unique technology and easily configure and manage in-licensed technologies. | You may have reduced complexity, but you haven't eliminated it. Managing in-licensed components increases configuration complexity. Incompatible business models may make the use of certain kinds of technologies practically impossible. I'll elaborate on this later in the chapter.<br><br>Another issue deals with the restrictions that may come with various components. Consider high-end cryptographic libraries, which are often subject to various forms of export restrictions. Licensing these kinds of technologies means that you're subjecting your software to all of their restrictions. |

| **Motivation / Reward** | **Risk** |
|---|---|
| You can obtain "protection" by licensing technology protected by a patent. | Really? Check the fine print of the licensing agreement. Indemnity, which is legal exemption from the penalties or liabilities incurred for using the component is hard to secure. In plain English, suppose you license a component from company A because they have a patent on a key technology. This license is not likely to protect you from being sued by company B who may claim that you're infringing on their rights. |
| You can reduce "time-to-market" by reusing technology. | Licensing technology does not always result in faster time to market. At the very least you have to invest time in learning the technology, integrating it into your solution, and verifying that  it works correctly for your total solution. There are times when it really is faster to build your own technology, from scratch, that meets your needs (consider your choices for creating of licensing a license manager, discussed in the previous chapter). |
| Vendor created components are "higher quality" than components you write on your own. | Many times this just isn't true, and technology that you license is often lower in quality than what you create from scratch. |
| Vendor created components are "lighter", and consume less resources, such as memory or processor cycles (presumably because they have been optimized). | Again, many times this just isn't true. Technology that you license may be surprising "heavy", consuming more resources or running more slowly than code you write on your own. Moreover, it is nearly impossible to substantially tune the performance of most in-licensed technologies: You're left with the "switches" the vendor gives you and not much else. You can't recompile someone else's library to turn on multi-threading or modify their code to perform I/O operations more efficiently. |

| **Motivation / Reward** | **Risk** |
|---|---|
| Licensing a component relieves some of the burden associated with technology currency because the vendor will be continually improving this component. | Vendors don't always update components as fast as needed. Vendors sometimes drop support for other components that you must still support. For example, vendors sometimes drop support for an operating system that you might still wish to support. |
| The component is "state of the art", and using it will "future proof" your application. | This sounds like another example of "resume driven design", in which developers seek to use a technology because it is "cool". You can either naturally and easily justify the use of a given technology based on your real needs – or you can't. If you can't, drop it. |
| Licensing technology is cheaper than building it from scratch. | Maybe. Maybe not. The claims that it is cheaper to license a technology are usually based on building an "equivalent" replacement. Chances are good that you don't need an equivalent replacement, which substantially lowers development costs. License fee structures can also ruin the economics associated with a good product, as discussed later in this chapter. |

### Fixing Their Code

In one project we had decided to license an essential technology from a new startup. Unfortunately, their libraries had some real problems. The APIs were poorly documented. The technology didn't scale to handle our needs. The code was written in C++ and riddled with memory leaks. The technology lacked a small number of functions that would greatly enhance our ability to use it.

We could have dropped the technology, but it was felt by all involved, especially product management, that we simply had to have it. Since the underlying algorithms were protected by some very strong patents, we couldn't simply create our own solution. In fact, even if the underlying technology was not protected by strong patents we probably would not have created our own version, as the component was based on an set of extremely sophisticated set of mathematical algorithms, and it would have taken several months for my team to build the knowledge necessary to create the basic technology needed to implement the desired functionality. Moreover, we had been given direct access to their source code, and developing a duplicate or replacement version would have put us on very precarious legal grounds (if you're really going to try and do this you need to employ a wide variety of techniques, including clean-room reverse engineering, to try and protect yourself as much as possible).

The best choice was to work with the vendor to help them modify their technology to meet our needs. I instructed my team to prepare a careful document that outlined all of the problems with the technology, including the memory leaks. We also documented what we'd like to see in the APIs, prioritized the additional features we wanted, and provided the vendor with sample code that we had written to test their technology. As you can guess, the vendor was stunned by our apparent generosity. They had never expected that we would work so hard to help them be successful. Of course, the truth is that we were working for our own benefit – we needed this technology. The vendor adopted our suggestions and I've maintained a strong, positive relationship with them ever since.

## 5.2 CONTRACTS – WHERE THE ACTION IS

The heart of any technology licensing is the contract that defines the terms and conditions associated with the use of the technology. This section outlines some of the basic elements associated with contracts and some of the terms and conditions commonly found in technology contracts.

### 5.2.1 CONTRACT BASICS

A valid contract requires three things: an offer, acceptance, and consideration.

1. An offer. An offer is a statement by an entity (the offeror, a person or a corporation) that indicates a willingness to enter into an agreement on the terms stated.

2. Acceptance. Acceptance occurs when the entity to whom the offer was addressed (the offeree) indicates a willingness to accept the offeror's proposed agreement.

3. Consideration. Consideration is anything of value that is exchanged by the parties. Technology licensing contracts usually specify monetary forms of consideration as payment terms, discussed in greater detail in the next section.

Pretty boring, huh? The real action is in the license terms, discussed next.

### 5.2.2 LICENSE TERMS

Before you can understand how license agreements and the terms they contain can affect your tarchitecture, you have to have an idea of what kinds of terms can exist. This section discusses terms that are commonly found in technology in-license agreements and what they mean.

#### DEFINITIONS

A precise, legal description of all of important terms referenced in the license agreement. Descriptions of technology often go something like this:

> "Software" means the object code software proprietary to {licensor} and listed on the pricing schedule.

or this:

> "Software" means the object code software proprietary to {licensor} and listed on Attachment A.

Pay attention to what is listed, because these definitions will often include specific version numbers, supported operating systems, and so forth. If you need something that isn't defined, you may find yourself unable to obtain support and/or upgrades, or you'll have to create another contract to pay your supplier to provide you with the needed or required technology.

#### USAGE OR GRANT

Defines the manner in which the in-license technology can be used. This section may be one of the longest in the agreement, as it often contains many of the other terms discussed in this section.

#### DURATION OR TERM AND OTHER KEY DATES

The duration of the agreement – when it begins, when it ends. By working together, and by negotiating precise agreements about the specific dates in the agreement, the marketect and the tarchitect can create a better overall result for their company. To illustrate, suppose that you agree to license a key technology under an annual license fee. You estimate that it will take you four months to integrate the technology and expose it as a set of new features to your customers, another month to QA the integration, and another two months to roll this out to your customers. From your perspective, the ideal contract would allow development with the new technology to commence once the contract was signed but would delay payment of the initial annual license fee until the technology was actually delivered to customers, under the reason that until the technology was used by a customer it was only a cost to the licensor. From the perspective of the technology provider, payments should begin when the contract is signed, because they are not responsible for how long it takes you to integrate the technology, and you're getting the benefit of using their technology from the moment it is delivered. Who's right depends on your role in the negotiating process.

Most in-license agreements specify a variety of other important dates beyond the beginning and end of the agreement. If you don't understand these dates you can be headed for a lot of expensive trouble. At a minimum, keep track of the following dates.

| **Date** | **Why Important** |
|---|---|
| Effective Date | The date the agreement is considered in effect. |
| Expiration Date | The date the agreement is ended. May be specified any number of ways, including an absolute expiration date or a calculated date (such as adding a fixed period of time to the effective date). |
| Payment Dates | The dates when fees are due are usually listed, including the payment terms (e.g., "Net 30" or "payment shall be made the 15th and last day of each month"). |
| Audit Periods | Specifies the period of time in which the licensor can audit how you've used their technology. |
| Termination Notice | Specifies the amount of time that must be provided to the other party in order to terminate the contract. You should specify this period of time so that it is long enough to properly replace the licensed technology should the license be terminated. Many agreements do not specify a long enough period of time. |
| Other Dates | Contracts can specify a wide variety of additional dates depending on the license and the contract requirements. You might be required to report various aspects of usage to the technology provider according to a well-defined schedule. OEM and partnership agreements may specify quarterly, bi-annual, or annual technology reviews. Timetables associated with new releases may be listed in the contract, along with penalties should these dates be missed. |

**The Costly Renewal**

One product I worked on had licensed a cross-platform database access library. The license agreement clearly specified that the term of the agreement lasted until a specific date. When that date arrived, the product development team had a simple choice: renew the license agreement or reengineer the product to remove the database access library. We chose to reengineer the product, primarily because the vendor wanted to raise the price by several thousand dollars! We also felt that we could improve quality and build a higher performance implementation using a mixture of freely available technology along with a new database access layer. Unfortunately, we couldn't finish the reengineering effort by the specified renewal data, so had to renew the license agreement, at a substantial cost to the company.

A variant of this specific example is the "automatic renewal" associated with many contracts. With an automatic renewal, once the contract has been signed, it automatically renews unless you explicitly cancel the contract. It is vitally important that you remain aware of these renewals. Circumstances change, and you may not want to renew the contact.

### TERRITORY

The applicable territory where the in-licensed technology can be used. It is especially important to be careful of geographic restrictions, because they can be very hard to honor in our web-connected world.

### SPECIFIC USE

A license agreement may distinguish between development, quality assurance, technical support, and commercial uses of the in-licensed technology. It is important to understand what the agreement covers. Note that general terms are sometimes captured in one agreement while specific terms are captured in other, related agreements.

### EXCLUSIVITY

Exclusivity refers to the degree to which the licensor agrees not to license the technology to anyone else. In general, exclusivity is hard to achieve, although there are a variety of ways in which it might be obtained. For example, suppose that you wish to license a key technology but only intend to use it in Europe. It might be possible to obtain exclusivity for the European market. Exclusivity can also be obtained for higher fees. In most circumstances, however, you don't really need exclusivity, so it is not a term that you should worry much about.

### SUBLICENSE

Sublicensing refers to the degree to which you can license the technology to a third party. Sublicense rights are usually required for embedded technologies. For example, suppose that you license in a core technology from Company A for use in an information appliance. Chances are very good that you're going to license your solution to your customers, which requires sublicense rights from Company A. As I will discuss later in this chapter, sublicense rights often have a substantial impact in your tarchitecture.

### TERMINATION

Termination refers to the set of clauses that allow one party to terminate the contract. All of the contracts I've seen contain at least one termination clause, and most of them contain several. Most contracts contain clauses that allow either party to withdraw from the agreement if there is a breach in performance. Breaches in performance are usually specifically stated, such as failure to deliver by a specific date, or failure of a system to meet defined processing requirements. Enumerating and describing potential breaches and remedies is often one of the more time consuming aspects of contract negotiation. Withdrawing from a contract because of breach is harder than it seems, because there is usually a process for recovering from the breach (the remedy).

Many technology licensing contracts have clauses that allow either party to withdraw from the agreement provided they give the other party sufficient advance warning, ranging from as little as 30 days to as much as one year. This period of time is usually not long enough, and you should always try to negotiate a longer period of time. Replacing an in-licensed technology with another or with an technology developed in-house can be added to that long list of tasks that "always take longer than planned"!

Additional clauses associated with termination include such things as terminating the contract if either party goes bankrupt or fails to meet defined financial criteria, terminating the contract if one party elects to drop support for the technology in question, or terminating the contract if there is a substantial change in control (such as when a competitor acquires a technology provider). Although termination sections can get quite lengthy, it pays attention to read them.

### RENEWAL

Technology license agreements often contain one or more renewal clauses. These clauses may be automatic, which can actually cause a company to pay too much money in fees depending on the evolution of the tarchitecture. Automatic renewal clauses can also create a false sense of security. In general, I recommend against automatic renewal clauses, as this forces you to evaluate each license agreement to make certain it is still meeting your needs.

### FEES OR PAYMENT TERMS

The foundation of a valid contract requires some form of consideration, and in a technology licensing contract the details of the consideration are usually found in this section. As discussed in chapter four, there is a wide array of creative licensing and billing models, all of which end up in this section of the contract. What is vitally important is that your tarchitecture support the payment terms required in the contract. If your in-licensing a technology based on a transactional business model, *your* tarchitecture needs to support *their* business model. A key point of contention is when the business model required in the license is different than the business model you use with your customers. Such differences can often only be resolved through careful negotiations, as discussed in the next section.

### DEPLOYMENT RESTRICTIONS

Some license agreements restrict one or more deployment options associated with the technology. For example, the vendor may require one agreement to use the technology for a customer deployment and a different agreement if the licensed technology is to be used as an ASP. Deployment options are discussed in greater detail in chapter seven.

### OTHER OR GENERAL RESTRICTIONS

In addition to the terms that define what you *can* do, license agreements will also carefully define a variety of things that you *cannot* do. As with the other terms, any of these restrictions can have an impact on your tarchitecture. For example, it is very common to see restrictions that prevent any form of reverse engineering or modification of the licensed technology.

The practical effect of this kind of restriction can be fairly surprising. Suppose, for example, that a developer finds a bug in a licensed component. A restriction against reverse engineering may prevent the engineer from analyzing the technology to identify the bug. Let's say that this restriction doesn't exist, and that you instruct the developer to research the bug. Even if they find a fix to the bug, you may still be out of luck, as a restriction to modify the licensed technology means that you can't apply the fix. You may only be allowed to supply the fix to the vendor and wait for them to issue a patch or new release. Since the best thing you can do is influence their development plans, you might be waiting a long time.

### NON-COMPETE

Vendors may require that the solution you create does not compete with their technology. In other words, you have to create a new offering. This may sound silly, but it prevents people from doing things like licensing a J2EE web server under the pretense of integrating the technology into a new product without really creating a new product. The net result is that this would result in a new solution that was substantially competitive to the original vendors solution.

### ACCESS TO SOURCE CODE

Technology agreements often specify that a copy of the source code will be placed in escrow and be given to the licensee should the licensor breach. In theory, this is great, because it makes certain that you can get full and complete access to the technology you need. In practice, this is often worthless. Suppose, for example, the vendor does breach and you're given access to the source code. What then? Chances are good that you won't have the necessary skills or staff to manage the source code you've acquired!

### MARKETING REQUIREMENTS

The agreement may obligate you to issue a press release, allow either or both the licensee and the licensor the right to use the name or corporate logo of the other on their web site, or any number of other marketing related requirements associated with the technology. Read these sections carefully, as marketing and branding requirements can impose nasty surprises on your tarchitecture.

## 5.3 When Business Models Collide, Negotiations Ensue

Before you actually license any technology, you have to make certain your business model is compatible with your technology providers business model. If they're not compatible, you're going to have to negotiate some way to reach an acceptable agreement. These choices can have a significant impact on the tarchitecture, so everyone needs to be involved in the decision.

Suppose, for example, that you're building an enterprise application and you want to base your business model on concurrent users. You want to integrate a search engine, and your preferred vendor wants to charge an annual fee. While the business models are different, this situation can be fairly easily resolved. If the annual fee is low enough, you can just pay it. If it is too high, you might be able to negotiate an annual fee that is acceptable, and again, pay it.

A more complex strategy is to negotiate a license fee based on the projected revenue from your concurrent users, with a "floor" (a guaranteed minimum amount you'll pay) and a "ceiling" (the maximum amount you'll pay). You can pay the floor to gain access to the technology, and at the end of the license term (one year), you provide the licensor with a statement that details your actual revenue and the amount of money you may owe them. This isn't an easy choice, because it requires you to disclose potentially sensitive information to your technology providers (such as the number of concurrent users and the amount of your annual revenue). Along the way, your tarchitecture, or more likely, your back office systems, will need to be checked to ensure that you can meet license payment requirements.

The situation is often significantly more complex than this example. Let's invert the above example and see what happens. In this case, your business model is based on an annual license and your technology provider's business model is based on concurrent user. Some ways to resolve these incompatibilities include creating a mutually agreeable estimate for the number of concurrent users accessing your system for a given annual license; negotiating with your technology provider to accept an annual license instead of charging by concurrent user; or offering your technology providers technology as an "optional" module and licensing it as a concurrent user option. If you really need their technology, and you can see no realistic way to make it an optional module, and they are adamant about maintaining their business model, you may have no other choice but to convert your business model to a concurrent user business model. As a result, it is vital that you understand the business models of all your technology suppliers and how they relate to your own.

**Nice Try, But…**

One product team I managed planned for their system to be built on a major J2EE vendors application server. The target market was both end-users and application service providers who would operate the product on behalf of end-users. The default license agreement of the J2EE application vendor explicitly prohibited the right to operate their product in an ASP or service provider environment. Product development stalled until I was able to negotiate a fee schedule that protected the interests of the vendor and was acceptable to our company.

A special case of business model negotiation is required when you change technology vendors. Let's say that you've created an enterprise application and have licensed a search engine from technology vendor A. Sometime later, for whatever reason, you decide to change to technology vendor B. Along with the payment fees associated with new systems sold using vendor B's technology, you're also going to have to calculate the fees associated with upgrading systems that contain vendor A's technology to systems that contain vendor B's technology. Depending on your licensing agreements, you may or may not be able to charge your customers for the upgrade and/or change to vendor B's technology. Marketects must model the complete set of costs associated with changing technology vendors, including the cost of converting the installed base.

## 5.4 HONORING LICENSE AGREEMENTS

Just about anything you can think of has, or will be, specified in a license agreement. Examples of issues that are commonly covered in license agreements that are likely to affect your tarchitecture include:

- *Definition of Technical Terms.* This is probably the biggest single area of license compliance. tarchitects should be aware of the technical definitions used within the contracts. Do all the definitions reflect the actual product under development? Are the version numbers in the contract correct? What about the defined operating systems and operating environments? Do you have the rights to use the product in the manner envisioned by your business plan? More specifically, can you operate the in-licensed technology as an ASP? Can you fully embed the technology in your product? What is the degree or nature of the embedding? Are there any geographic or export restrictions?

- *APIs.* Can you simply expose any third party APIs that are provided in the license agreement? Chances are you can't. And trivially wrapping third party APIs with your own API won't cut it. Most license agreements require you to "substantially enhance" the functionality of an API before it can be "exposed" (whatever *that* means...).
- *Support.* Who fields what kind of support question? What kinds of information must you capture and forward to the third party? Certain license agreements can get very precise.
- *Branding.* Do you have to include visible or direct attribution of the third party component? Consider the ubiquity of the "Intel Inside" marketing campaign and you'll get a sense of just how important third-party technology suppliers consider such attributions.

## 5.5 MANAGING IN-LICENSED TECHNOLOGY

A proven technique for managing in-licensed technology is to create a "wrapper" or "adapter" for the technology. Instead of programming to the API provided by the vendor, you create an abstraction that allows you, if needed, to replace this abstraction if needed. A common example of this approach in Java is JDBC, which provides a common interface for databases. While this approach may make it easier to replace one vendors technology with another, it is not without its drawbacks. Wrapping frequently introduces a least-common-denominator approach, in which the development team cannot avail themselves of superior, but proprietary, technologies. Wrapping takes time and it must be tested. If the technology is never replaced, or it is extremely unlikely that it will be replaced, then the additional expense associated with wrapping an in-licensed technology is not justified. The decision to insulate or protect your tarchitecture from direct access to an in-licensed component must be made by with the support and involvement of both the marketect and the tarchitect.

## 5.6 OPEN SOURCE LICENSING

Open source software presents a wide variety of options for both marketects and tarchitects. Using key open source technologies within your products can provide substantial benefits to you and your customers. This section assumes you've evaluated a given open source technology and found that it meets your technical requirements. This is neither a blanket endorsement or indictment of the quality or appropriateness of a given open source technology for your solution. It is merely stating that, for whatever reason, you want to use an open source technology as part of your overall solution.

The first step is read the specific license that governs your technology. Not all open source licenses are created equal, and there are differences. While it may seem that these differences are minor, differences are the stuff that lawyers love to bill about!

When you're finished reading the license, look for the sections that govern how the licensed technology can be incorporated into other technologies, for that is likely to your most important area of concern. According the *Open Source Definition, version 1.9*, your concerns are likely to be unfounded: it is entirely permissible to incorporate a portion of an open source technology into your product (see also the GNU Lesser General Public License), *provided* you maintain the same rights for the incorporated technology, and *provided* you meet the other terms of the specific license.

It is beyond the scope of this book to provide detailed legal advice for doing this (remember, I'm not a lawyer). Practically, this means that you can create a new, for-profit work using a variety of open source technologies. This is likely to your advantage, and open source strategies should be considered by both the marketect and the tarchitect.

## 5.7 License Fees

Third party technologies come with a variety of license fees. A good way to think about license fees is that anything you license represents the business model of some technology provider. As a result, you may have to deal with any of the business models described in the previous chapter, or, for that matter, any business model the vendor has identified as being useful. Fortunately, technology providers tend to offer their products in a reasonable number of business models. The most common approaches, and their likely impact on your tarchitecture, are described below.

*Pre-paid fees*: In this arrangement you pay an agreed upon fee for the use (time-based access or usage) of the technology. These fees are paid whether or not you actually use the licensed technology. This arrangement usually results in minimum impact on your tarchitecture, as you are often given maximum flexibility in integrating this technology into your system. These fees must be included in the cost estimates provided by the marketect to justify initial development and in the ongoing costs associated with maintaining the system.

*Usage-based fees*: In this arrangement you pay an amount based on some measured usage of the in-licensed technology, often with some minimum payment required to access the technology (metering). This arrangement always has an impact on your tarchitecture because you must ensure you are able to comply with the terms of the license agreement. Specifically, you must make certain that your tarchitecture can capture the metering data associated with the contract. Clearly, it is advantageous for the marketect to negotiate a usage model based that is conveniently implemented.

As described earlier, when the fees are variable the in-license technology vendor will often require a minimum payment, often referred to as a "floor". You'll want a "ceiling", or the maximum amount you'll have to pay – which they will resist. The strength of the technology supplier, the kind of technology that is being provided, the quality of the relationship, and the overall volume of the expected deal are all factors that play a part in negotiating usage-based fees.

*Percentage of revenue fees*. In this model you little or no up front fees to license the technology, but instead pay the technology provider a percentage of the gross or net revenue associated with those elements of the system that use the licensed technology. Like pre-paid fees, this arrangement tends to have little impact on the tarchitecture. It can, however, have a fairly substantial impact on the marketecture, requiring both companies to agree on precise definitions of the fee.

While the specific fee structure specified by the technology vendor will motivate the exact negotiating strategy, there are some universal strategies. I've found negotiating for the following useful.

- *Protection from product obsolescence.* A marketect needs to know that their technology providers are going to be able to support their needs for as long as necessary. If they intend to support a given platform or operating system, they should make certain that their technology providers are also going to support this platform.

- *Protected pricing.* Whatever fee structure is chosen, marketects should try to negotiate such provisions as capped price increases, favored pricing plans (in which no other licensor will be given better terms than the licensee; and should such better terms be offered, they will also be automatically applied to the licensee), and volume or usage discounts.

- *Milestone payments*. One client of mine made a very costly licensing mistake: they licensed a key technology from a vendor, paid a very large up front fee, and then subsequently failed to deliver their technology to the market. A better approach is to base fees on key milestones that represent revenue you're going to receive from the use of the technology.

  Suppose, for example, that you've decided to license a core technology based on an annual fee (the usage fee). You should structure the payments according to key milestones. During initial development, when the technology is being incorporated into your product and you're not making any money, the most you should pay your technology provider is a small access or development fee. The next payment, and typically the largest payment, should be upon release or launch of your product. The final payment may be structured some months after product has been in the market.

- *Training and development costs*. You may be able to obtain free or discounted training or educational materials, preferential access to support or development organizations, or special review meetings in which you're given the chance to meet with key members of the technology provider to make certain your needs are being met.

Whatever the licensing arrangement offered by your technology provider, the actual costs must be given to your marketect for proper financial modeling. Simply put, too many license agreements, or even just one license agreement with onerous terms, can seriously erode any profit margins you have for a product. In extreme cases the wrong fee structure can actually kill a project.

### Negotiate Fees Based On Value

Modularly designed software systems in which each module is a separately identified and priced can provide the marketect the greatest advantage in making certain in-license fees are kept as low as possible. In one system I created we had licensed in technology from two vendors based on a percentage of revenue. We isolated the usage of these in-licensed technologies in specifically created modules and separately priced each module. These modules were not optional, but they were separately priced. We then structured the overall price of the system so that the largest cost element was the core system, which consisted of technology that we had created without any license fees.

This approach served two fundamental purposes. First, it was inherently fair. Because the in-licensed technology was not used in the core system, our technology vendors had no right to receive fees associated with this component. Second, it reduced the percentage of the average selling price that was subject to license fees resulting in greater profits.

## 5.8 Licensing Economics

Let's say you're one of those lucky developers who have been asked to explore new technologies for your next system. Lucky for you, because you want to learn Java and J2EE, as you think it will be a good career move. You download the trial version of SuperSoft's J2Server, a fully J2EE 1.1 container, and use it to build a slick prototype. Indeed, the prototype is so useful, you show it to your boss and use it to convince her that using Java, a J2EE-based architecture, and especially SuperSoft's J2Server is the obvious choice for your project. Fortunately, your boss wants to make certain her next release is fully buzzword compliant, so when you suggest this approach she readily agrees.

Believe it or not, you might have just designed yourself into a real jam. Many evaluation and/or trial licenses explicitly prohibit the development of software for commercial uses. And even if you bend the rules and ship the prototype to a customer for evaluation purposes, you *certainly* cannot release your software unless you've secured the necessary rights.

As discussed throughout this chapter, licensing issues must be aligned with your tarchitecture. Saying that you're going base your application on J2Server isn't going to buy you anything if you don't know how SuperSoft is going to charge you for their products. In fact, integrating something like J2Server could break the economics associated with your application, something that a good marketect will be able to tell with a basic economic analysis The following exercise illustrates why.

Let's assume that you're company requires a gross profit margin of 30% and that your marketect estimates the average selling price of an annual license at $100K. The current estimated direct and indirect costs associated with this project are $65K, without SuperSoft's licensing fees. So far, so good; your average estimated gross profit is $35K, or 35%. Now, let's assume that SuperSoft wants to charge 15% of the gross annual revenue. This means that on a $100 license you will be paying

SuperSoft $15K. You can model this as an increase in costs (from $65K to $80K) for each sale, with the net result that you're average estimated gross profit just dropped to $20K, or 20%. Something needs to change.

While the above example is intentionally simplified, it illustrates that it is vitally important that the *total* projected and/or actual costs of technology in-licenses must be incorporated into the product and business plan. Without them, you're not likely to create a winning solution.

## CHAPTER SUMMARY

- Every system has some in-licensed technology.
- Effective technology in-licensing requires a full understanding of both the risks and the rewards by both the marketect and the tarchitect. This understanding creates a stronger negotiating position and leads to more advantageous terms.
- Every technology in-licensing contract comes with a plethora of terms that directly or indirectly affect both the marketecture and the tarchitecture. Both the marketect and the tarchitect must read and understand these contracts so that they can properly manage the in-licensed technology and honor all necessary license terms and conditions.
- It is best to align all of the business models that comprise the total solution. Usually this is accomplished through several rounds of negotiations.
- There are a variety of techniques to manage in-licensed technology. The most fundamental technique is to insulate your tarchitecture from the in-licensed technology. Theoretically, this is easy. Practically, this is usually very hard.
- Understand the fee structures and associated costs the license agreements.
- Technologies based on open source licenses are increasingly attractive as sources for in-licensed technologies.

## CHECK THIS

- ❑ We have a valid contract for every licensed component or technology.
- ❑ We understand the key terms and conditions associated with each in-license. We have looked at all causes for breaching the contract and are certain that we won't breach because of ignorance.
- ❑ We have assessed the impact of replacing a licensed component or technology.

- ❑ We have created compatible business models.

## Try This

1. What in-licenses do you have?

2. What are their key terms? important dates? rights?

3. Do you have substitutes/alternatives to the licensed technology? Can these substitutes/alternatives be used to improve your operating and/or negotiating strength?

4. Are you properly supporting all aspects of the business models required by your technology in-licenses in your tarchitecture?

# 6. Portability

Marketects and tarchitects alike often pursue portability with a fiery passion. What fuels this fire? Customer need? "Me-too" feature management ("Our competitor supports this feature -- clearly, we MUST")? The desire to prove technical virtuosity? By exploring the business issues associated with portability you can determine how hot to build your fire.

## 6.1 The Perceived Advantages of Portability

Here are some of the claimed advantages of portability. Do they hold up?

Claim: By supporting multiple platforms, we can address a new market segment.

This is a compelling claim. Addressing more market segments with essentially the same product is a good thing. But be careful. Effective marketing practices teach us that the key to long term success is *successful* market segmentation. If you're basing your market segmentation on the kind of operating system or hardware platform you're customers are using, you may be choosing the wrong approach.

Claim: By supporting multiple platforms, we demonstrate that we can meet our customers idiosyncratic needs.

This marketecture claim has a tarchitecture corollary: By supporting multiple platforms (or standards) we can demonstrate technical skill. Sadly, technical skill rarely wins customers. Solving their needs does. Portability can detract from this, especially when customers who choose a platform for specific features find that you're product does not support these features because they are not "cross-platform" portable. I refer to this as a the *portability paradox*. Customers choose a platform for the unique features and perceived benefits of that platform… but… most portable applications are explicitly designed to avoid "platform specific features"!

My experience indicates that the real motivations for portability aren't that impressive. Here are some that I've stumbled across.

Real Motivation: Developers think writing portable software is cool.

It can be a lot of fun to write something that is portable. Portability can stretch your limits as an engineer, and creating an architecture that achieves key objectives (such as performance or stability) across multiple operating environments is an impressive accomplishment.

It can also be a real pain and no fun at all to create a portable application. Subtle differences in operating systems, even operating systems from the same vendor, can cause that “should” work to break. Sometimes there is simply no way to offer the same capabilities in a portable manner, which means you have to compromise either your product objectives, your technical approach, or both. Many developers find learning about multiple operating environments tiring.

Real Motivation: One or two early, key customers demanded different solutions.

Unless a product is managed with considerable discipline, it is easy to do “anything” to secure early, key customers. Sometimes this includes porting the application to a different platform in order to secure a new customer. This is usually a mistake. When a marketect finds customers who demand different solutions in the early stages of the product they should continue to search for a better customer (where “better” is defined as “within the target market”) and not direct the development team to port the code too quickly. Experience has shown that this is arguably the biggest motivation.

## 6.2 THE BUSINESS CASE FOR PORTABILITY

The collective experience of hundreds of projects demonstrates that writing cross-platform, portable code is well within the skill level of most development organizations. But just because you can write such code, should you? The only valid reason for creating portable solutions is that doing so will ultimately result in a larger, more profitable product.

The revenue side of this equation is based on whether or not you’ll obtain a sufficiently large market and charge them enough to be profitable. Marketects are often good at identifying these revenue drivers. Unfortunately, they often forget some of the key cost factors. Consider the following:

1. The costs of training the developers, QA and support people in how to develop within, test, and support each platform.

2. The costs of purchasing, configuring and supporting the hardware and software for each supported platform. Note that each group has different needs with respect to each of these

activities. Developers are most productive when you give them fast hardware-often the best choice is the fastest possible hardware. QA and support need a range of hardware choices to adequately reflect the performance options that match target customer segments. Someone has to support all of this hardware. If your IT department isn't up to the task someone in the development, QA, or support team will have to assume this responsibility. This often creates a hidden cost.

3. The testing time for the developers and QA to make sure that the product works properly on each platform. As a general rule, the larger and more complex the matrix of pain (the complete set of your supported platforms) the longer it will take to release your product. This isn't necessarily a cost, but instead lost revenue.

4. The complexity of managing multiple release cycles. Suppose that you choose to support Solaris, HP-UX, and Linux. This means that you must track three operating systems. When Sun releases a new version of Solaris, customers will want to know when you are going to support the same. Each time you choose to support another platform you relinquish a bit more control over your release cycle.

These costs can only be justified by a sufficiently large target market. One company I worked with supported Solaris-Oracle and Windows-SQLServer. Approximately 90% of their installed base ran Windows-SQLServer. This installed base accounted for more than 80% of the total corporate revenue. The cost to support a cross-platform product was not recovered and the company actually LOST money on a spurious marketing claim.

The following conditions should be met before building cross-platform code.

1. Your market analysis identifies sufficient revenue to justify the incremental total product development and support costs associated with each platform.

2. You have included the total cost associated with developing, testing, and supporting ALL of the supported platforms. This means the necessary hardware and the skills necessary to maintain and configure these machines.

3. You have sufficient development resources that allow you to create, test, and support multiple platforms.

4. You understand the relative impact of the various platforms you must support on your development efforts and can manage the same. For example, if you're going to support Solaris and Windows you have accounted for the differing release schedules of Sun and Microsoft.

A good rule of thumb is that it is easier justify a portable *technology* than a portable *application*. By “technology”, I mean a solution that is designed to be a component of a larger solution. Examples of portably technologies include such things as relational databases, or communication libraries that are offered to large target markets. The example of Oracle in the book *Crossing The Chasm* is a classic example about how a portable technology helped a company win significant market share.

Applications, which are not often designed to be a component of a larger solution, can often achieve a sufficiently large and profitable market share within a given operating environment that portability doesn’t make sense. There are countless applications from profitable companies that run only on one operating environment (e.g., Sun Solaris, MS-Windows, IBM 360, and so forth). These companies, for a variety of reasons, have decided to focus on a given target operating environment.

Rules of thumb are just that: rules of thumb. There are also numerous counter-examples of platform-specific technologies and portable applications. The issues are complex, but all of them boil down into well-defined target markets that ultimately determine how the development team should create their application.

**Portability is *Always* About the Money**

I’ve managed both ends of the portability spectrum. In one job in which we were creating a new kind of enterprise class system for an emerging market, we explicitly designed a system that was not portable. It was designed to run only on Microsoft products – MS-Windows ‘NT/2000 and SQL Server, written using Microsoft development tools. Our first challenge to this approach was when we tried to sell the system to Sun Microsystems. As you can guess, the initial response was “No thanks, we don’t want a system based on Microsoft technology ” (although the original wording associated with the rejection was a *bit* stronger). Sun wanted us to port the system to Java/J2EE running on Solaris and Oracle. We also received a “No” from Pfizer, on the grounds that our Microsoft based solution didn’t fit their corporate requirements for Solaris and Oracle.

It was difficult to receive these early rejections, because we knew that Sun and Pfizer would be good customers, and we were a young and hungry startup. But a simple cost analysis showed that we couldn't afford to port the application with the available resources. When we asked Sun for the necessary development funds, they said "No". I don't blame them, for we estimated it would cost several million dollars to port the application.

Although the situation felt bleak, everyone supported the decision to maintain a platform specific focus – an amazing show of discipline, especially for a young and revenue hungry startup! Nonetheless, we kept improving our technology and growing our customer list. After each release, we would talk again with Sun and Pfizer. Eventually, something amazing happened: The system had reached the point where its feature set was so compelling that these two companies simply couldn't live without it. Furthermore, their competitors were adopting the system – they had to license it. And they did. First Pfizer, and then later, Sun, licensed the system. The key lesson I learned in this scenario is that building the good solution on a single platform is more important than building a mediocre solution on many.

I've also managed at the other end of the portability spectrum. In a different job I managed server-side software that ran on Microsoft, Sun, and Linux operating systems, and client software that ran on Windows and Macintosh. Unlike the previous example, these were not complete applications (solutions), but core technologies that were used embedded in our customers environments as part of a larger solution. In this case, the technology requirements justified a portable solution. On the server side, the revenue distribution was almost evenly split between customers running Windows and Unix-based operating systems. On the client side, our biggest customers, including Adobe, Symantec, Macromedia, Corel, and Roxio, required both Windows and Macintosh technologies. Indeed, they continually asked us to port our technology to additional platforms.

## 6.3 Creating Portable Applications

Let's assume that there is sufficient economic justification for writing a portable application. Here are some techniques that I've found helpful in making this happen.

- *Use an interpreted language.* From lisp to Smalltalk, Java and perl, interpreted languages have an immediate portability advantage over compiled languages because the interpreter provides for a much-needed layer of insulation from the underlying operating system. Of course, this doesn't mean that an interpreter is foolproof, because differences in operating systems often find their ways into interpreters. Resource handling, threading, file system manipulations, network libraries, and user interfaces are all areas that present challenges to the portability of interpreted languages. As a general rule, however, it makes sense to use an interpreted language if you're planning on building a portable system (provided, of course, you have checked to make certain that your choice in language has an interpreter on every platform you intend to support).

If you're using a compiled language, you also have to check to make certain that a compiler exists on every target platform. In addition, you will need to educate the development team on the specific idioms that enhance portability within that language. If you're using C or C++, these techniques include using carefully designed source code management and build systems that select the proper file for a given target platform as well as conditionally including specific source code based on the language. Handling conditionally compiled code is more challenging than normal code, because it is harder to write and maintain than platform-specific counterparts, but the overall process is quite manageable. I know, because I've done it. You can too.

- *Persistent Storage.* By persistent storage, I mean the ability to store and retrieve data on a persistent media, such as a hard drive. For simple data use XML stored within whatever "file" abstraction is provided by the target platform. For complex data use a relational database, accessed through an intermediary layer like OBDC, Java's JDBC, or Perl's ODB. Although ANSI SQL is far less portable than it should be, strive to use it. Limit the use of vendor-specific embedded procedures, such as Transact SQL.

- *Business logic should be portable.* The area of your system that should be most portable is your business logic. A well-architected system should isolate the business logic; several authors discuss this at length, including POSA, Fowler, Hohmann, and Yourdon.

- *Closer to the user means less portability.* The areas of greatest portability lie in backend infrastructure. Moving from the server or backend towards the user, portability tends to decrease. Despite a variety of vendor claims to the contrary, the user interface is the area of lowest portability. To illustrate, there are substantial differences in color schemes, fonts, display resolutions, and so forth, all areas that are intimately associated with the user interface. These realities, which don't appear likely to change, suggest a very practical approach to managing investments in portability. The greatest investments in portability should be in backend or infrastructure (e.g., invest in "server-side" portability before "client-side" portability).

- *Use XML for standardized, interoperable communications between subsystems.* In complex distributed systems that run on different platforms, you will eventually have to resolve the issue of how various subsystems should interoperate. Fortunately, you won't have work too hard, for the bright and talented people who have invented the core technologies that are powering web services, including XML, SOAP, WSDL, and XSLT have already done all the heavy lifting. Use XML to support interoperable communications between subsystems – you'll be happy you did!

  When an application is centralized on a single platform, or when you require extremely high performance, you might want to consider a non-XML solution. Converting in-memory data into XML and back again consumes a fair amount of resources and is not the best choice for every situation. Do this very carefully, as my own experience shows that XML is, in general, a far better choice than other choices, even though it is slower.

- *Avoiding hiding the power of a specific platform in the name of portability.* Consider the area of database portability. While the major vendors claim to support ANSI SQL, they all provide vendor-specific variants tailored to different operations. Oracle's commands for manipulating hierarchies are too powerful to ignore for many applications. SQLServer 2000 commands for manipulating XML are just too fast to ignore. Challenge the notion that portability must mean taking the "lowest common denominator". Structure your tarchitecture so that you can take advantage of the power that exists from specific platform choices.

## 6.4 The "Matrix of Pain"

Let's assume that you've made the decision to support more than one platform. Perhaps the market is fragmented, and that supporting multiple platforms is the only way you can create sufficient revenue. Perhaps your target market simply demands multiple platforms. This is common in enterprise software when a customer standardizes on one platform for their IT infrastructure. Or, perhaps your development organization has chosen a very low cost and highly portable implementation, such as a web solution written in Perl that interfaces to MySQL, making the actual cost to support an additional platform extremely low.

I've found that one of the most important things that can be done to minimize the pain of portability is to make certain that the development and QA organizations understand the relative priorities of the market. Without clear priorities, *everyone* is going to waste precious time developing and testing the wrong parts of the system, or not sufficiently testing the parts of the system that matter the most to customers. The best technique for prioritizing is to create market-driven configuration matrices for use by the development and QA organizations (the "matrix of pain"). They are a variant of the "all pairs" technique that is used by QA for organizing test cases. The marketect should drive this process, with participation from development, QA, services, sales and support.

Suppose that you're building a web based system and you'd like to support the following:

- five operating systems on two platforms (Solaris 2.7 & 2.8, MS-Windows NT 3.5.1, 4.0, & XP Server),
- two web servers (IIS & Apache, for simplification, omitting versions);
- two browsers (Netscape & Internet Explorer), and,
- four databases (Oracle 8i & 9i, SQLServer 7.0 & 2000).

The total number of possible combinations for development and testing is thus {OS * WebServer * Browser * DB } = { 5 * 2 * 2 * 4 } = 80, which I'll take as a given is too large for your development staff. That's OK, because if your staff was large enough to handle each individual combination its too large anyway. Your QA manager estimates that his team of three people can handle perhaps 7-9 configurations. Your development manager will follow the QA managers lead, under the agreement that development will work on the primary configuration, and that QA will certify all others. As is common in cross-platform development, the strongest constraints come from QA, not development. Thus, you've got You've got to trim 80 possible configurations down to 7-9, in way that ensures your most important configurations are covered.

Step 1: Remove configurations that don't make sense or are explicitly not supported

A simple reading of the list of supported components demonstrates that many don't make sense. No company that I know of runs SQLServer 2000 on Solaris 2.8! In addition, the marketect may explicitly choose not to support a possible configuration. For example, the marketect may know that Oracle 8i is only supported on Solaris 2.6 because an important customer has not yet migrated to Solaris 2.7. Discussions with this customer indicate that when they migrate to Solaris 2.7 they still want to use Oracle 8i, until the system has proven itself stable, when they will migrate again to Oracle 9i. Thus, step 1 of reducing this matrix is to identify possible configurations, as shown below in Table 6-1.

| | Operating System | | | | | |
|---|---|---|---|---|---|---|
| | Solaris | | MS-Windows | | | |
| | 2.6 | 2.7 | NT 3.5.1 | NT 4.0 | XP Server | |
| Apache | PC | PC | PC | PC | PC | |
| IIS | NA | NA | PC | PC | PC | |
| | | | | | | |
| Netscape | PC | PC | PC | PC | PC | |
| IE | NA | NA | PC | PC | PC | |
| | | | | | | |
| Oracle 8i | PC | PC | NS | PC | PC | |
| Oracle 9i | NS | PC | NS | PC | PC | |
| SQLServer 7 | NA | NA | PC | PC | PC | |
| SQLServer 2000 | NA | NA | NS | PC | PC | |
| | | | | | | |
| Total Configurations: | 1 | 2 | 4 | 16 | 16 | 39 |
| PC = Possible Configuration | | | | | | |
| NS = Not Supported | | | | | | |
| NA = Not Applicable | | | | | | |

**Table 6-1**

An interesting effect emerges as you engage this reduction. Because you need *one* of each of the elements to create a complete configuration, the total number of configurations is dramatically reduced. There is only *one* supported configuration for Solaris 2.6: Apache, Netscape, and Oracle 8i, and only *two* for Solaris 2.7.

Step 2: Rank order the configurations

Although 39 is smaller than 80, this matrix is still not sufficiently prioritized. The next step to creating a manageable matrix is to work with other groups to prioritize every possible / supported configuration. In this process you will gain insight into a variety of things, including which configurations:

- are actually installed in the field, by actual or perceived frequency of installation;
- are used by the largest, most profitable, or otherwise "most important" customers;
- are going to be most heavily promoted by marketing in the upcoming release (these *have* to work);
- are most easily or capably supported by the support organization;
- are most likely to provide you with the coverage you need to test full functionality.

There are a variety of techniques to achieve a suitable prioritization. I've spent the bulk of my career working in enterprise class software, and for most of the products that I worked on we were able to prioritize the most important matrix pretty quickly (usually in one afternoon). Once you're finished, consider color coding individual cells red, yellow, and blue for "must test", "should test", and "would like to test but we know we probably won't get to it" to convert your matrix into an easily referenced visual aide.

For larger, more complex software or for developers or managers who insist on numbers, presumably because they believe that numbers will help them make a better decision, assign each major area a number between zero and one so that all the areas add up to one. These numbers represent consensually created priorities. Using this approach, suppose that in discussing the matrix it became apparent that no known customers actually use Netscape or Oracle 8i on Windows XP Server, nor are any expected. This results in a further reduction of the number of configurations to be tested, as shown below in Table 6-2.

| | Operating System | | | | | |
|---|---|---|---|---|---|---|
| | Solaris | | MS-Windows | | | |
| | 2.6 | 2.7 | NT 3.5.1 | NT 4.0 | XP Server | |
| Apache | 1 | 1 | 0.2 | 0.5 | 0.3 | |
| IIS | | | 0.8 | 0.5 | 0.7 | |
| | | | | | | |
| Netscape | 1 | 1 | 0.2 | 0.2 | 0 | |
| IE | | | 1 | 0.8 | 1 | |
| | | | | | | |
| Oracle 8i | 1 | 0.5 | | 0.1 | 0 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Oracle 9i | | 0.5 | | 0.3 | 0.3 | |
| SQLServer 7 | | | 1 | 0.5 | 0.2 | |
| SQLServer 2000 | | | | 0.1 | 0.5 | |
| | | | | | | |
| Totals: | 1 | 2 | 4 | 16 | 6 | 29 |

**Table 6-2**

Step 3: Make the final cut

You've still got more work to do, as 29 configurations is clearly too many. You've got to finalize this matrix and get the number of configurations to a manageable number. Be forewarned: This will take at least two passes through the matrix. As you begin this process, see if you can find additional information on the distribution of customer configurations, organized as a Pareto chart. This will prove invaluable as you make your final choices.

In our first pass, consider the impact of the installed base. Suppose that while only 20% of the installed base uses Solaris, they account for 53% of overall revenue. In other words, you're going to be testing all three Solaris configurations! This leaves you with four to six possible configurations for MS-Windows.

Your product and technical roadmaps (see appendix) tell you that NT 3.5.1 is going to be phased out after this release, and that only a few customers are using it. Based on this information, everyone agrees that one configuration, NT 3.5.1, with IIS, IE, and SQL Server 7, will be OK.

You know you need to test Apache and Netscape. Furthermore, you believe that the bulk of the customers are going to be on NT 4.0 for quite some time, and want QA to concentrate efforts here.

Your knowledge of the architecture leads you to believe that the database access layer uses the same SQL commands for Oracle and SQLServer on any given operating system. Just to be sure, you ask your tarchitect or development manager to ran a quick binary comparison on the source code. Yes, the SQL is the same. This doesn't mean that the databases will operate in the same manner, just that if you certify your code on one of them, such as Oracle 8i, you have reasonable evidence that it should work on the other. This knowledge produces Table 6-3. Note that all major configurations are tested at least once.

| | Operating System | |
|---|---|---|

| | Solaris | | MS-Windows | | | |
|---|---|---|---|---|---|---|
| | 2.6 | 2.7 | NT 3.5.1 | NT 4.0 | XP Server | |
| Apache | X | X | | X | | |
| IIS | | | X | X | X | |
| | | | | | | |
| Netscape | X | X | | X | | |
| IE | | | X | X | X | |
| | | | | | | |
| Oracle 8i | X | X | | X | | |
| Oracle 9i | | X | | | X | |
| SQLServer 7 | | | X | X | | |
| SQLServer 2000 | | | | | X | |
| | | | | | | |
| Totals: | 1 | 2 | 1 | 8 | 2 | 14 |

**Table 6-3**

Unfortunately, you have 14 configurations. This is at least 5 more than your estimates allow. Removing IIS from NT 4.0 removes four more configurations. The final set of configurations is listed in Table 6-4, which is more than you think you can test in the allotted time.

| | Operating System | | | | | |
|---|---|---|---|---|---|---|
| | Solaris | | MS-Windows | | | |
| | 2.6 | 2.7 | NT 3.5.1 | NT 4.0 | XP Server | |
| Apache | X | X | | X | | |
| IIS | | | X | | X | |
| | | | | | | |
| Netscape | X | X | | X | | |
| IE | | | X | X | X | |
| | | | | | | |
| Oracle 8i | X | X | | X | | |
| Oracle 9i | | X | | | X | |
| SQLServer 7 | | | X | X | | |
| SQLServer 2000 | | | | | X | |
| | | | | | | |
| Totals: | 1 | 2 | 1 | 4 | 2 | 10 |

**Table 6-4**

From here, you'll have to find additional ways to either test the product as you wish or further reduce the number of configurations. The most likely ways to make the cut is by obtaining more hardware, either internally or from your customers, or engaging a beta program in which your beta customers handle testing configurations that QA won't.

I have intentionally simplified this example for the purposes of this book. In a real web-based system, you would probably support more versions of the browser (4-8), more versions of the web servers (2-4), and would probably specify other important variables, such as proxies and firewalls. You also need to add an entry for each version of the application that might exist on the same platform in order to check for any negative interactions between versions of the same application. These would dramatically increase the potential number of configurations. In one multi-platform, multi-lingual, internet-based client application, we had more than 4,000 possible configurations. We were able to reduce this to about 200 tested configurations – through a lot of work.

This approach to prioritizing configurations works for smaller numbers of configurations, but it starts to break down when there are a lot of configuration elements or lots of possible values for each configuration element. When this happens, you need to resort to more automated approaches for organizing your initial set of configurations and then prioritizing these based on the market parameters I mentioned earlier. The technique is referred to as the *all pairs* technique, and it ensures that all pairings of parameters are covered in the test cases, without covering all combinations. James Bach has posted a free tool to calculate all pairs test cases at www.satisfice.com. I want to stress that the output of all pairs should reviewed to ensure that all of the market issues presented earlier are covered. Specifically, when you get into a "don't care" condition, prioritize based on marketing needs.

Understanding the matrix of pain makes it crystal clear that *every time* you add another platform you're increasing the total workload associated with your system.

## 6.5 BEWARE THE PROMISES YOU MAKE

Be vigilant about managing contracts and stating what you support. I once inherited a contract that had been signed with a customer committing the company to supporting "Macintosh OS 8.1 and later". This committed our product marketing team to a future that we may not have wanted to embrace. Even more drastically, it created a nearly impossible task for my development team! Practice vigilance when creating portable software with respect to contracts. At the very least, make certain the versions and platforms you support are very clearly identified. You may also wish to specify individual support policies for various platforms, such as limiting the amount of time you will provide support for an older platform.

## Chapter Summary

- Portability advocates make many claims to support their desires. Common claims include:
    - addressing a new target market;
    - supporting existing customers, who may prefer a different platform but have accepted the current platform;

  The real reason you may have, or may be creating, a portable applications are often not as grandiose. These may include:
    - Developers think that it is easy and/or cool;
    - One or two early innovators or early adopters demanded different solutions.

  Carefully check whatever set of claims your portability champion is making.
- The only valid business case for creating portable applications is that you'll make more profit by doing so. Marketects are often good at modeling the revenue, but not necessarily the costs, of creating portable applications.
- Creating portable applications is harder than you think. And it *will* take longer than you want.
- It is usually easier to justify a portable *technology* than a portable *application*.
- There are a variety of proven techniques to create portable applications. Learn them.
- Your "matrix of pain" is the complete set of configurations that must be tested in a given release. To create a manageable matrix of pain:
    1. Remove configurations that don't make sense or are explicitly not supported.
    2. Rank order the remaining configurations.
    3. Review the result and make the final cut.
- Always be explicit about the configurations that you support. Be *very* cautious about making commitments to support new configurations.

## Check This

- ❑ Each platform that we support has sufficient development, quality assurance, technical support, and sales resources.
- ❑ We have allocated sufficient time to test on each supported platform.

- ❑ We have made entries into our tarchitecture roadmap that capture the significant releases of our platform vendors.

- ❑ We have created a market-driven configuration matrix to help guide our testing efforts.

## Try This

1. What platforms do you support? Why?
2. What is the…

   market share of each platform?

   cost / benefit of each platform?
3. What might happen if you…

   dropped support for a platform?

   added support for a platform?

# 7. Deployment Architecture

Enterprise class software systems have seen many phases of evolution. Centralized mainframe systems evolved into client/server systems. Client/server systems have evolved into distributed systems. Distributed systems, which are still in their infancy, are now being recast as reconfigurable "web services". Each of these distinct styles of deployment architecture spawned many variants. Emerging web architectures extend these ideas and add new ones, adding alleged new benefits for users and keeping system architects busy. Not surprisingly, we are also carrying each of these major architectural styles into the future, as much for the unique set of advantages they offer as the legacy systems they've left behind.

The wide variety of possible deployment architectures for enterprise class software is starting to trickle down into deployment architectures for common packaged software applications (applications which are "shrink wrapped" and usually cost less than $500). We're now seeing traditional applications offered as a service. As bandwidth continues to increase and hardware devices continue to become more sophisticated, we're going to see a continued explosion in the number of deployment choices.

I use the term *deployment architecture* to describe the manner in which a customer deploys the system for use. This is related to the UML definition of "deployment architecture" but differs in that I am focused more on the strategic implications associated with a deployment choice and less on the lower level details associated with such decisions as how to allocate work among multiple processors in a multi-processor computer system.. Emerging web technologies and business models present tarchitects and marketects with considerable creative flexibility. This chapter will help you sort through some of the business and technical issues associated with choosing a deployment architecture.

## 7.1 Deployment Choices

Common choices for software deployment architectures include the following:

*Customer Site.* This type of deployment is the most traditional and the most common. The software is installed at the customer's site and is configured, operated and maintained by the customer. For common packaged software focused on the consumer market, such as the software

I'm using to write this book, it means software I've purchased, installed, and configured on my laptop. Enterprise class software systems, such as a corporate financial system, warehouse management system, or customer relationship management system (CRM), are almost always under control of the corporate IT department.

Larger, more complex software applications targeted for business uses are usually installed and configured through a consulting assignment. The system vendor may provide professional services to assist the customer or subcontract these services to a major consulting firm (such as EDS, KPMG or Accenture). The duration of the assignment is often a function of the system's effect on existing corporate processes and systems, any or all of which may have to be substantially modified before, during, and often after the installation.

*Application Service Provider (ASP).* An application service provider operates an application for the customer, providing a limited set of services, and rarely providing a complete solution. For example, the ASP may provide 24x7 system monitoring, routine backups, and automatic access to additional bandwidth to the internet, but may not provide phone or technical support should the application encounter a problem. The specific services provided by the ASP must be negotiated. ASPs are more slightly common for business applications, although is changing. For example, many early ASPs were focused on offering large, enterprise applications to smaller customers. Newer ASPs are focused on offering consumer applications, such as estate or tax planning software, as a service.

I am not making a distinction between a software publisher who is offering their application as an ASP and a software publisher who licenses their application to a third party who then offers the application as an ASP. These distinctions, while important for the publisher and the third-party provider, are largely irrelevant for the purposes of this chapter.

*Managed Service Provider (MSP).* A managed service provider extends the concept of an ASP by offering an array of services in addition to the operation of the application. In marketing terms, an ASP competes at the level of the generic/expected product while an MSP competes at the level of an expected/augmented product. The exact set of services offered varies, but usually includes extensive monitoring and backup services, and may include call center support, commerce operations/management, and security/firewall management. Early MSPs targeted the business market. Some recent MSPs have targeted very specific consumer markets, such as corporate email hosting or digital photo appliances.

I anticipate that MSPs focused on very well-defined market niches associated with specialized hardware will continue to emerge. An example of this has already occurred in the financial services industry, where some very large MSPs have proven adept at offering online banking services to millions of consumers. Of course, you don't know this, because the MSP has allowed the bank offering these services full branding control.

In both service provider relationships it is common for service level agreements (SLAs) to be signed that precisely define acceptable performance parameters. Note also that in both types of service provider relationship, when the application is a traditional software system, at least some customer data is stored at the service provider. This has important tactical and strategic implications for security and operations that extend far beyond the actual service being offered to the customer.

*Transactional (Web Service)*. I define a transaction deployment as one that computes an answer in a single whole transaction, often implemented through web service protocols. This style of system is commonly used to provide services to individual users, such as when you are asking for a map on the internet. In certain cases end user data may be stored at the service provider, but this data is rarely corporate data. While this style of system is not yet common in enterprise software, recent efforts to build complex systems around collections of transactional web services may dramatically increase this style of deployment architecture. Ultimately, web services based application architectures will eventually become common for every type of application. This does not mean that they will "win", for web services are not appropriate to every market segment. It does mean that we are going to be faced with increasingly sophisticated choices.

While these four broad categories capture the basic deployment options, savvy marketects have created subtle variants to gain a competitive or positioning edge. For example, some service providers classify themselves as an "Internet Business Service Provider" or IBSP, focusing on a single element of a complex solution (e.g., loan portfolio management for banks). Others define themselves as Enterprise Application Providers (EIPs) and focus their service offerings on a single kind of enterprise class system, in the hopes that they can leverage the experience curve associated with this application across their entire customer base. In the framework presented above, an IBSP or AIP would be classified as either a managed service provider or an application service provider, depending on the specific set of services offered to a customer.

### The Hybrid Deployment Architecture

I've been presenting deployment architectures as relatively exclusive choices: either you choose an ASP or an MSP. In fact, there is no specific technical requirement that the deployment architecture be all one way or another. As is true with other aspects of architecture, it is often best to let the specific needs of your customer and the exact nature of the problem guide you in your choice of a deployment architecture.

One of the more successful architectures I helped create was a true hybrid architecture. In this case, customers needed real time access to more than five terabytes of data while also requiring certain kinds of private transactional data. The solution design was straightforward: put some parts of the solution, such as the transactional data, at the customers site; put other parts of the solution, such as the five terabytes of data, at a service provider. In practice, making this hybrid architecture work was a bit of a challenge, but the benefits to our customers were worth the effort of making it happen.

## 7.2 CUSTOMER INFLUENCES ON DEPLOYMENT ARCHITECTURES

Choosing a deployment architecture that meets your customers expectations is an important part of creating a winning solution. Consider the following customer concerns when making this choice.

*Control.* Customers vary in their perceived needs for "control" over the software system. To illustrate, as a reasonably sophisticated end-user of software, I want the ability to control what extensions or changes I make to my system. Thus, I'm pretty certain that I don't want a managed service for my laptop, because I want the ability to add or remove software as I see fit. But I don't want to control everything! I want my high speed internet connection firewall to just "work". I don't want to spend a lot of time configuring or adjusting this system.

Customers of enterprise class software often have similar demands associated with controlling their systems expressed in a number of dimensions. They may wish to exert control over the systems operational infrastructure. Specifically, your customer may believe that the level of control afforded by a system operating within their premises is absolutely crucial for truly mission critical applications ("I know *exactly* who is going to answer the pager if the system breaks and how long it will take them to fix it”). Conversely, they may be happy to “give up” operational control to a service provider based on their perceptions of the service providers ability to provide high quality, uninterrupted service.

Another issue that is associated with control is the long term retention and management of data. Service providers who approach this topic carefully may be able to make a convincing argument that they can do a better job than the customer. Alternatively, the customer may already have sophisticated policies in place, and may simply feel more confident in their own ability to manage data.

It is imperative that the marketect understand the issues of control that are most important to the target customer when offering an ASP or MSP deployment scenario. Failing to do so will prevent you from creating a winning solution. Conversely, understanding the deeper concerns that your target customer has regarding their deployment choices will enable you to handle them with skill.

*Integration.* As integration needs increase so does the likelihood that the system should be deployed at the customer site. This can range for integration between common desktop applications to linking your CRM system with your inventory management and fulfillment system.

*Concern for data security/privacy.* The nature of the data manipulated by the system influences customer perceptions of an appropriate deployment architecture. Consider a hospital. Payroll data, for example, may not be viewed as sensitively as patient records. As a result, the hospital may opt for a managed service solution for payroll processing but license a patient record management from a reputable vendor and mandate that it be run internally by their own MIS staff. A professional services firm, on the other hand, may have strong opinions about the need for privacy with respect to payroll data, and require that all payroll data be managed in house. In the case of my home firewall, there is no data to manage, so there is no need for data security. Nonetheless, I’ve purchased a hardware based firewall solution because I didn’t want to pay a monthly service fee to my ISP. You need to understand your customers’ relationship to the data managed by your application. I will talk more about data later in this chapter, as developers can easily make poor choices based on faulty assumptions.

*Handling "Peak" Loads.* Depending on the application, it can be more cost effective to place the application at an ASP/MSP that has equipment and/or communications bandwidth designed to handle "peak" loads that are not cost effective to handle with your own dedicated equipment.

*Costs*. There are times when an “xSP” can offer a sophisticated, expensive application at a fraction of the cost it would cost the customer to license and deploy the application at their premises. Sometimes this reduction in expense occurs when the xSP obtains rights to sub-license an application in way that provides access to smaller business. At other times the total cost of the application is reduced because the customer doesn’t have to invest in the total infrastructure needed to make the solution work.

*Vendor Confidence*. How does your customer perceive you? Are you a stereotypical internet startup with programmers sleeping on cots or are you an EDS or IBM style systems integration firm where formal dress is still common? Put it another way: Would you trust your company's most sensitive data to an internet startup that can't properly manage the passwords associated with their internal servers? Answering these questions will help you understand why a customer may be somewhat reluctant to entrust the operation of their mission critical system to your company's engineers.

Earlier, I mentioned that failed ASPs have made customers wary about trusting their data and operations to outside parties. Be aware that your promotional materials may have to move beyond simply touting the benefits of your solution. You may have to provide access to detailed financial statements in an effort to demonstrate that your company has long term viability in the marketplace. This is often referred to as a “viability” test, and unless you can demonstrate that your company will be viable for the next few years, you may not win the deal.

As stated earlier, another aspect of confidence concerns the manner in which you will archive and retain data on behalf of your customers. As your total solution evolves, you will inevitably change the nature of the data that you’re storing. Reassuring your customers that they can always get at old data is an important consideration in the design of your complete solution.

*Skills and Experiences of Operational Staff*. The nature of your application and its deployment architecture will dictate the skills and experience required by your customers’ operational staff. When the application has relatively simple operational requirements, this issue isn’t really a factor that influences a deployment choice. It is when the application exhibits complex operational requirements that the choices become interesting.

The easiest deployment choice, from the perspective of the application vendor, is at the customer: Your customer is becomes responsible for all operational aspects of the systems. They have to "care and feed" the application, monitoring it, backing it up, administering it, and so forth. All of these are skills that must be learned.

If you elect to offer the application as an xSP, or elect to license an xSP to offer the application on your behalf, you or your xSP partner are going to have to assume the appropriate level of operational responsibility to meet the needs of your customer. Be careful, as customers often place greater demands on xSPs than their own MIS staff. You or your xSP partner will have to hire a staff with the experience and skills necessary to create the necessary data center, with appropriate focus on your customers needs. You can subcontract these skills, and many companies do, but you still need someone with the necessary experience to manage the subcontractor.

It is important to consider the operational culture associated with the deployment architecture. If your company started as a highly dynamic, "let's just get it done" entrepreneurial company, you may find it hard to switch the more formal demands associated with operating a data center. In an entrepreneurial company, for example, password management is often lax. In an xSP, you must have tight operational control over such things as passwords.

**Please Disregard Those Credit Cards**

One of my clients who was running a managed service had a rather embarrassing episode when their director of managed services operations distributed a spreadsheet that detailed the growth in transaction revenues for their customers. Unfortunately, he forgot to delete the worksheet that contained the complete full contact information, including credit card numbers, of the customers! As you can guess, a much more appropriate set of operational controls were put in pla ce after this incident. As the vendor, you should make certain this kind of mistake must never happen. As a potential customer of an ASP or MSP, you have the right to demand a careful review and audit of all operational procedures.

*Geographic Distribution.* It can often be easier for a customer to provide an application to geographically dispersed employees or work groups when an ASP manages it.

As you consider these factors, keep in mind that it is your target customer who will provide you with the strongest requirements for a deployment architecture. Earlier in the book I talked about the dangers of "resume driven design", in which designers make technical choices in order to pad their resumes. I've witnessed the same phenomena in choosing a deployment architecture. "Investor driven design" swept through Silicon Valley in the late 1990's as many startups were created to capitalize on the growing wave of popularity associated with xSPs. Many of these xSPs subsequently failed, for reasons that included an inability to understand their customers true motivations for a deployment architecture. Unfortunately, these failures were often quite catastrophic. Many customers of early ASPs irretrievably lost crucial corporate data when these companies failed. Such losses have made corporations justifiably wary about storing key data at a service provider.

For this reason, and others, it is essential that the marketect work to understand not just customer requirements but corporate objectives when choosing a deployment architecture.

### When The Risk Really Is Too Great

One client had created a traditional, enterprise class system for manipulating extremely sensitive data. These data could represent several hundred millions of dollars worth of intellectual property. Their initial customer-site deployment model was a success, because customers were allowed complete control over every aspect of the system. Unfortunately, this deployment model came with a rather high price tag, and my client felt that they had to offer an ASP model to broaden their market. Unfortunately, they failed to understand that the factors described above are not considered in pure isolation by a customer. Instead, each factor is considered and weighed against the other in the context of a winning solution.

In this specific example, the extreme sensitivity of the data meant that my client would have to build a substantial infrastructure and ensure that the data center was being operated under the most exceptional and stringent conditions. They had neither the organizational or operational maturity to undertake this task, and balked at the price. In addition, many of the target customers were global companies who required 24x7 access and support. Preliminary research also indicated that customers didn't want an ASP – they wanted an MSP. Unfortunately, my client didn't want to invest in this additional infrastructure to create an MSP until the new model had proven itself successful.

I urged my client to avoid offering the ASP solution or to create a truly winning solution. To my dismay, they went ahead with their plans and introduced the new model. It failed, and the company eventually went bankrupt.

## 7.3 CORPORATE INFLUENCES ON DEPLOYMENT ARCHITECTURE

The marketect must consider more than customer requirements when choosing a deployment architecture. Sustainable winning solutions require the marketect to understand the capabilities, desires, needs, short and long term strategies of the corporation offering the solution. Here are some of the strongest corporate influences affecting deployment architectures.

*Sales cycle.* The sales cycle refers to the length of time and number of steps that a corporation must undertake in order to make a sale. In general, the sales cycle is correlated to the price and complexity of the software. Consumer software, with low price points and simpler implementation and integration requirements, and a very small number of decision makers (usually one or two) has a relatively short sales cycle. Business software, with substantially larger price points, complex implementation and integration requirements, and many decision makers (usually a committee which may or may not have approval authority) usually has a longer sales cycle. When a corporation is considering a multi-million dollar purchase, they're going to think about it carefully. I've been involved with sales that took almost two years!

In general, most consumer software is still deployed on a PC (a customer site deployment), so I won't discuss its relationship to the sales cycle further. Business software is a different matter. If you seek a shorter sales cycle, consider deploying your software as an xSP or licensing your software to a properly qualified xSP. The sales cycle is usually shorter. The implementation cycle should be. This is important, as once customers have made the commitment to your product they're going to want it up and running as quickly as possible. But watch out! Choosing this option without understanding customer influences is not going to create a winning solution.

Experience with xSPs indicates the situation is even more complex than I've just described. There are times when a customer wants to use an xSP as a quick "starter" solution and then migrate the solution to an customer-site deployment. I've also had to manage reverse migrations, in which customer-site deployments were migrated to a service provider, primarily because of the service providers superior infrastructure. While this kind of deployment migration is extremely rare, at least when this book was written, I expect that it will become more common in the future.

*Infrastructure investment.* When an application provider considers offering their solution as an xSP or Web Service, they must carefully consider the investment that must be made to create a long-term, viable offering. Companies offering their applications in a service-based model routinely underestimate the often significant investment that is required to create a reliable infrastructure capable of handling projected demands.

Alternatively, you don't have to create a solid infrastructure, but then you risk providing poor customer service. Investment calculations must include technical (hardware, infrastructure, and so forth) and non-technical resources (support staff experienced in running data centers, support staff, and so forth). Unless your corporation has the necessary capital to invest in creating a solid infrastructure, and the willpower to do so, I recommend against pursuing an xSP or Web Service.

*Cash flow.* Related to sales cycle and infrastructure investment, the cash flows associated with various deployment architectures must be carefully modeled by corporations. Suppose, for example, that an MSP has arranged to offer a complex enterprise application to customers on a rental basis, with no minimum contract. If the MSP is paying an annual license (common) then they have a single payment due at the beginning of their year of service. If the MSP's customers are paying a rental, they may not have paid enough to the MSP to allow them to pay for the next annual license. Ultimately, xSPs must take a careful, sober approach to the level of cash reserves that are needed for a successful long term operations.

*Flexibility.* An installed base of customers who have your software deployed on their premises can limit your flexibility to rapidly innovate and improve your solution. Customers who have invested time and money making a given release work are usually quite reluctant to modify their working solution. As a result, chances are good that if you go with a customer-site deployment option you will be simultaneously supporting several releases in the field.

This is one of the most appealing aspects of choosing to offer your solution as an xSP or Web Service. By maintaining complete operation control, you gain considerable flexibility in choosing such things as when to upgrade your current solution. In an emerging market, where rapid release cycles are often required to continue to grow, you have the luxury of upgrading your system as often as necessary. Patches can be quickly obtained and installed by development. Of course, appropriate care must be taken whenever you modify your own operational environment, but installing a new release in an operational environment that you control is usually much easier that coordinating upgrades across dozens, hundreds, or thousands of customers.

*Geographic distribution.* An additional element associated with both the sales cycle and the infrastructure investment is the geographic distribution of your target market. If you're trying to sell an application to a global company, they have the right to expect that the deployment choice will provide them with the necessary support and service options. When they choose a customer-site deployment architecture, they can control the level of service provided by their in-house MIS staff, including such things as the language used to answer the phone. When considering using an xSP, the company may require that the xSP provide local customer support.

A global company may also make unforeseen technical demands on the total solution. If the application is deployed in house, the customer is responsible for installing it wherever the application is needed and maintaining it as appropriate, subject to appropriate license terms. If, on the other hand, you're providing the application as an xSP, you may be required to provide the application in multiple locations around the world to guarantee such things as performance and availability. Communications networks are fast and increasing in speed every day, but for the most part applications and data that are maintained local to their users are faster.

*It is about service, not price.* Early xSPs competed on price. Most of them have not survived. Second and third generation xSPs have begun to compete on service – convenience, reliability, knowledgeable support staff, reliability. These xSPs have a chance to make it in the long run, provided they maintain their focus on service.

## 7.4 CHOOSING A SOFTWARE DEPLOYMENT ARCHITECTURE

Figure 7-1 brings the concepts of the previous sections together in an effort to help you choose a good deployment architecture by illustrating the rough relationships that exist between customer and corporate influences and deployment architectures. At the same time, it also captures the effects that one dimension often has on others. Using enterprise-class software as an example, if your enterprise-class customer has a high need for integration with existing or legacy systems, wants more control over the operations of the system, and is dealing with highly sensitive data, chances are good that they will favor a system that can be deployed at their site under the control of their MIS staff. Relaxing these constraints make other deployment choices viable, and in many cases even preferable.

In drawing Figure 7-1 I've intentionally simplified many of the variables that are associated with deployment architecture choices. For example, even if the customer has a high preference for deploying the system at their site, they may still choose an ASP/MSP deployment option if the skills of their operational staff are insufficient or if the application is projected to have peak demands that exceed the capabilities of an in-house deployment. The variables that are illustrated in blue – desired control, need for integration, and concern for data – all share the same effect on the choice of deployment architecture (when they are high, deploy at the customer site). The variables illustrated in red are on an opposite scale – when they are high, deploy as an ASP, MSP, or web service.

The upper right hand corner of Figure 7-1 captures some of the corporate influences described in the previous section. Figure 7-1 also captures the migrations that can occur between various deployment options as light gray lines.

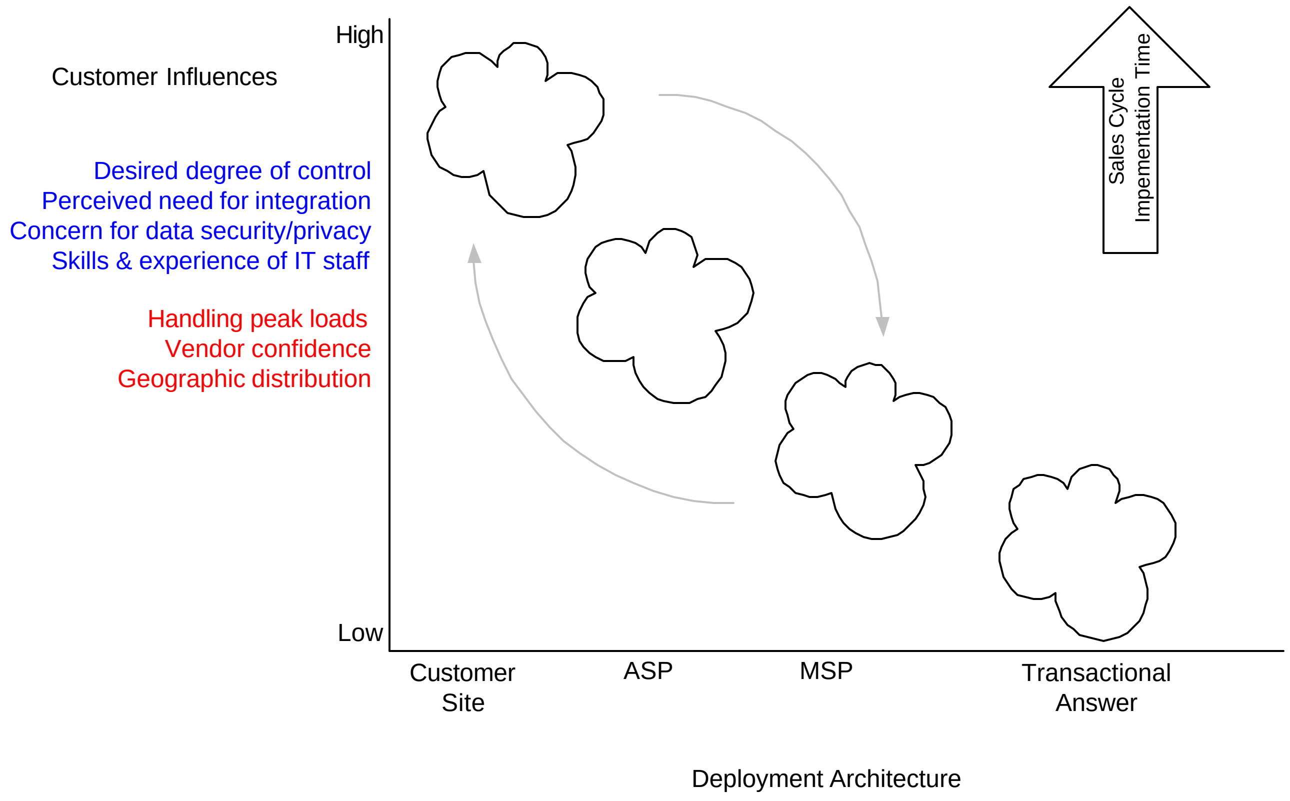


**Figure 7-1: Choosing a Deployment Architecture**

## 7.5 Deployment Architectures and the Distribution of Work

No matter what deployment architecture you choose, someone is going to have to install the system, maintain it, perform administrative functions, answer telephones when users have questions, and so forth. These basic support and service functions can't be ignored. In a customer-site deployment, or when integrating with a web service, the customer is responsible for the bulk of the total workload. In an ASP, these responsibilities are shared. In an MSP and transactional service, the service provider is responsible for these functions. What the various deployment choices do is shift the locus of control for performing these functions among various entities, as shown in Figure 7-2.

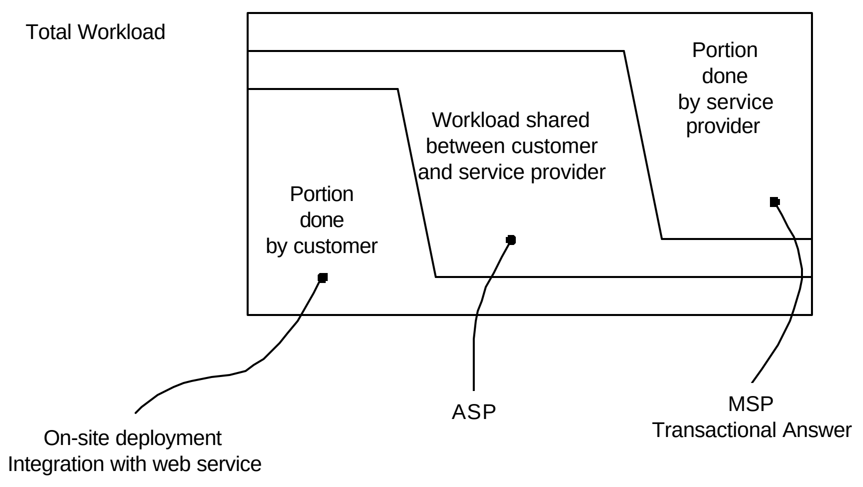

Total Workload
Portion done by service provider
Workload shared between customer and service provider
Portion done by customer
On-site deployment
Integration with web service
ASP
MSP
Transactional Answer

**Figure 7-2: Distribution of Work in Deployment Architectures**

## 7.6 THE INFORMATION APPLIANCE

A completely different kind of deployment architecture, and one that can be operated almost equally well at a customer site, an ASP, or an MSP, is the information appliance. I define an information appliance as a specialized device that provides one or more specific functions. These functions are usually designed to create a better total solution for a well defined target market. Examples of information appliances include:

- the Linksys EtherFast DSL Firewall / 4-port hub that I have installed in my home, which provides me with a simple firewall and 4-port LAN;
- Aladdin Knowledge Systems, Inc. eSafe appliance, which provides anti-virus and malicious mobile code protection for enterprises;
- the TiVo and Replay Digital Video Recorders, which provide a single device for managing television viewing.

There are a variety of trends that are motivating the growth of information appliances. One of the biggest trends is the continued adoption of the Linux operating system. Linux provides appliance vendors the assurance they need to create an appliance with a reliable operating system, free from expensive licensing models, that can be easily customized and embedded. While there are other excellent choices for appliance operating systems, most of them come with a license fee, which directly increases total cost in a way that provides little value to the user of the appliance.

One point in favor of proprietary operating systems are the tools they may have created to support a specific market niche. These tools may be better than similar tools available on Linux. In other words, there is no simple way to make the choice, as you will have to compare a full set of issues including license fees, development tools and related infrastructure, development team experience, and so forth. What is undeniable is that Linux is a strong platform of choice for a growing number of information appliance vendors.

Another important trend is the need to simplify the creation of complex solutions. Appliance vendors usually place a premium on simplicity. Just install the appliance and set a few simple parameters. Most appliances don't have or need things like keyboards, monitors, or even expansion slots. The absence of such items serves to simultaneously drive down costs while simplifying use.

Information appliances are not appropriate for every kind of software. Any system that creates or manages data is probably not a candidate for an information appliance. Any system that requires substantial customization or programmatic integration is probably not a candidate for an information appliance. And any system that is probably just better run on existing hardware is probably not a good candidate either. That said, the trends associated with the Linux operating system and other open source software projects will continue, as will our desire to reduce complexity for our customers. Thus, we will see the continued growth of the information appliance.

## 7.7 Deployment Choice Influences on Software Architecture

A given deployment choice may exhibit any of the following influences on your software architecture.

*Flexible, parameterized, or no integration options.* A system deployed at a customer's site experiences the greatest demand for flexible integration options. A system deployed as an ASP or MSP may experience demand for flexible integration, but the xSP vendor is not motivated to provide these capabilities because of the extremely high costs associated with flexible integration options. In fact, the more "standard" the xSP can make their offering, the better. Standardized offerings are simpler to create, simpler to manage, and more profitable to operate. Any deployment choice can be offered with no integration options, which is surprisingly appropriate in many circumstances in which you're creating a system with very well defined functions and relatively simplistic boundaries.

*Upgrade Policies.* Different deployment architectures make different demands on the people who upgrade them. Upgrades for systems deployed at a customer site must be carefully planned to be minimally disruptive to the operations of the customer. This can be especially challenging if the customer has crucially important data or has extensively integrated the production system via a large amount of customer programming. As a result, enterprise class systems deployed at a customer site are rarely upgraded more than once every nine months. In contrast, I know of one world-class MSP who safely modifies their production system approximately every 10-12 weeks, rapidly introducing new features to their customers in an emerging market. They are able to do this because early in the development of the system they made substantial changes to their application to make it easier to upgrade. Upgrades are discussed more thoroughly in chapter twelve.

*Data Protection and Access.* Application data must be maintained in a way that is generally appropriate based on the application, the users, and the sensitivity/importance of the data. When the system is deployed at a customer site the vendor can generally transfer all responsibility for handling these issues, especially as they relate to corporate data, to the customer. The converse, as mentioned earlier, is also true: Storing customer data at, or as, an xSP requires that the xSP or your entire engineering, product development and operations personnel follow strictly defined guidelines that deal with properly handling the data. Would you let your company store your confidential information?

*Migration options.* Although it is currently rare, I expect that there will be an increase in the number of solutions that can be deployed either as an ASP/MSP or on a customer-site. Furthermore, I suspect that these solutions will ultimately need to support migrations as previously described in this chapter. The possible effects on migration should be considered in the overall design of your architecture.

## 7.8 THE FUTURE OF CONSUMER SOFTWARE

Web services enthusiasts paint a picture of the future in which savvy, web-connected users won't actually license software for use on their home computer. Instead, they will connect to web services via a persistent, reliable, high speed Internet connection and use the software on a license the software for use on a rental or subscription basis. Fortunately, the deployment architectures presented above, which are currently most applicable to enterprise class software, will guide you in thinking about how you can offer your software in the brave new web services world.

Users, like enterprises, will make their deployment choices based on the quality of the *solution*, not the technology powering the deployment. For people like me, who rely on their laptop to get things done during international travel, the thought of only using software via an internet connection seems a bit crazy right now. Perhaps in the future, but certainly not now. However, for many other people, the idea of accessing software via a trusted vendor through the internet is very appealing. I'd like to see certain data, such as financial records, properly maintained for the rest of my life. Other data, such as my digital photos, *want* to be shared, making a web services model a natural fit. Because of these forces, and others that will emerge, I envision a similarly complex environment emerging for consumer software that currently exists for enterprise class software, creating very interesting choices and tradeoffs for marketects and tarchitects and the customers that they serve.

## Chapter Summary

- Your deployment architecture is the manner in which the system is deployed for use by a customer. Common choices include:
    - Customer site;
    - Application Service Provider (ASP);
    - Managed Services Provider (MSP);
    - other variant of a service provider (an xSP);
    - Web services.

  Hybrid models, which involve some aspect of the system being deployed at a customer site and other aspects of the system being installed at a service provider will become increasingly common.
- Customer influences that motivate the selection of a given deployment architecture include:
    - the degree of control the customer wishes to exert over the operation of the system;
    - the degree to which this application will or must be integrated with other systems;
    - concerns for data security / privacy;
    - the ability to handle "peak" loads;

    - the initial and ongoing costs associated with the deployment;
    - the confidence your customer has in your abilities to reliably provide the services they need;
    - the skills and experience of the operational staff required to manage the system.
- Corporate influences on the selection of a deployment architecture include:
    - the desired and actual sales cycle;
    - the infrastructure investment required to offer services;
    - the financial model associated with the architecture, most notably cash flows;
    - the degree to which you want to move quickly and efficiently in managing your customer base;
    - the geographic distribution of your company relative to your customers.
- The choice of a deployment architecture does not change the total distribution of work associated with a successfully managed system. It may change the distribution of this work.
- Information appliances are a growing category for deployment architectures in a wide variety of environments. Open Source licensing models, which can create lower total costs of ownership, are in part fueling this growth.

## CHECK THIS

- ❑ Our deployment model matches our target markets need for:
    - ❑ control
    - ❑ integration
    - ❑ data security / privacy

- ❑ We have created sufficient models of performance and can ensure that our deployment architecture can handle all anticipated workloads (see also chapter ten).

- ❑ We have instituted appropriate operational policies.

- ❑ We have accounted for the following in our choice of deployment architecture:
    - ❑ sales model and sales cycle
    - ❑ required infrastructure investment

- ❑ We have defined the specific amount of work that we expect the customer to perform.

## TRY THIS

1. Using Figure 7-3, identify how your software is deployed and the key forces that have motivated the choice of this deployment architecture.

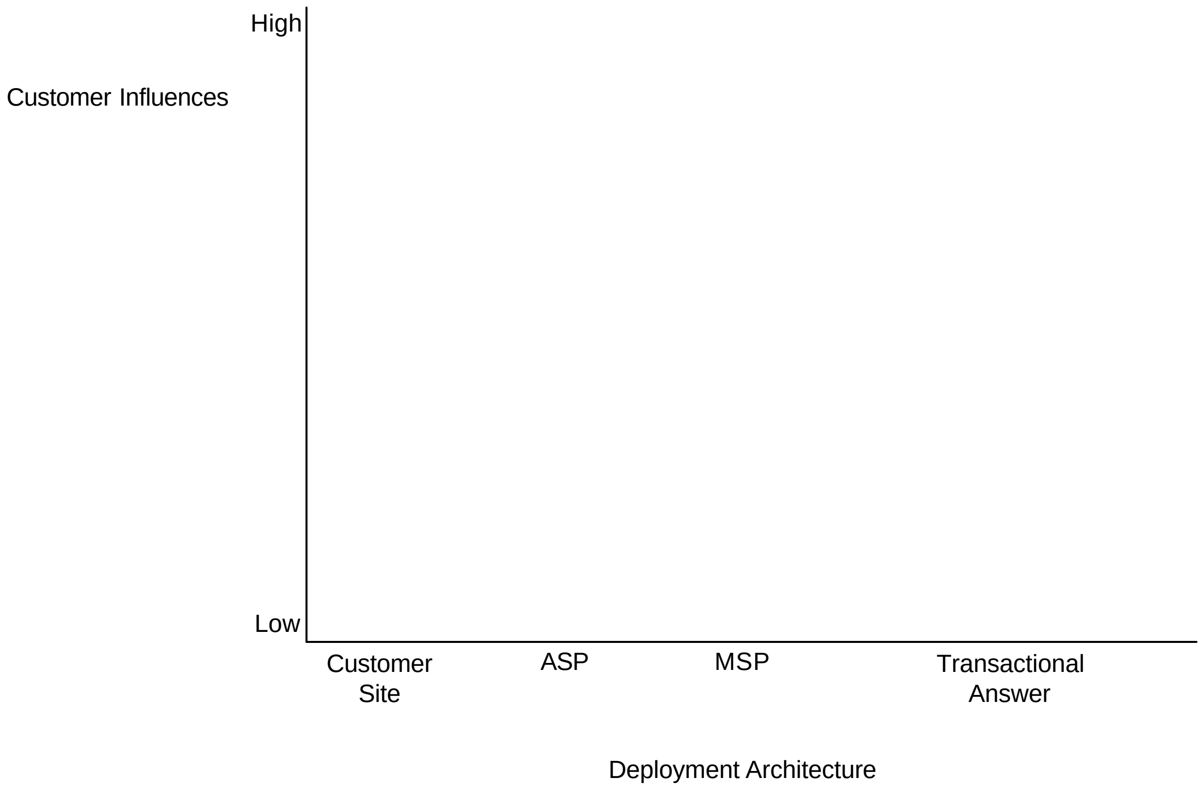


**Figure 7-3: Identifying How Your Software is Deployed**

2. What would you have to change to deploy your solution in a manner that is different than its current deployment? Would doing so enable you to expand your current market or allow you to reach a new one?

# 8. INTEGRATION & EXTENSION

As described in chapter 3, integration refers to the degree to which your system can or must be integrated with other systems, usually programmatically, in order to produce the expected product. Extension refers to the degree with which your system can be extended to produce an augmented product. In this chapter I will discuss the strongest motivations for creating architectures that can be integrated and extended with relative ease and the business ramifications of these choices.

## 8.1 MOTIVATIONS

There are similar motivations for integration and extension: in complex systems, both provide the ability to create a superior product. In some circumstances, integration and extension shift work from you to your customer. Paradoxically, this can increase customer satisfaction, much like our satisfaction with an ATM machine even though we're assuming responsibilities formerly performed by tellers. Part of the reason is that we're in control, and most of us like increasing amounts of control, even when it requires to assume or perform new duties. Some of the strongest motivations for creating systems that can be integrated and/or extended with other systems include the following.

- *You can't predict the future, but you can plan for it.* In today's complex environment it is impossible to predict precisely how your customer will want your system to behave in every given context. By providing customers with one or more ways to integrate or extend the system, you don't have to predict the future – but you do have to plan for it. Plug-in architectures, such as those used in Adobe Photoshop or Web browsers, allow components to be added to the system in well-defined ways, and are an excellent example how you can add functionality that enhances the operation of an existing system in a way that isn't predicted but is planned.

- *Customers hate to hear you say "We can't do that".* When you can't extend a system, you remove your ability to respond to customer needs. Ultimately, this means you'll have to eventually say "No" to a customer request. Not surprisingly, customers hate to hear "No". Providing a good strategy for integrating and/or extending your system will help you convert "No" to "Yes, we can do that. It might take a bit of work, but here is what we propose".

- *The larger solution is comprised of multiple, smaller solutions.* Many enterprise-class business systems are not designed to work in isolation. Instead, they work in conjunction with the overall supply chain to solve a larger problem. A web storefront or online ordering system is an example of a total architecture that cannot work without integration. In most implementations several systems must be integrated to produce the expected product, including such things as a catalog system, storefront engine, a content engine, a payment processing engine, and backend order tracking and accounting systems. If you've participated in the creation any of these kinds of systems you already know that creating an integrated architecture is essential to success.

- *You want to access information not contained within your system* . There are times when the only way to produce an interesting report or interesting analysis is to integrate the data contained in one system with the data that is contained within another. A common example of this occurs when you integrate clickstream data from web servers with customer purchase transactions.

- *You want to increase switching costs.* Switching costs refer to the "cost" your customer will pay should they stop using your system and start using another. Switching costs include a number of things, including the cost it would take your customer to re-integrate your solution into one offered by a competitor. Providing extension and integration options doesn't increase switching costs until they are used by your customer. Once your customer has performed some level of extension or integration, switching costs are dramatically increased, because your customer not only has to replace your system, they have to replace all of the integrations and extensions that they have created on top of your system. The more "proprietary" the extension and integration options, the greater the switching costs.

- *Creation of a product ecosystem.* Regardless of the success your company may have in creating, marketing, and selling your product, it is often the case that you can achieve greater success by encouraging other companies to participate in your success. I think of this process as creating a product ecosystem, in which various entities interoperate to achieve one or more goals, and in which you can work with others to achieve mutually beneficial goals. A classic example of an ecosystem is the set of companies that create plugins for Adobe Photoshop. These plugins extend the product in well-defined ways, usually providing specific functions that are required by niche markets. Key participants in this ecosystem benefit, including the Adobe, the company creating the plugin, and most importantly, the customer.

- *Common sense.* A good tarchitect will just about always add some form of API to their system to make it easier to test via a solid automated regression test framework. A good marketect will seek to leverage this technical investment – after all, who knows what the future may bring, and all that may be needed to add this feature to the product is some documentation.

**Some Assembly Required**

The Aladdin Privilege Software Commerce Platform (PSCP) includes a Storefront Engine that software publishers use to sell their goods directly to customers via a web site or indirectly through a multi-tier distribution channel that consists of any arbitrary number of distributors and resellers. Integration with other systems is required in either approach.

In the direct sales approach, the Storefront Engine must be integrated with credit-care payment and other transaction processing systems, catalog systems that provide product descriptions and price quotes, fraud detection systems, and so forth. In the indirect approach, the Storefront Engine must be integrated into a network of servers that are linked together to support transactions. In both cases, the system must be integrated with other systems in order to function properly.

Regardless of your motivation, it is essential that everyone associated with the development effort embrace the notion that an API represents a *commitment* to your customer. When a customer, partner, or other entity, such as a system integrator or Value Added Reseller (VAR) chooses to use your APIs they are tightly bound to your product and your company. The commitment is for the long haul.

## 8.2 LAYERED BUSINESS ARCHITECTURES: LOGICAL STRUCTURES

One of the most common system architectures for business applications is the layered architecture, in which subsystems are logically and physically arranged in a series of layers (refer to Figure 8-1). Such architectures, which form the foundation for such frameworks as J2EE, provide an excellent case study when considering approaches to integration and extensibility. In this section I'll briefly review the principles associated layered architecture. In the next I'll discuss a variety of approaches to extending the architecture.

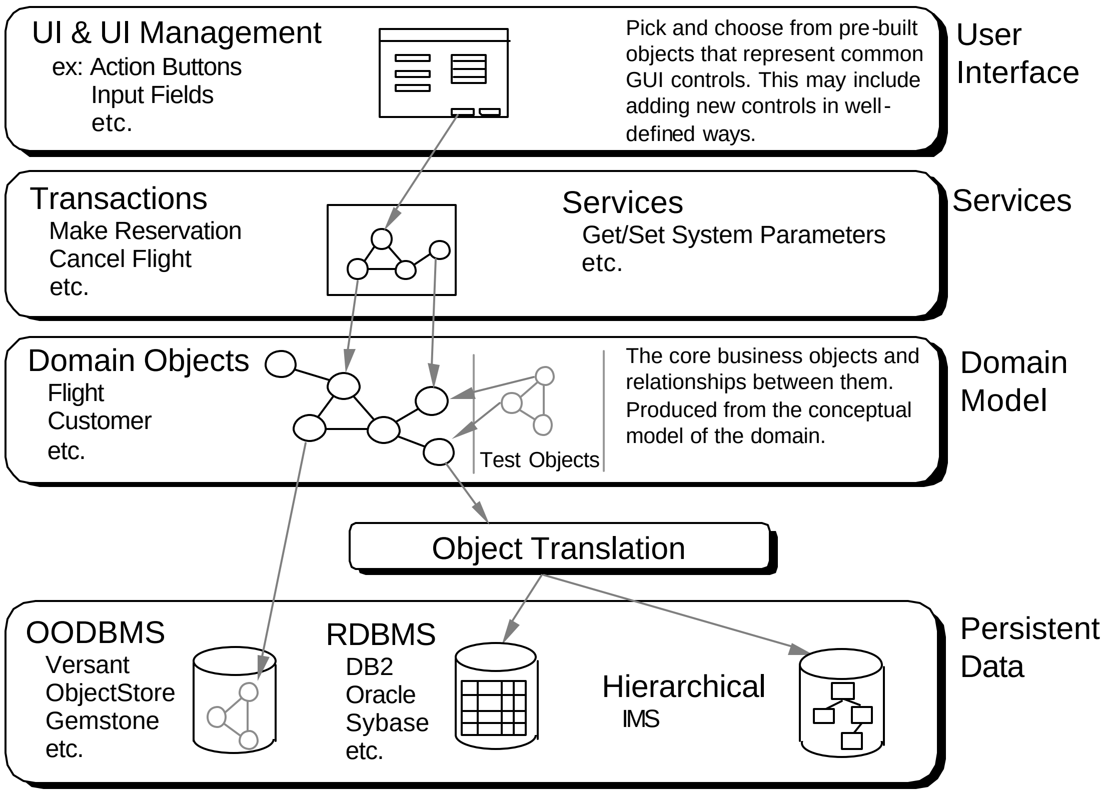


**Figure 8-1: Layered System Architectures**

### THE USER INTERFACE LAYER

The *user interface* layer is responsible for presenting information to the user and managing the users' interactions with this information. It is common for the user interface to be graphical, either as a "heavy" client that runs as an application, or as a "thin" or "light" client that runs in a browser. Other forms of user interfaces are increasingly common, including voice and hand-held user interfaces. The user interface layer is often the layer that must shoulder the bulk of the work when dealing with internationalized applications. This is always more difficult than it seems, and some of the most vicious user interface problems teams I've worked with had to resolve dealt with internationalization.

The most essential criteria to keep in mind when constructing the user interface is to make certain that it does not contain any application or business logic. This kind of logic belongs in other layers. I have found that a good way of thinking about where to place application logic is to ask yourself the following question: "What parts of my system would have to change if I replaced the current user interface with an entirely new type of interface, such as an automated voice response system?" If the answer indicates that you would have to change substantial portions of your application code (e.g., code that performs edit or validation checking), chances are good that the user interface layer contains logic that should be more appropriately placed in the transaction management or domain layers.

Many enterprise applications split the user interface into two layers. One layers deals with the actual presentation of information to the user. The other mediates the "micro workflow" that may exist between a given kind of user interface and the services layer. For example, suppose you're creating an application to manage flight reservations. If the specific user interface is a browser, you may be able to acquire all of the necessary data to request a reservation in one screen. If the specific user interface is a phone, you may need to coordinate multiple dialogs in order to collect all of the necessary data. Because these operations are based on a specific kind of user interface, they belong in the user interface layer. The "micro workflow" layer may also be responsible for reformatting any domain specific data for the specific user interface.

I strongly recommend the creation of a simple "command line" interface. Such an interface is easy and fast for developers to use, makes many forms of automated testing easier, and is trivially scriptable using languages such a Tcl or Perl. It can also be easily implemented on top of the domain model via simple API.

## THE SERVICES LAYER

The *services* layer provides various services defined by your application to the user interface and other applications. These services may be simple, such as obtaining the current system date and time, or complex, such as changing a flight or canceling a reservation in an airline reservation system. Complex services are often implemented as transactions, with complete transactional semantics (e.g., rollback).

There is often a close correlation between services and use cases. At times, a use case may be represented as a single service. At other times, individual steps within a use case may represent a service. CRUD operations (Create, Reference, Update, and Delete) are often represented as services.

## The Domain Model Layer

The *domain model* or *domain layer* represents the fundamental business concepts and rules of your application domain. I consider the domain model an optional layer in enterprise applications, only required when business rules are too complex to represent in simple services, or when object structures are more efficiently represented by in-memory representations of objects.

When the need for the domain model arises, the domain model often emerges as the "core" of the application. Instead of thinking of your architecture in layers, it is often useful to think of it as an "onion". The center of the application is the domain model. Other layers are built, or grown, depending on your development method, around this core. It needs to be correct.

I admit this visualization suffers a bit because it doesn't provide a good way to think about persistent data. More dangerously, it could imply that the domain model is more important than the persistent data model. In most applications this is untrue, and these two elements of your tarchitecture are equally important. However, it reinforces that the domain model is the core of a good application.

Decoupling the domain layer from the user interface and transaction management layers provides for substantial flexibility in the development of the system. We could replace the screen presented to a service agent with an interactive voice response system or a web page without changing the underlying application logic (provided they have reasonable interfaces and appropriate service objects to make this replacement – more on this later). It also contributes to cohesion: each layer of the architecture is responsible for a specific set of related operations.

I do not mean to imply that the domain model *must* be constructed before the user interface is designed. While it is often helpful to design the user interface after a preliminary domain model has been designed, I have worked on several successful projects in which the user interface was prototyped using paper-and-pencil ("lo-fidelity") techniques before the development of the domain model. Once the user model of the system was validated the development of the domain model was relatively straightforward. During implementation the domain model was implemented first, and the user interface followed quickly thereafter based on the previously agreed upon public interface provided by the domain model.

## THE PERSISTENT DATA LAYER

Most business applications rely on a database management systems to manage the persistent storage of objects. The most common approach in enterprise applications is to use a relational database, and to create a separate layer to manage the mapping between objects within the domain and objects within the relational database.

This is not as easy as it may sound. Complex object structures, objects comprised of data obtained from multiple sources, objects with complex security restrictions, such as operations that can only be performed by certain classes of users, or transactions that involve large numbers of objects all contribute to the challenge of efficiently mapping domain objects to relational databases.

Because of these reasons, I’ve found that there are times when it makes sense to structure the domain model so that it can work easily and efficiently with the underlying database schema. Indeed, if the schema is unusually complex, or if the performance requirements are particularly severe, it may make sense to forego the domain model entirely and simply connect the services layer to the database schema. This may seem counter-intuitive, especially if you’ve been trained in many object-based design methods. However, the reality of many enterprise applications is that creating a rich domain model and then relying on an object-to-relational mapping to store and retrieve these objects just isn’t worth it. A better approach is to define an appropriate set of services that connect to the database through SQL statements and the occasional stored procedure and/or trigger.

## VARIATIONS ON A THEME

As a “generic” architecture, Figure 1 provides a reasonable starting point for the design of loosely coupled, highly cohesive systems with the flexibility necessary to effectively handle complex problems. Of course, this figure is a simplification and abstraction of real systems: distributed computing, legacy systems integration, and/or structural relationships or choices made to support specialized databases or hardware can all change the picture.

## 8.3 CREATING LAYERED BUSINESS ARCHITECTURES

Successful projects rarely concentrate on building all layers of the architecture at the same time. Most of them build the application through a process I refer to as “spiking the architecture”. A *spike* is a user-visible piece of functionality (usually described by a use case or an XP-style story) that has been completely implemented through all appropriate subsystems or layers of the tarchitecture. It is suitable for alpha (and sometimes) beta testing and is stable enough for the end-user documentation team to begin online help and related aspects of end-user documentation. The key attribute of a spike is that it "drives" some specific aspect of functionality through all relevant subsystems (I use the term "nail" to represent a connection made between two subsystems).

Spikes are a way of thinking about incremental development. I'm not the first person to advocate this approach, and I'm certain I won't be the last. That said, I've found that my choice of language provides a good metaphor for the work of the team. Imagine, if you will, that a single subsystem is a 4"x6" board. Your architecture can then be thought of as a series of 4"x6" boards stacked one on top of the other. A nail connects two boards together -- but a spike drives through all of the boards. One of my development managers recently pointed out that "spiking the architecture" is both a process and an event. It is a process because driving the spike through many layers can be hard work, and it's new to many development teams. When finished, it is celebrated as an event.

While the first spike proves the basics of the architecture and is the foundation for sound risk management practices, there is still room to adjust the boards to make certain the system is lining up as intended. As more spikes are added, the system becomes more stable. The spikes also happen more quickly, as the team becomes skilled in driving the spikes through all of the layers. The overall organization also benefits, as the specific functionality of the system is developed over time. If you've had some trouble with adopting an iterative development process, consider spiking your architecture.

I’ve found that even teams who have adopted a “spike the architecture” or other iterative development approach can have trouble on determining where to begin the development efforts. If you get stuck, try starting with either the domain or the user interface, as shown in Figure 2.

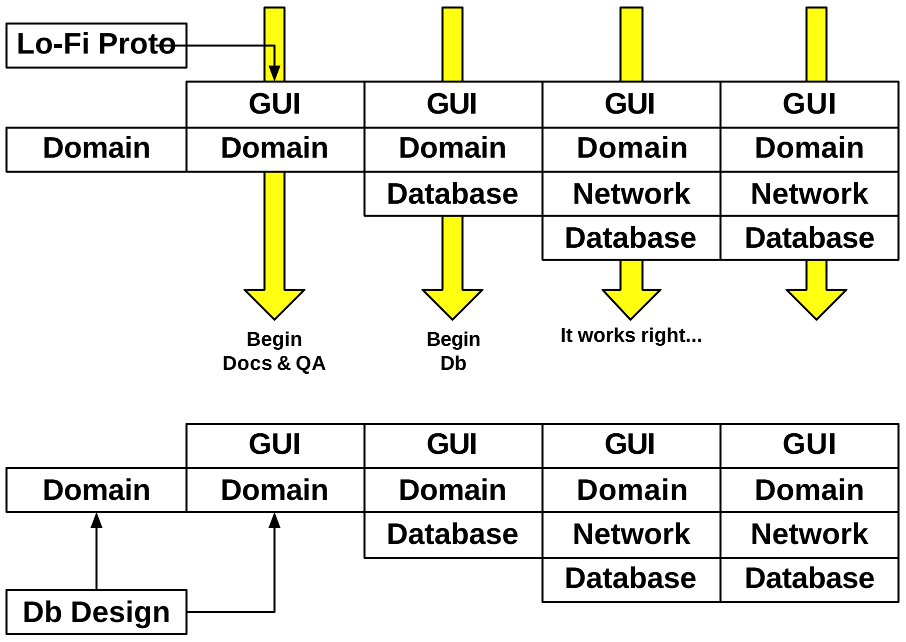


**Figure 8-2: Spiking The Architecture**

Figure 2 illustrates the approach of starting with the domain, preferably with a lo-fidelity user interface prototype to set expectations or the design of the persistent storage layer. Getting the domain right makes it is far easier to get the rest of the tarchitecture right. Getting the domain wrong puts everything at serious risk. Eventually, it will have to be fixed. Agile modeling and agile development methods are particularly useful when trying to build a good domain model, especially if the application or product is new. After the domain has been validated, projects should create, extend, or otherwise handle other layers in a predictable manner. An application that relies heavily on legacy data may proceed to work on the database layer. Another might instead choose to concentrate on the user interface.

If there are sufficient resources, multiple layers of the architecture can be addressed in parallel. The key to this approach is to establish the core subsystems that comprise the architecture early. This, in turn, is best done accomplished by agreeing upon subsystem interfaces early in the development of the project and using daily or weekly builds to ensure that subsystems remain in sync.

If you are going to attempt parallel development make certain the developers associated with each layer agree that the domain guides the development of the other layers. For example, suppose that the user interface development team discovers through usability testing that an important attribute was missed during analysis. Rather than adding this attribute (and its associated semantics) only to the user interface, the user interface development team should negotiate with the domain team to add this attribute (and its associated semantics) to the appropriate domain objects. The domain team, should, in turn, negotiate with the database team to ensure the appropriate changes are made to the persistent storage model.

Unfortunately, there can be unpleasant delays in waiting for all of the layers to catch up with each other. I have found that a two-stage process works best. First, the domain layer makes the necessary changes and simulates the database model through an "in-memory" database. This allows the user interface team to proceed and allows the domain team to verify the changes to the application. While this is happening the database team is hard at work making their changes. When finished, the domain is connected to the database layer, and the entire application works. No matter what, you should not delay "spiking" the architectures as often as possible, for it is a proven technique for reducing risk.

## 8.4 Integration and Extension At The Business Logic Layers

There are a wide range of technical approaches that you can use to create systems that can be integrated and/or extended. This section draws from my experience creating the necessary architectural infrastructure for providing integration and/or extensibility. These techniques are easily extended to other kinds of application architectures. Refer to the following diagram as you read this section and the next.

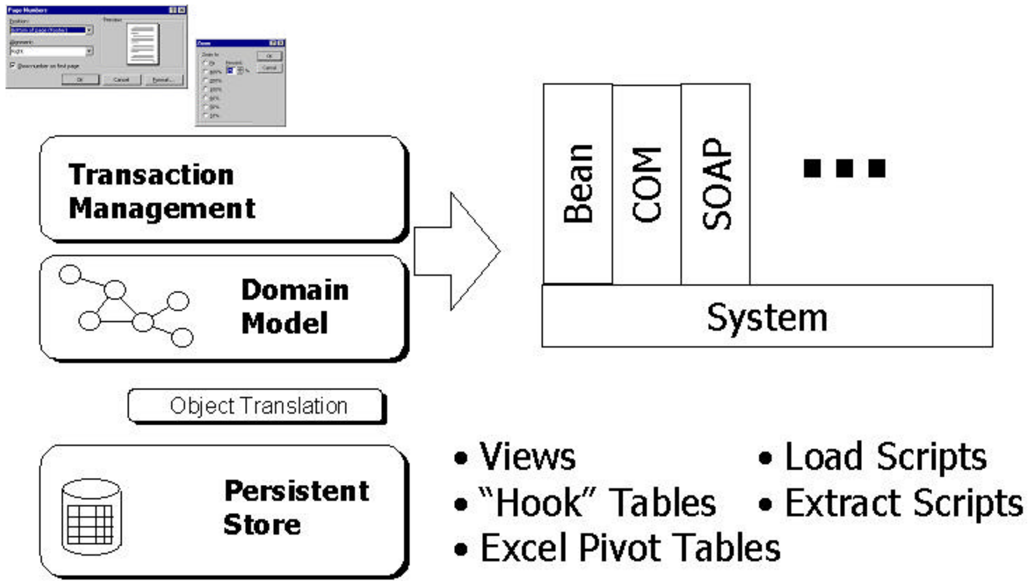


**Figure 8-3: Integration and Extension Techniques**

### 8.4.1 INTEGRATION THROUGH APIS

Application Programming Interfaces (APIs), which expose one or more aspects of functionality to developers of other systems, are based on the model that the other developer is creating an application that is the primary locus of control. Your system becomes a set of services that are accessed through any number of means that may be appropriate, such as a library, such as C or C++ library; a component such as a COM, JavaBean, or Enterprise Java Bean (EJB); a web service; and/or a message-oriented interface. The use of APIs is primarily associated with the desire to integrate your application into some other application.

If you've constructed your application according to the principles espoused earlier, it is usually a trivial manner to create APIs that allow customers access to the functionality of your system. The most direct approach is to start with exposing the services layer, possibly exposing other layers as required. This approach is far superior to allowing your customer direct access to your database, as direct access to the database bypasses all of the business logic associated with your application. Consider the following items as you create these APIs:

- *Platform preferences*. While C-based interfaces may be the universal interface of choice, every platform has some variant that works best. For example, if you're creating an application under MS-Windows, you will almost certainly want to provide a COM based interface. Other customers may require a J2EE-inspired approach and demand EJBs that can be integrated into their application.

- *Market segment preferences*. In addition to the pressures exerted on an API by the platform, different market segments may also exhibit preferences for one approach or another. If you're working with innovators, chances are good that they'll want to use the most innovative approaches for integrating and/or extending your systems. As of the writing of this book, this means providing web services. In the future, who knows? Other market segments may request different approaches. You may need to support all of them.

- *Partner preferences*. In chapter two I discussed the importance of the context diagram is identifying possible partners to creating an augmented product. Your partners will have their own set of preferences for integration. Gaining an understanding of partner preferences will help you create the integration and extension approaches that are most likely to gain their favor.

- *Establish naming conventions.* If you've ever been frustrated by a vendor providing a nonsensical API, then you should have all of the motivation you need to work at creating a sensibly named, sensibly structured API. Try to remember that making it *easy* to use your APIs is in your best interests.

- *Security & session data.* Many business applications manage "stateful" data, often based on the concept of a "user" who is logged in and working with the application (a "session"). State or session data may be maintained in any number of ways. For example, in web applications a session is often managed by embedding a session identifier in each transaction posted to the server once the user has logged in, either as part of the URL or as part of the data in the web page. Alternatively, if you're using a "heavy" client (e.g., a MS-Windows client application) you can manage state or session data directly in the client or in a way that is shared between the client and the server.

  Developing APIs for applications that rely on session data is harder than developing APIs for applications that don't. You have to find a way to provide your customers with a means to specify and manage session-related parameters, such as timeouts and security protocols. Depending on how you're application works with other applications, you may need to provide a way to identify another application through a userid so that the rights associated with the application can be managed. You may also need to provide facilities for managing session data as functions are invoked. For example, if your server manages session data through an opaque session identifier, you have to make certain you tell your customers not to modify this handle.

**Timing is Everything**

Your documentation (discussed later) must make certain session management semantics are clearly presented to your customer. I remember helping a customer track down one particularly troublesome bug that wasn't a bug at all. The application was a multi-user, web-based application. Each logged in user represented a session. Because sessions consumed valuable server resources and because each session counted against one concurrent user, we associated a session timeout parameter that could be set to forcibly log off a user in cases of inactivity or in case of an error.

The underlying application allowed system administrators to associate various kinds of application functionality with specific userids. Thus, "userid-1" could perform a different set of operations than "userid-2". Our API was constructed to follow this convention. Specifically, applications wishing to use the functions provided by the server were forced to log into the server just like a "human" user. This enabled the system administrator to control the specific set of functions that an external application was allowed to use.

Our customer had set their session timeout parameter to expire sessions after 20 minutes of inactivity – but then wrote an application that expected a session to last an arbitrarily long amount of time! Since normal users and "programmatic" users were accessing the same transaction model, both were held to the same session management restrictions. To solve the immediate problem the customer modified their application to re-login if their session expired. They were unhappy with this solution, and we eventually accommodated their needs by allowing session timeout data to be set on a per-user basis. The customer then modified the session timeout parameter associated with the userid representing their application, and everything worked just as they wished.

- *Expose only what your customers need.* You should be careful about liberally exposing all of the APIs that may exist within your system. Every API that you expose increases the total work in creating and sustaining your product. It is also unlikely that every function exposed by your API provides the same value to your customer; not everything is needed. In a complex system, different operations exhibit different performance profiles and make different demands on underlying system resources. Some APIs may invoke complex functions that take a long time to execute and should only be used with proper training. You don't want your customer to accidentally bring the system to a grinding halt through improper use of the APIs.

  Tarchitects should work carefully with your marketect to make certain that they're defining the smallest set of APIs that makes sense for the target market. Because there may be many constituents to satisfy, consider defining multiple sets of APIs, one (or more) for internal use, and one for integration/extension, possibly governed through security access controls. I know of one large application development team that created *three* APIs: One for internal use within the team, one for use by other application development teams within the same company, and a third for external customers. The APIs differed in functionality and performance based on such things as error checking and calling semantics.

- *APIs tend to stabilize over multiple releases.* It can be quite difficult to predict the specific set of APIs that are needed in the first few releases of a product, and it is rare that you get it right the first time. Be forewarned that it may take several releases to stabilize the right set of APIs for a given target market.

- *Be clear on what is not allowed.* Your application may provide for certain features that are simply not available in the API. Stating what can – and *cannot* – be done with an API makes it considerably easier for a customer or system integration consultant to determine the best way to approach a complex problem.

### 8.4.2 EXTENSION THROUGH REGISTRATION

Integration based approaches allow your applications to be included in other systems or applications. Registration is a process whereby the capabilities of your application are extended in one or more ways by registering a component with your system. A great example of this is a web browser, which can be extended through a plugin architecture.

Registration-based approaches allow an external component to register itself in some way with your application. Then, according to some pre-defined criteria, your application invokes this component. Examples of registration based APIs include callbacks, listeners, some forms of plugins, and event notification mechanisms. Consider the following when providing for registration based solutions.

- *Define the registration model.* Provide developers with detailed technical information on such things as the language(s) that can be used to create registerable components, when and how these components are registered, how to update registered components, and so forth. To illustrate, some applications require that plugins be located in a specific directory and follow a platform-specific binary model. Other applications require that all components register themselves through a configuration file. Some applications allow you to change registered components while the application is running while others require you to restart the application to acquire new or updated components. You have a wide variety of choices – just make certain you're clear on those that you have chosen.

- *Define the event model.* The specific events available to the developer, when they occur, the format of the notification mechanism, and the information provided in a callback must all be made clearly available to developers.

- *Define execution control semantics.* Execution control semantics refers to such things as blocking or non-blocking calls, thread and/or process management, and any important timing requirements associated with external components. Some applications, for example, transfer control to a plugin to process a request within the same process space. Others invoke a separate process, then hand control to the plugin. Other applications invoke the registered component as a simple function and block, waiting the results.

- *Define resource management policies.* All of the decisions associated with resource management, from provisioning to management and recovery must be defined. Consider all forms of resources that may affect your application or be required for a successful integration, including, but not limited to, memory, file handles, processing power, bandwidth, and so forth.

- *Define error/exception protocols.* Errors and exceptions in one application must often be propagated through your API to another application. You may also have to define conversion semantics, as what might be considered an "error" in one application may be considered an "exception" by another.

## 8.5 INTEGRATION AND EXTENSION OF PERSISTENT DATA

In the vast majority of cases, you don't want your customer directly accessing or changing your schema or the data it contains. It is simply too risky. Direct access bypasses the validation rules that usually lie within the domain layer, causing corrupt or inaccurate data. Transaction processing rules that require the simultaneous update or coordination of multiple systems are only likely to be executed if they are invoked through the proper services. Upgrade scripts that work just fine in QA can break at a customers site if the schema has been modified.

All of that said, there are times when a customer wants access to persistent data – and there are times you want to let them. They may want to write their own reports or extract data for use in other systems, and it may be just easier and faster to let them do this on their own without your intervention. They may need to extend the system with additional data that meets their needs rather than wait for a release that may be months away. Of course, your marketect may also push to allow customers direct access to persistent data, as this makes switching costs exorbitantly high.

Since chances are good that you're going to have to provide some access to your persistent data, here are some techniques that should help.

**Putting Business Logic Into the Database**

One of the most generally accepted principles of enterprise application architecture is to out business logic within the services and/or domain model layers of the architecture. While this is a good principle, it is not an absolute rule that should be dogmatically followed. There are many times when a well-defined stored procedure or database trigger is a far simpler and substantially more efficient solution than placing the equivalent logic in the services and/or domain layers. Moreover, you may find that they only way to safely offer integration options to your customer is to put certain kinds of logic in the database: I would rather rely on the database to invoke a stored procedure than rely on a customer to remember to call the right API!

### 8.5.1 VIEWS

Providing a layer of indirection between different components is one of the broadest and most time tested principles of good design. By reducing coupling, well placed layers of indirection enhance flexibility.

Views are logical databases constructed on top of physical schemas. They provide a layer of indirection between the use of data in a schema and how it is actually stored. The value of a view is that it provides you with some flexibility in changing the underlying schema without breaking applications or components that rely on a specific implementation of the schema. They are useful in a variety of situations, such as when you want to provide your customer with a schema optimized for reporting purposes. Thus, the first, and many times, most important approach to providing access to persistent data: Always provide access to persistent data through a view.

### 8.5.2 Hook Tables

Suppose you have created an inventory tracking and warehouse management system. Each operation on an item, such as adding it to the inventory and storing it in the warehouse, is represented by a discrete transactions. Your customer, while pleased with the core functionality of the application, has defined additional data that they would like associated with certain transactions. Your customer would like to store these additional data within the same database as used by your application to simplify various kinds of operational tasks, such as backing up the system. Like every other request, your customer wants these data added to the system *now*. They don't want to wait until the next release!

Hook tables are one way to solve this problem. Hook tables provide your customer with a way of extending the persistent storage mechanism that can be preserved over upgrades. The proper use of hook tables requires coordination among multiple layers in your architecture, so be careful with this powerful approach.

Begin the process of creating hook tables by identifying those aspects of the schema that the customer wishes to extend. Next, identify the events that are associated with the most basic operations that modify these data: create, update, and delete. Include operations that are initiated or handled by any layer in your architecture, including stored procedures. Take care, as you need to identify *every* such operation.

Now, create the hook tables. A hook table is a table whose primary key is equivalent to the primary key of a table in your current schema and that has been designed to allow customers to add new columns. This primary key should be generated by your application using a GUID or an MD5 hash of a GUID. Avoid using auto-increment fields, such as automatically incrementing integers, for this key, as such fields make upgrading databases very difficult.

Create, update, and delete modifications to the primary table are captured as events and a registration or plugin architecture is created to allow customers to write code that responds to these events. Thus, when some action is taken that results in a new record being added to your database, some kind of notification event is received by customer code of the change. Upon receiving this notification, your customer can perform whatever processing they deem is appropriate, creating, updating, and/or deleting the data that they have added to the schema under their control.

In general, event notification comes after all of the work conducted by your application. A typical creation sequence using a plugin architecture is that your application performs all of the work necessary to create a record and performs the insertion in both the main application table (with all of its associated data) and the hook table. The hook table insertion is quite easy, as all it contains is the primary key of the main table. The list of plugins associated with the hook table is called passing the newly created primary key as a parameter.

More sophisticated structures allow pre- and post-processing transactions to be associated with the hook table to an arbitrary depth. Pre-processing can be important when a customer wishes to perform some work on data that is about to be deleted.

Hook tables are not designed to solve every possible situation that you may encounter, mostly because they have a number of limitations. Relational integrity can be difficult to enforce, especially if your customer extends the hook table in surprising ways. Because hook tables might also introduce unacceptable delays in transaction processing systems they should be kept small. You also have to modify your upgrade scripts so that they are aware of the possible presence of hook tables.

### 8.5.3 SPREADSHEET PIVOT TABLES

Quite often a customer makes a request for dynamic reporting capabilities that may be difficult to support in your current architecture. Before going to a lot of work trying to create such capabilities in your architecture, check to see if your customer is using or has access to any of the powerful spreadsheet analysis programs that provide for interactive data analysis. I've had good results with Microsoft Excel, so I'll use it as my example.

Excel provides a feature called a pivot table that allows for hierarchically structured data to be presented in such a way that users can dynamically manipulate. Using a pivot table, users can quickly sort, summarize, subtotal, and/or otherwise "play" with the data. Data can be arranged in a variety of formats. My experience is that once users are introduced to pivot tables they quickly learn to manipulate them according to their own needs.

It is important to remember that pivot tables are based on extracts of the data contained within your application. Once the data has been exported it is no longer under the controls provided by your application. Security, privacy, and accuracy are just some of the concerns that using pivot tables create. Pivot tables are often associated with the next form of integration and extension of persistent data.

### 8.5.4 Extract, Transform, and Load (ETL) Scripts

When managing persistent storage, extract, transform, and load (ETL) scripts refer to a variety of utility programs that are designed to make it easier to manipulate data. Extract scripts read data from one or more data sources, extracting a desired subset of data, and storing these data in a suitable intermediate format. Transform scripts apply one or more transformations on the extracted data, doing everything from converting these data to a standard format to combining these data with other data to produce new results. Finally, load scripts take the result of the transform scripts and write the data to a target database.

If you're application is in use long enough, chances are good that sometime during its use customers are going to want to extract and/or load data directly to the schema, bypassing the domain layer. There are several motivations for doing this, including the fact that the programmatic model provided by the API is likely to be too inefficient to manage transformations on large data sets. A special case of ETL scripts concerns upgrades, when you need to migrate from one version of the persistent storage model to another. Although it may be tempting to let your customers do this work, there are distinct tarchitectural and marketectural advantages to providing

From a tarchitectural perspective, providing specifically tuned ETL scripts helps ensure that your customers obtain the right data, that transformations are performed in an appropriate manner, and that load operations don't break and/or violate the structure of the existing schema. An alert marketect can also take advantage of productized ETL scripts. As part of the released product, these scripts will be tested and documented, considerably enhancing the value of the overall product.

**Charging For ETL Scripts**

Recall from chapter two that the key to effective pricing is to relate the pricing to the value perceived by your customer. Pre-packaged ETL scripts are an excellent example of this. In one application we created several ETL scripts to help our customers write customized reports. I estimate that these scripts, which were fully tested and documented, cost us about $50K to create. We were able to charge about $25K for these scripts, which quickly became one of our most profitable “modules”. Another advantage to this approach was that these scripts further tied the customer to our application.

### 8.5.5 Tell Them What is Going On

Even though I generally recommend against providing your customers with direct access to your schema, I also recommend that you provide your customers with complete information about your schema so that they can gain a proper understanding of your systems operation. This means providing them with properly created technical publications that detail such things as the data dictionary, data, table, and naming conventions, important semantics regarding the values of any special columns, and so forth. such documents become invaluable, must-have references that usually motivate customers to work within the specific guidelines you have established for system integration. The lack of usable, accurate technical documentation is the real source of many integration problems.

## 8.6 Business Ramifications

Providing ways for the system to be integrated will have a number of business ramifications for both the marketect and the tarchitect. Handle all of them and you increase your total chances for success. Miss any of them and you may seriously cripple your product’s chance for success.

### 8.6.1 Professional Services

The various technical options available for integrating a system can create a bewildering array of choices for a customer. Too often, they are left unsure of the best way to integrate a system within their environment or with the fear that the integration process will take too long, cost too much, and ultimately result in failure. Some of these fears are justified, as many large-scale integration projects fail to realize key corporate objectives. To address these issues, corporation should establish a professional services organization to help customers achieve their integration goals, to answer questions the customers may have about the system, and to reduce the amount of time necessary to integrate a system within the customers existing environment.

The marketect should assist in the creation of the professional services, helping to outline their offerings and pricing models. The degree of control can vary. I’ve worked in organizations where my product managers had complete responsibility for establishing the professional services offerings. In other companies, professional services was responsible for establishing prices and creating some offerings, in close collaboration with product management. My strong preference is that product management be given responsibility for setting the standard service offerings because this forces them to understand the needs of their target market. The most essential work of the marketect in an organization that has professional services is the role they play in mediating requests from professional services to development.

The marketect also plays a key role in determining how professional services are organized. The two basic choices are in-house professional services or a partnership with an external firm. In practice, these choices are mixed according to the size of the company and the complexity of the integration. Smaller firms have small professional services organizations and rely heavily on partners; larger firms can afford to bring more of this in-house. This is a decision of strategic importance, and the marketect should provide data that helps senior executives make the best decision. My own experience indicates that smaller companies do a very poor job of enlisting the aide of larger consulting partners, to their detriment.

Development (or engineering) plays a strong, but indirect, role in assisting customers. Instead of working directly with the customer, development must work instead with professional services, making certain they understand the product and the capabilities of the technical approach. A key service that development can provide is the creation of sample programs that show how APIs are used. I've found it especially helpful if one or more members of the development team, including the tarchitect, join services to help them create some of the initial programs. The feedback on the structure, utility, and usability of the API is invaluable, and both services and developers welcome the opportunity to work on "real customer" problems instead of artificial examples.

The tarchitect plays a special role in assisting the marketect select external professional services partners. Like any other developer, the tarchitect should assess the skills of the external agency, examine their portfolio of successful projects, and interview key customers to assess their overall satisfaction with the quality of work of the potential partner.

### 8.6.2 Training Programs

Training programs are required for the successful use of just about any system. All software requires a common base of training programs and materials help ensure that its users are obtaining the greatest value from each of their normal application tools. Such training is increasingly required as programs increase in sophistication. Training is also vital to make certain users have a favorable reaction to a product.

The role of marketect in this kind of training is to coordinate the efforts of technical publications, quality assurance, and development to ensure that these materials are created and are technically correct. A tarchitect committed to usability can play a surprisingly influential role in this process, ensuring that the training materials are not only accurate, but truly represent capture and convey "best practices".

**How Do I Add A Document?**

One of the most uniquely frustrating experiences I've ever had concerns my experience in working for a company that used Lotus Notes. When I joined the company, I received a total of two minutes and 57 seconds of training in how to use Notes. Specifically, I was taught how to login, read, and send an email.

Words cannot express my frustration over the next several months as other people in the organization used Lotus Notes in a variety of ways that seemed mystical to me. While other people were successfully using the collaboration and file management tools inherent in Notes, I was simply struggling to manage my calendar. It took me several months of frustrating trial and error to achieve a modicum of skill with Lotus Notes.

You might wonder why I didn't use the built-in help and/or tutorial materials to learn more about the system. I tried this, but to no avail. In my opinion, the built-in help and tutorials were useless. At this stage I've been left with a very poor impression of Lotus Notes, which is sad, because the problems described above are easily corrected by a motivated marketect working in partnership with a similarly motivated tarchitect.

The base level of training provided to end users must be substantially augmented for systems designed to be extended and/or integrated with other systems. Returning to the Adobe Photoshop example referenced earlier, training programs must be created that clearly detail how a developer is to write a plugin. Training programs for enterprise class systems can be surprisingly extensive, including several days of introductory to advanced training in dedicated classrooms.

When designing the full suite of training solutions for their target market, the marketect must take into account all of the stakeholders associated with the system. In one enterprise class system I worked on, we designed training programs for:

- developers who wanted to integrate the system with other systems;
- system administrators who wanted to ensure the system was configured for optimal performance;
- partners who were integrating their software into our software;
- consulting partners who used our system to provide enhanced consulting services to our mutual clients.

This list is not exhaustive, and can include programs designed to service the needs of different constituents depending on your specific system. What is important is to recognize that complex systems rarely stand alone, that many different people within a company "touch" these systems, and that their job performance can be improved through the right training.

I've found that these kinds of training materials suffer from a number of key flaws, several of which can be mitigated or even removed by involving the tarchitect. One is that APIs and example programs are often incorrect: They examples don't compile, they fail to represent best practices, and they fail to expose the full set of features associated with the product. This last point is an especially serious mistake, because it exposes your product to serious competitive threats. A slightly less serious, but related, flaw, occurs when the provided examples fail to provide sufficient context for those developers using the APIs to build a good mental model of the system and how it operates. The result is that it takes developers far longer to effectively use the APIs of your system.

### 8.6.3 Certification

The next logical step beyond training is certification. This is a rare step, and one that is only required for systems that command extremely large market shares. This step is almost exclusively the domain of the marketect, and in deciding whether or not to create a certification program I recommend considering the following factors.

- *Product ecosystem.* Certification makes sense in product ecosystems characterized by both a large number of third parties who provide services to customers, and by customers who wish to know that their service providers have achieved some level of proficiency in the services they are offering. Two examples of these kinds of ecosystems are hardware markets, like Storage Area Networks (SANs), that are typically sold and distributed through Value-Added Resellers (VARs), and the IT/system integration and management markets, where certifications such as a Microsoft Certified Systems Engineer (MCSE) have value.

- *Competitive edge.* Certification must provide some form of a competitive edge to the person obtaining the certification to provide them with the necessary motivation to invest their time, energy, and money in the certification process.
- *Currency.* Well designed certification programs include some kind of on-going educational requirement. In general, the more specialized the knowledge, the shorter its half-life. If you're not willing to invest the time to create an on-going program, don't start.
- *Professional recognition.* Another element of well designed certification programs concerns the professional recognition that is associated with a particular kind of certification. In practice, this means that the particular kind of certification must be hard to attain that you have to work to achieve it. If anyone can obtain the certification, why bother?
- *Independent certification.* Although many companies benefit by designing and managing their own certification programs, it is often better if a company supports certification programs designed and managed by one or more independent organizations. This usually reduces overall design and implementation costs and has the chance to include marketplace acceptance. One example of this kind of certification is the Computerized Information System Security Professional (CISSP) certific ation program, created and managed by an independent third party organization. The CISSP is a demanding certification that requires detailed knowledge of a wide variety of vendor-neutral concepts as well as specific knowledge of various vendor products.
- *Academic credentials.* Some certification programs can be used towards degrees offered by major universities. This is, of course, an added advantage.

### 8.6.4 User Community

A healthy and active user community provides a number of benefits to the marketect and the tarchitect. For the marketect, user communities are a place you can go to obtain primary data on desired features, product uses, and product futures. For the tarchitect, understanding the user community means you can understand how people are *really* using your application. I am always amazed at how creatively people can use API if given the chance!

**Thanks, But I Think I'll Build My Own User Interface**

My team had labored mightily to build a great user interface. Not a good one, but a *great one*. A testament to usability. When we learned of a "power user" who both loved our system and wanted to share some of his ideas for improving it, we jumped at the chance.

After a long day of travel, we arrived at his cramped office. He was a senior research scientist working for a major chemical company, and amid the many stacks of paper was a workstation. With an excited wave of his hand, he motioned us over to review his use of our system. Imagine our surprise when we watched him type a few values into Excel, hit a button, and then manipulate the results in a pivot table. What happened to our beautiful user interface? "Oh, it was nice, but it didn't really meet my needs. Fortunately, you guys built a really cool COM API, so I just programmed what I wanted in Excel. Isn't it great?". Yes, it was.

User communities don't magically "happen". They need a place where they can be nurtured. The marketect, especially, should seek to foster a healthy and active user community. Here are some activities that can help in this endeavor.

- Establish a corporately supported community web site where customers can share tips and techniques associated with all aspects of your system. If you've elected to provide an API, provide sections where they can post questions, receive official and unofficial answers, share sample programs, and so forth.
- Distribute unsupported educational and sample programs with your core product distribution so that customers can learn how to create such applications on their own. It is best if these programs are created or certified by your development and/or professional services organization. Make certain that this kind of reference and/or sample code is high quality—you don't want your customers to emulate poor practices. The programs themselves don't have to be tested to the same level of detail as the released software, but it should use the system in a corporately approved manner.

- A very low cost way to stay in touch with your customers is to create an email-based mailing list. While such an email list is a marketing program and should be managed by your outbound marketing department, the contents of each email should be driven by the marketect in consultation with development, QA, support, and professional services.
- A vitally important activity is to have one or more user conferences. Depending on the size of your user base, this can be a single annual event, or there can be multiple conferences throughout the year organized around the world. Organizing an effective user conference is a very specialized skill, and like your email mailing list, the objectives and structure should be driven by the marketect but the conference itself should be managed and run by your outbound marketing department.

### 8.6.5 LICENSE AGREEMENTS

The marketect needs to carefully review proposed lic ense agreements need to make certain that the legal team does not put in language that unnecessarily restricts the use of the API. Such language is usually added with good intentions, but the end result often puts the customer in a tough position.

**No License For Testing**

Simplistic license agreements often inadvertently prevent customers from utilizing systems as intended by marketects and tarchitects. In one enterprise class system I worked on, the original terms of the license agreement restricted our customers from using the system on more than one hardware configuration, but did allow them full and free access to our API. The problem is that such a license agreement does not meet the legitimate needs of the customer to install the system on development and staging (or “model office”) configurations to create and test their integrations and extensions before installing them on production hardware.

One way around this problem is to make certain that your legal team understands your objectives regarding the use of the API. If you’ve created a system that is designed to be integrated and/or extended, your customers are going to need development and testing licenses. Some vendors charge for these rights; others provide them as part of the basic license. I recommend against charging for these rights, because such charges will inhibit customers from integrating and extending their systems. Regardless of whether or not you choose to charge for these rights, you need to make certain that these rights are clearly de fined in the license agreements.

## 8.7 MANAGING APIS OVER MULTIPLE RELEASES

Earlier in this chapter I noted that publishing and API or any other specific technique for extending and integrating your system is an important public commitment to the people relying on your API. The level of commitment associated with an external API provided to customers is *very* different than the level of commitment to an API that developers may have when they are creating this API for other members of their development team. It is relatively easy to negotiate changes to an API within a team, and healthy teams do this frequently during development. It is very difficult to negotiate changes to an external, published API, especially when hundreds or thousands of customers have created applications that rely on it. I've read some accounts about how web services will change all of this, and I don't believe any of them. An external, published API represents a serious commitment from your company to your customers. You can't change it on a whim.

That said, like other aspects of your system, the techniques you put into place to provide for extension and integration, including APIs, will change over time. Early versions of the system may not expose the right kinds of services, or may not expose the services using the right underlying technology. The trick is to carefully manage the evolution of APIs so that the entities that are relying on this integration/extension technique will have time to plan for and accommodate any changes to have to make. Here are some approaches that will help you manage APIs.

- *Give plenty of warning.* Customer should know about changes to extension and integration approaches as soon as possible. A good rule of thumb is to announce changed planned for release $n_{+1}$ of a product in release $n$. To help in the transition from the old API to the new API, consider providing tools to identify old APIs and, when possible, automatically convert old APIs to new ones.

- *Provide one release of overlap.* Once the new APIs are released in version $n$, the old APIs should not be fully deprecated until release $n_{+1}$. In other words, removing an API must be done in no less than two full releases of the product.

- *Provide backward compatibility layers.* Once the API is no longer officially supported you may find it helpful to provide an "unsupported" backward compatibility API (or layer), provided that the underlying functionality of the product hasn't changed. This is often implemented as an optional module that can be installed as needed by users of your system. Be careful with this approach, as it may create a false sense of security among your users that portions of the API will always be present.

- *Provide automated tools that identify or converted calls to deprecated APIs.* Tools that can automatically identify and/or convert code to deprecated APIs and convert them to the new platform can be an invaluable resource to your customers. They also benefit your application, by making it that much easier to move to the upgraded version of the system.

## Chapter Summary

- Integration is the process of extending your system or other systems by programmatically linking your system with others.
- Extension is the process of adding new functionality to your system through well-defined approaches, such as plug-in architectures.
- Integration and extension provide your customers with the chance to create the product that they seek. As an added bonus, integration creates tighter customer relationships and decreases the likelihood that a customer will leave you for a competitor.
- The layered architectural pattern, which organizes various system functions in various logical and physical layers, provides several excellent choices for creating integration and extensions points in enterprise class software systems. The primary layers of a layer architecture are the:
    - user interface layer;
    - services layer;
    - domain model;

  Additional layers may exist within these layers, and other layers may be created to handle specialized requirements.
- Whatever the structure of your architecture, it is helpful to build it in "spikes" – user-visible functionality that is "driven" through all layers or subsystems.
- There are several ways to provide integration and extension points at the services and domain logic layers. Two common approaches include:

    - programmatic techniques, such as exposing an API;
    - registration techniques, such as creating a plugin architecture (like a web browser).
- Integration and extension of persistent data can be accomplished through:
    - views;
    - hook tables;
    - spreadsheet pivot tables;
    - extract, transform, and load (ETL) scripts.
- The business ramifications of creating applications that can be extended and/or integrated include:
    - creating a professional services organization to guide your customers in these activities;
    - creating training programs to ensure your customers understand how to do this on their own;
    - creating certification programs to create an ecosystem associated with your application;
    - establishing and supporting a community of users around your application;
    - ensuring that license agreements explicitly support the integration and extension activities.
- Any form of customer-facing method for integrating and/or extending your application must be carefully managed. You're making a public commitment to stability. Honor it.

## CHECK THIS

- ☐ For each way that the system can be extended or integrated, we have created the necessary supporting materials to ensure that our customers will be able to create the desired extensions or integrations.
- ☐ We have used the context diagram to define how our application might be integrated with other applications.
- ☐ We have established naming conventions to create a consistent API.
- ☐ The performance and resource implications of each call to our API are well documented.

## TRY THIS

1. To what degree can your system be integrated with other systems? How do you know that these techniques work for your target market?

2. To what degree can you system be extended? How do you know that these techniques work for your target market?
3. What kind of certification program have you established?
4. What kinds of third party consultants have worked with your application? What information are you providing to these consultants? How are you using their knowledge and experience to improve your application?
5. What steps have you taken to foster a community of users?

# 9. BRAND AND BRAND ELEMENTS

Product, service, or corporate brands, and their related brand elements, all have a substantial impact on your solution. Recall from chapter two that your branding is *vital* to your total success. You can create a great technology, and even a good solution, but a winning solution will be reflected and supported by a winning brand.

Brand elements, which are often woven throughout the product and become, often surprisingly, part of the architecture, need to be managed. Among other things, you need to select which brand elements will become part of your solution, how these relate to other brand elements of other products, when and how to change them, and how your partners may modify them. In this chapter I will explore how to manage brands and brand elements to create a winning solution.

## 9.1 BRAND ELEMENTS

The brand elements that most likely to impact your product are names, slogans, graphic symbols, designs, and other, external, customer facing or customer visible elements. Even URLs are part of your brand. Brand elements can be registered for legal protection as trademarks within the U.S. Patent and Trademark Office (USPTO). Registered trademarks have special restrictions on their usage, a topic explored later in this section.

### 9.1.1 NAMES

The brand elements that often have the biggest affect on your system are the various names associated with your product and your company. Here are some of the areas in which names can affect your tarchitecture.

- *Physical location of system components.* Most deployment models, including service provider deployment models such as an ASP or MSP, result in one or more software components being installed on the customers or users computer. These can range from simple browser plugins to the full executable and its supporting files, including DLLs, sample data, documentation and help files. Once you've identified the platform-recommended location in which to install the software, chances are you’re still going to have to create a subdirectory or subfolder in which to store your application components. I recommend using the combination of your company name / product name / sub-component name, or, in rare cases, just the product name / sub-component name.

This approach has several advantages. It provides you with additional branding information for your company and your product. It supports expansion in multiple ways: you can conveniently add sub-components to a given product or other products. Even if you sell just a single product, chances are good that you’re going to create, over time, additional modules, and you’re going to need a place to store them.

If you choose to support multiple versions of your application or product on the same system, you may find it convenient to incorporate version numbers in directory names, although this may become irrelevant as operating systems provide increasing support for managing component and component versions. Regardless of operating system support, carefully consider your choice to support multiple versions of the same application, because supporting multiple versions of an application or product on the same system increases testing complexity (remember the matrix of pain?) and often increases support costs associated with version questions. See chapters twelve and fifteen for more information on supporting multiple versions of the same product.

This approach does not remove all risks associated with identifying the right physical location to store or install a product. A company, product, or sub-component name, or their concatenation, may be longer than the allowed length of a subdirectory or subfolder. Any of these may contain special characters that are not allowed by the supporting operating system. And while rare when you’re using fully qualified names, there is always the possibility of name collision. Special symbols that may be associated with the product in printed material, such as ©, ®, or ™, are probably not be allowed. Ultimately, when trying to determine the physical location of your product, your company name / product name / module sub-component is the best place to start.

- *Names of key components.* Complex products often have many customer visible components. In addition to the overall product name, marketects must name each of these components. It is best if the marketect or other person trained in marketing names the product. This is a very important, strategic decision, and you want to make certain that individuals with the proper training and background create these names. Some companies allow developers to name products. I don’t like this approach, because developer-created product names are often quite confusing.

Names must also be created for customer-visible components and/or features. I encourage marketects to include developers in naming these items, because many times the goal is to “technically” accurate names that convey exactly what is going on. These names can sometimes be pretty exciting, and a good marketect can use their straightforward, direct, and descriptive nature to both quickly educate a potential customer on key features and also differentiate the product from the competition. Creative technical names may also be easier to protect.

Tarchitects and developers on the product should realize that developer-created component names, which probably had a strong correlation to the tarchitecture in its first few releases, are much less likely to maintain this strong correlation to the tarchitecture over subsequent releases. If a developer created name catches on, you’re going to want to keep it. At the same time, chances are good that the team will over time want to modify the underlying implementation and possibly even the tarchitecture. The result is that the “technical” name and its underlying implementation will diverge.

Instead of fighting this divergence, accept it. Names associated with the marketecture evolve more slowly than the tarchitecture and its implementation. The usually substantial cost associated with changing the name of a component is not justified just because you want to maintain a strong correlation between the marketecture and the tarchitecture.

I realize that these are somewhat paradoxical arguments. Fortunately, the marketect is the final arbiter and selector of customer visible names. By working with developers, a skilled marketect can usually select the best possible set of names.

**Developers Should Name Variables, Not Products**

Product names are *extremely* important, especially in the consumer market. Product and marketing managers spend a lot of time trying to create good names. They should. A well named product has a significantly greater chance for success than a poorly named product. Like a well named class, variable, or function, a good product name has many desirable attributes. A good name is short, descriptive, describes or highlights one or more positive attributes, is a good fit for the company, can be legally protected via a trademark, has a freely available URL, is unique relative to its competitors, can be easily pronounced and spelled by members of the target market – to name just a few!

To illustrate, here are some of my favorite high-technology product names and why I like them:

- Palm. Can you think of a better name for a hand held computer?
- Latitude. It sounds like a computer made for people on the go. And it is.
- Photoshop. A software workshop for image manipulation.
- Easy CD Creator. The product lives up to its name—a double win!

I'll admit that product names are less important in the enterprise market than in the consumer market. There are a variety of reasons for this, chief of which is that customers of enterprise software often refer to the company more than the product: *Siebel* Sales, *Oracle* Financials, *SAP* R3, *Tibco* ActiveExchange. There are a few exceptions, such as *OpenView* (from HP), or *CICS*, *DB2* and *VTAM* (from IBM). In some sense, the importance of the product name vs. the company name is a bit of a toss-up when it comes to enterprise software—as long as one of them is really good and drives your branding strategy you're probably OK.

The desirable attributes associated with a product name are sufficiently different than technical names that it is almost always better to ask your marketing experts name products. I've inherited a few projects in which the names of products were created by developers. In a few rare cases, this was OK, as the customers found these names acceptable, and the name appropriately addressed a sufficient number of attributes. Most other times, the names had to be changed, because it was too "geeky", the company couldn't secure a trademark, it was too close to a competitors name, or it conveyed a poor brand (such as when a name sounded cool to a developer but a customer viewed it negatively). As discussed later in this chapter, changing the name of a product is a surprisingly expensive operation. Often, this expense can be avoided simply by letting someone skilled in product marketing select the name.

To illustrate just how important it is to review names, one of my developers actually named a database upgrade program after himself! We missed the name in our review and the program was actually shipped to our customers. Quite an embarrasment. A friend to me that some test data on a new server made it into the field. Fortunately, the test data was free from obscenities (is yours?).

- *Names may or may not be internationalized.* One element of internationalization that needs both marketectural and tarchitectural consideration is that the actual name of the product may be different in different target markets. Installing the product into a US-centric directory structure may cause confusion among your user population in countries where English is not the dominant language. In addition, many product names that make good sense in English must be translated into different words in other languages. Any good marketing book can provide you lots of horror stories about good product names created in one culture that were *horrible* product names in another. Ultimately, you have to check all names before blindly assuming that locally created names will work on a global scale. If you suspect that this might be a problem, make certain you're storing brand elements in appropriate resource files or are using some other technique to extend your internationalization architecture into brand elements.

- *Configuration and log files may contain brand elements.* Product and company names can affect the location, name, content, and structure of configuration and log files.

- *Error, diagnostic, and information messages may contain brand elements.* Messages presented must often refer to the marketing names for key components or functional areas of the product. Failure to do this can leave your user and technical support organizations confused on how to identify and resolve problems in the field.

- *Names are volatile, especially in the first release.* The first release of any project is where names are most volatile. This volatility can have negative effects on such things as the source code management system. I recommend that developers refer to the product through a code name for the first release, such as the name of a fruit or some mythological creature. You don't want to create entries in your source code management system that closely reflect the name of the product with the risk that this product oriented name may change two or three times before the release date because marketing is testing yet another name! Because names may change during any release, try to continue using this code name for as long as the product is in development.

### 9.1.2 Graphics, Slogans, and Other Brand Elements

It seems obvious, but sometimes development organizations forget that *graphical user interfaces* will be affected by *graphics and iconic* brand elements. Similarly, *voice based user interfaces* will be affected by *slogans* and other brand elements. While many of these affects are not tarchitectural in nature, they should nonetheless be addressed by the development team. It pays to pay attention to the following areas.

- *Icons and splash screens.* Product management should approve every icon and/or graphic presented to the user as they are all brand elements. I've worked on two projects where the product didn't have a "logo" until we realized that we simply *had* to place some kind of icon on the desktop. In the first project I was fortunate to have a developer who was also a skilled graphic artist. He created a logo that marketing approved, and it later became the approved trademark of the product. On the other project we had to add time to our schedule to have an outside design firm create the necessary icons and splash screens, have marketing approve the design, and ensure certain the icon was sized correctly and used the right set of colors. As you can guess, we missed our initially projected delivery date.

- *Follow brand colors.* Companies often specify everything they can about their brand, including such things as the specific colors, presentation templates and document templates that are either associated with the brand or must be used with the product. Tarchitects should check in with their marketect counterparts to make certain that these elements of the brand are properly managed.

- *Voice branding.* An emerging area of brand management concerns the voice that is chosen for creating voice-based user interfaces. Picking the right voice talent is a powerful way to make certain your brand is properly represented.

### 9.1.3 When To Use "TM"

There are legal implications to the proper use of various brand elements, especially those that have been registered as trademarks. The most important things to keep in mind are as follows.

- *You may lose your legal rights if you don't use your trademark*. Trademarks are based upon use. Failing to use a mark may result in losing it.

- *Give notice.* You have to tell the person viewing the registered information that it is, in fact, a registered trademark. The easiest way to do this is to use the symbol ® following the registered brand element (referred to as a "mark"). There are other ways – consult your in house legal counsel for their preferred approach, as it may differ from the above. Do *NOT* use the symbol ® if the mark has not been registered with the USPTO. Perhaps surprisingly, the government claims that to do so is a crime. Instead, use the symbol ™ for unregistered trademarks.

- *Use marks ONLY as adjectives.* A mark should never be used as a noun or verb, never pluralized, and never used in the possessive form.

- *Include registered trademarks within the distribution and/or prominently display them on usage.* Traditionally, marks were affixed to goods and services. In the world of software, marks must somehow be displayed to the user. Include marks in the distribution, installation, and execution of your software.

- *Domain names can be trademarked, subject to certain restrictions.* If you've created an interesting URL that is associated with your product, service, or company, you may wish to register it as a trademark. Additional information can be found at http://www.uspto.gov/.

## 9.2 MANAGING IN-LICENSE BRANDS

In-licensed technology may have any number of branding requirements. The marketect and tarchitect should work together to ensure that brand elements of in-licensed components or technologies are properly handled.

### You Must Click "OK" to Continue

In one project I worked on we wanted to operate a key in-licensed technology as a service under Windows ‘NT. This would allow us to automatically restart the service if it failed. Unfortunately, the licensor required us to display a splash screen for at least 10 seconds, with an “OK” button, to ensure that when a human operator was involved they would see the licensor’s brand. Unfortunately, the service wouldn’t start until a user clicked the “OK” button. The licensor was unwilling to negotiate this requirement, and we needed to make certain this component could be started and managed as a service. We solved the problem by writing a manager program that would start the service and sleep for 11 seconds. If the splash screen was still displayed it would simulate clicking on the “OK” button.

## 9.3 BRAND ELEMENT CUSTOMIZATIONS

Depending on the type of software you create, your target customer might have legitimate requirements to override or customize key brand elements. Some circumstances that motivate brand element customization include software that is sold as a component in another system, software that is deployed in a service model (where the service provider wants their own brand recognition), or software that, while stand-alone, is integrated into a larger solution by a Value-Added Reseller (VAR), where the VAR wants their brand recognition. Consider that whoever supports the software or whoever is most visible to the user often wants brand recognition.

Marketects should be able to identify requirements for brand element customizations fairly early in the development cycle. Knowing which elements need to change is essential, because you’ll still have to install the software, name key components, and present messages to the user. In addition, make certain you know which brand elements may be customized and those that must be customized. The former will require sensible defaults; the latter will require a way to handle the customization process.

Be certain to include precise details about every brand element that may or must be customized. These details include such things as the size of bitmaps, supported graphic files, default brand elements, and so forth. Keep in mind contract requirements in this process, as the specific requirements of brand element customization may be specified in a contract.

## 9.4 Changing Brand Elements

Changing brand elements is usually harder than changing other parts of the system that present themselves to the user, such as in an internationalized user interface that can be readily localized to different languages. I attribute this to two primary reasons.

First, brand elements change quite slowly, and are therefore are not generally designed with a requirement to support change. Internationalized software, on the other hand, has been designed to support such changes. I am not advocating that you create a set of requirements to support changing brand elements, unless this is specifically required as part of our overall strategy. Specific requirements to support changing brand elements are usually rare.

Second, brand influences a product and its architecture in very subtle ways. The product development team (marketing, engineering, QA, technical publications – everyone associated with creating a winning solution) often does not realize the full requirements when one or more brand elements are changed. I vividly remember the challenges one of my teams faced when we decided to change the name of a product. We debated, and eventually changed, a large number of items that didn't initially appear to be related to the decision to change the product. These ranged from the labels used in our source code management system to the categories in our bug tracking system (more on this later).

There are, of course, many legitimate motivations to change brands and brand elements. Products and components do change names. Companies change names, either voluntarily or as a result of any number of transactions, such as when a company is acquired or when a product line is sold from one company to another. When you're faced with changing a brand or brand elements, consider these areas of the product.

- *Subsystem names.* A tarchitect must name a subsystem or a component *something*. Ideally, this name is functional, but quite often it is related to the product or the brand. As described earlier, it also happens that functional or technical names become part of the product and brand. Changing the name of a product or any of its brand elements may motivate a change to the name of internal subsystems or other components.

- *Source code repositories.* It is easier for everyone if the names of the product and the brand elements either are not reflected in the source code repository (e.g., name things via "code words") or match, as closely as possible, the names of products. If you choose to name repository structures after the product names, spend the time and effort needed to keep these synchronized. It is very confusing to try and track down a set of changes when the source code repository refers to a product that no longer exists!

- *QA and technical support issue tracking systems.* Good QA and technical support systems provide categorizations of bugs. These categorizations inevitably reflect product and brand elements. They should be maintained in close alignment with the actual shipping products. This is especially important if you utilize any kind of customer self-service solutions, such as a web site where customers can review case histories of problems (which also reinforces that your brand is really your *total customer experience*, including technical support!).

- *Physical location of components.* As described earlier in this chapter, it is often easiest to place physical components in locations that reflect the company, the product, and the module. Your upgrade process may have to account for these changes. If you change any of these, be patient. I've worked on products where it took two full releases before all of the changes associated with the new product name were implemented, including the changes associated with installing upgraded versions of the product.

- *Naming and structure of APIs.* APIs often reflect a variety of brand elements, either by name or by function. Company or product names, for example, may appear as prefixes on functions. Functions may be organized or grouped according to product specific branding requirements. Because of this, changing the name of the product or any of its associated brand elements may even require changes to your APIs.

- *Error, information, and diagnostic messages.* Because any of these may contain references to corporate or product brand elements, each needs to be examined for potential changes.

- *QA and test automation infrastructure may need to change.* If you're changing any subsystems, error messages, user interfaces, your automated build system or any externally visible APIs, chances are good that you'll also have to make some modifications to your QA and test automation infrastructure.

- *Sales collateral.* Sales collateral refers to all of the materials that are created to support the sales effort. This includes web sites, white papers, case studies, "glossies", and so forth. These will also require changes if the brand or brand elements change.

The QA effort associated with changing a brand or key brand elements is often quite substantial. With so many parts of the system changing, and with the potential for tarchitectural change, you have to perform a full QA cycle. One of my teams had to track down a particularly nasty bug when the installer, under certain circumstances, put the new software in the old location! Fortunately, this was captured in the beta testing process (which is essential when brand elements change because it gives your customers time to adjust to the change).

## CHAPTER SUMMARY

- Brand elements have a subtle but pervasive impact on your tarchitecture, ranging from the icons that represent the product to the names and location of where it is stored.
- Product names are the domain of the marketect. Good sub-component and feature names may come from developers (but are still controlled by the marketect).
- International poses special challenges to brand elements. It pays to hire a specialist if you're going to go international (and chances are good that you are).
- Understand how to use various legal designations associated with product names.
- Technology in-licenses may impose brand element requirements.

- Review all aspects of the total customer experience when considering changes to a product name.

## CHECK THIS

- ❑ All required brand elements have been identified and approved.
- ❑ All brand elements have been internationalized as required.
- ❑ Error, diagnostic, and log files have been checked for approved brand elements.
- ❑ All marks, including ® and ™, are used properly.
- ❑ We have identified and fully specified any brand elements that may be replaced or changed by a partner or technology licensee.

## TRY THIS

1. How are brand elements used within your application? Consider product and names, graphics, URLS, and so forth.
2. Are there any plans associated for changing any brand elements in the current or near future release? How do you intend to manage this process?
3. Do you work on software that requires you to give your customers brand or brand element control? How do you do this?
4. How do you coordinate brand elements with other product lines?

# 10. USABILITY

Winning solutions are usable. They may not be perfect. They may not win design awards from fancy committees where people dress in black, sip champagne, and pontificate on issues like "convergence". But they are usable: They allow their users to accomplish necessary tasks easily, efficiently, and with a minimum number of errors. They enable them to achieve their goals with little, if any, frustration.

I've already touched on two important, but overlooked aspects of usability: deployment and integration/extension architectures. Subsequent chapters will focus on other aspects of usability in the context of operations and maintenance, notably installations and upgrades. This chapter briefly explores some of the most important aspects of usability, including performance and scalability. Simply put, just about everything that the tarchitect does affects usability in one way or another. Gaining an appreciation of usability, and the manner in which architectural decisions affect usability, and how usability affects architecture, is critical to creating winning solutions.

As you read this chapter, keep in mind that usability is a very large topic. Several terrific books have been written about usability. Even if you've read these other books you'll find the information in this chapter useful, because I discuss usability from the context of a winning solution. My primary focus is how tarchitecture impacts usability, with an emphasis on various performance factors. This differs from books that address process or technology issues.

## 10.1 USABILITY IS ABOUT MONEY

Historically, usability has been associated with the user interface, and human-computer interaction (HCI) professionals have tended to concentrate their work here. This makes perfect sense, because most of our understanding of a given system and most of our perceptions of its usability are shaped by the user interface.

Usability is much deeper than the user interface. In my view, usability refers to the complex set of choices that end up allowing the users of the system to accomplish one or more specific tasks easily, efficiently, enjoyably, and with a minimum of errors. Many, although not all, of these choices are directly influenced by your tarchitecture. In other words, if your system is perceived as "usable", it will be usable because it was fundamentally architected to be usable.

This view of usability makes certain demands on marketects and tarchitects. Both, for different reasons, must gain an understanding of the user and the context in which they work. The marketect needs these data in order to ensure that their system is providing a competitive edge, that it is meeting the most important needs of their users, and that it is providing the necessary foundation for a long-term relationship with their customers. The tarchitect needs these data to understand how to create the deeper capabilities – from performance to internationalization – that result in usable systems.

A trivial example can be constructed by considering users in sufficiently different domains. An application designed to help a biochemist manage the vast amount of data created by today's automated experiment equipment is vastly different than the self-service kiosk I use to obtain a boarding pass before boarding a flight. Developing a clear understanding of what makes these systems usable can only be done by understanding representative users and the context in which they work. The marketect needs to understand the use cases associated with both sets of users, their goals, their motivations, how creating these systems will serve their needs, and so forth. The tarchitect must also understand the uses cases associated with these systems, even though the underlying technology associated with each solution is quite radically different.

A more subtle reason why understanding users and the context in which they work is important concerns the quantitative and qualitative dimensions of usability. Quantitative aspects of usability address such things as performance or data entry error rates. Qualitative aspects of usability address such things as satisfaction or learnability. Because no application can maximize every possible quantitative or qualitative value, tradeoffs must be made. Making these tradeoffs well produces usable applications and winning solutions.

There are a variety of techniques that you can use to gain an understanding of users, including observation, interviews, questionnaires, and direct experience. The last approach is one of my favorites, a technique I refer to as "experiential requirements". Simply put, perform the tasks of your users as best you can to gain an understanding of how to create a system that they would find usable. This approach isn't appropriate for many situations (e.g., surgery or high speed race car driving) but it is one of the best!

The fundamental, almost overpowering, motivation to creating usable systems is money. A large amount of compelling data indicates that usability is an investment that pays for itself quickly over the life of the product. Whereas a detailed summary of the economic impact of usability is beyond the scope of this book, one system I worked on was used by one of the world's largest online retailers. A single phone call to customer support could destroy the profits associated with a dozen or more successful transactions. Usability, in this case, was paramount. Other applications may not be quite as sensitive to usability, but my experience demonstrates that marketects and tarchitects routinely underestimate the importance of usability. Benefits of usable systems include any or all of the following, each of which can be quantified:

- Reduced training costs
- Reduced support and service costs
- Reduced error costs
- Increased productivity of users
- Increased customer satisfaction
- Increased maintainability

It is generally not possible to simultaneously achieve every possible benefit of usability. A system that is easy to learn for novices is likely to be far too slow for experts. Because of this, it is vital that marketects and tarchitects work together to ensure that the correct aspects of usability are being emphasized in the project.

**Market Demands And Usability**

When you go to the store to purchase a piece of software for your personal use you have the right to refuse any application that does not appear to meet your needs—including your subjective preferences. This effect of market forces continues to drive the designers of shrink-wrapped software to create systems that are increasingly easy to use.

Unfortunately, designers of software applications in large corporations are usually not driven by market forces. Instead, they attend to other forces—typically the needs of senior management. As a result, the usability of many applications in large corporations is abysmal: poorly designed, manipulative, and even hostile applications serve to denigrate the dignity and humanity of users.

If you are a tarchitect working on an internal application in a large corporation I urge you to pay special attention to the suggestions listed in the rest of this chapter. While you may not be driven by the same market forces as tarchitects creating shrink-wrapped applications, the benefits of usability are as applicable within the corporation as they are in the external marketplace.

## 10.2 MENTAL MODELS, METAPHORS, AND USABILITY

Once you've gained a concrete understanding of how users approach their task, you can begin to develop your understanding of their mental models associated with these tasks. A *mental model* is the set of thoughts and structures that we use to explain, simulate, predict, or control objects in the world. Mental models are shaped by the task and the tools used to accomplish this task, which means that they can change over time (more on this later). As far as I know, there is no language or notation that formally documents mental models. At best, mental models are informal observations that the development team can use to create more usable systems. For example, consider a designer creating a new project planning tool. Through several interviews she has discovered that managers think of dependencies within the project as a web or maze instead of a GANTT or PERT chart.

In creating the new planning tool, the development team will create one or more conceptual models to share their understanding with others. A *conceptual model* is some representation of a mental model, in words or pictures, using informal or formal techniques. Thus, a UML class diagram of the primary concepts in the system is a conceptual model.

An understanding of mental models, and the clarifying role of conceptual models, form the foundation for the creation of metaphors that can be used in the creation or modification of a system. *Metaphors* are models that help us understand one thing in terms of another. Tarchitects often use metaphors to structure their application architectures. In one system I worked on, we had a requirement to support message passing between distributed systems in such a way that any system in the chain could add or remove data to the core message without altering the contents of the core message. To help explain this concept to potential customers, I created the metaphor of a Velcro covered ball, where the ball represented the core message, and where the data that customers could add to the message were "attached" or "removed" to the message much like we could "attach" or "remove" pieces of Velcro to the Velcro ball.

A well-chosen metaphor shapes both the tarchitecture and the user interface. In the planning example described above, the system's underlying tarchitecture could easily have one or more functions that support manipulating project plans in terms of a web or a maze. This choice, in turn, could provide insight into creative new ways of organizing tasks, displaying information, or providing notification of critical path dependencies.

Metaphors are often most apparent when considering the marketecture, primarily because the metaphors chosen by a tarchitect must be communicated in a targeted manner to a well-defined market. Metaphors can influence a variety of brand elements, including names and iconic representations of products or processes. Metaphors often affect the pricing model, by highlighting those areas of the system that are of greatest value to the defined market segment. Entirely new products rely on metaphor to both explain how they work and to provide marketects with the vehicle for shaping future customers mental models about these products, how they work, and the benefits they provide, to prospective users. All of these reasons motivate close collaboration between marketects and tarchitects in the development of the system metaphor. In the example of a project planning tool, the marketect may prefer the metaphor of a maze over the metaphor of a spider web because a maze is a better fit for the positioning selected for the resultant product.

The final benefit associated with well chosen metaphors, and arguably the most important reason to pursue them, is their effect on usability. A well chosen metaphor substantially improves the usability of the system by tapping into users existing mental models of their work and how it should be performed. Training costs are substantially reduced, satisfaction improves, and overall comfort with the system increases. An excellent catalog of prospective metaphors can be found in *Designing Object-Oriented User Interfaces* by Dave Collins.

## 10.3 TARCHITECTURAL INFLUENCES ON USER INTERFACE DESIGN

You can find quite a lot of tarchitectural advice, including my description of layered architectures in chapter eight, that recommends separating the user interface logic and design from the rest of the underlying system. This is good advice, because such elements of design as your business logic should be maintained in a manner that allows for multiple representations. It is also true that you want the user interface to have the maximum flexibility in presenting data in a way that is most meaningful for the intended user. To illustrate, consider an application that monitors processes on a factory floor. One of the monitoring devices may be a Thermostat, which is integrated with a specific piece of equipment. The user interface designer should be free to represent the Thermostat in whatever way makes most sense: as a gauge, a dial, a simple numerical display, or perhaps something even more creative that hasn't already been done before.

That said, my experience shows that in most applications there is no practical way to purely separate the user interface from the rest of the tarchitecture. In fact, as application complexity increases you often find yourself relying on the idiosyncratic benefits provided by a given kind of user interface, whether it be command line, graphical, haptic, auditory, or any combination of these things. Here are some specific areas where tarchitectural influences user interface design and usability.

- *Cardinality*. Cardinality refers to the number of entities that participate in a given relationship. It is an important element of overall tarchitectural and user interface design because the techniques associated with storing, processing, and representing small numbers of entities vary quite a bit from the techniques associated with storing, processing, and representing large numbers of entities. As cardinality increases, the demand for visual representations of the data and their interactions also increases. Fortunately, the number of tools available for visualizing large data sets continues to increase.

- *Feedback*. One of the most important heuristics associated with creating usable systems is to provide various kinds of feedback to the user. Feedback assumes a variety of forms, including such things as informational messages, responses to user actions (such as a menu being displayed when selected by the user), progress meters, and so forth. The tarchitecture must be created with an awareness of feedback demands.

  Consider the effects of a progress meter. Progress meters are usually used to indicate that the system is processing a request or transaction that is going to take a fairly long amount of time. This means that the portion of the system that is servicing the request must be created to allow periodic feedback on its processing status. This can be relatively simple, such as when you've issued a request to perform some operation over a finite number of elements, and in which processing each elements takes a well-known period of time. It can also be quite complex, as there are a variety of operations for which you cannot predict the amount of time it takes to complete in advance. Note that this fact also influences the specific kind of progress meter chosen, another example of how underlying tarchitectural issues influence the design of the user interface.

A special form of feedback concerns early validation. In early validation, the system is constructed so that it can perform partial validation or near real-time evaluation on every piece of data that is entered. A trivial example is a data input field for a numeric part number that will only accept numeric characters. A more complex example is an input screen that can partia lly validate data or an input screen that will enable/disable various controls based on previously entered data. Early validation can't be performed on many web-based systems because the underlying capabilities of HTML and a browser can't provide it.

- *Explicit user models.* One mechanism to take advantage of your understanding of your users mental models is to create in your system an explicit model of their understanding and adjust system behavior over time according to this model. This technique is used in everything from my word processor, that automatically corrects certain kinds of spelling mistakes, to my favorite online bookstore, that recommends books according to my previous purchases as compared to a massive database of other users purchasing history.

- *Workflow support.* System architectures and markets often mature together. This is one reason why so many initial releases of a system have poor support for workflow, and that so many mature releases have one or more elements that in some way provide explicit support for workflow. As the market and use of the system have matured, product development teams were able to capture "best practices" and codify them through any number of simple (e.g., such things as wizards or animated help systems) or complex (e.g., scriptable workflow engines) workflow support structures. Supporting workflow usually builds on top of the understanding we develop regarding the users mental model, and is in itself a kind of explicit user model.

- *Transaction support.* Many of the operations that we perform in a system can be thought of as a "transaction". To illustrate, simply typing a character into a program that supports undo is a kind of transaction. Inevitably, transactions, and transaction semantics, inextricably affect the user interface, and vice versa. Consider a web site constructed to obtain flight information, in which the web site is constructed according to the layered architecture presented in chapter eight. In the process of extending this system to support a voice-based user interface, you're likely to find that while the underlying domain objects and many of the services are perfectly reusable, certain transaction-oriented services are at all reusable. This is because transactions are often designed to support a particular kind of user interface capability. In the case of the user interface just described, you can create a very large transaction object that captures many fields from the user in a single screen. This is simply not possible in a voice based user interface, where user input must be obtained via smaller "chunks" because of the complexity associated with asking the user to correct errors.

Another example concerns the presentation of information to the user. In a graphical user interface, it is easy to provide the user with cues as to sorting order of information. Thus, your web-based email program might present all of the messages in your inbox, with simple visual cues as to how the information is sorted. In a voice based user interface, chances are good that you won't be able to reuse this kind of logic when presenting information to the user, and that the underlying logic that obtains these data will also need to change.

- *Validation response techniques are user interface dependent.* How you gather, validate, and respond to user input errors are determined by a complex interaction between your underlying system architecture, your user interface, and the tasks associated with the user. My favorite C++ compiler presents a hyper-linked list of errors in a window; simply selecting an error takes me to that place in the source code where the compiler thinks the error has occurred. This technique doesn't work for an eCommerce web site, which must respond to errors with some kind of notification screen (perhaps the fields are in red or preceded by as asterisk). The choices made regarding presentation of errors will likely effect the underlying tarchitecture.

- *Internationalization and localization are strong influences on the architecture.* Internationalization is the process whereby the underlying tarchitecture is constructed to support multiple languages and localized data formats. Localization is the process of specifying a given language and style of interaction and/or presentation. The ramifications are substantial, and range from formatting monetary values, dates, and numbers to hiring special firms to translate all information presented to a user into the set of supported languages. Operating systems and their associated platforms provide the necessary infrastructure for determining which language the user is using. It is up to you to make certain that you're taking advantage of this.

It is important to consider everything that is presented to the user in the context of creating internationalized applications: error messages, informational messages, dialogs, input/output formats, log files, and even the names of external APIs are all candidates for internationalization. Fixing the amount of screen real estate, as when a team decides to use fixed sized dialogs throughout their application, are the bane of internationalization efforts. The most basic heuristic is to always refer to information presented to the user within your code through some sort of numeric identifier and create a component to handle the responsibility of converting this numeric identifier into the desired localized output. This module can draw the localized information from resource files, from a remote server, from a special "installation only" component that selects the proper language once during installation, or from some other proprietary storage format that can be dynamically selected at run-time.

Because of the very far-reaching impacts of internationalization and localization, it is reasonable to demand that marketects specify their target languages very early in the development process, as well as those languages likely to be used down the road. If in doubt, assume the possibility of Far Eastern and Right to Left languages, and design the architecture accordingly.

**Details, Details, Details**

Some of my most challenging experiences have been in creating software that is localized for use around the world. Here are some of the problems I've encountered. In one project, in which our technology was embedded in other users systems, one Japanese customer continually refused our deliverables because of a variety of spelling or grammatical mistakes. Their attention to detail was extraordinary, and I was generally glad to make their requested changes because of my belief that it improved the overall quality of the product. But when they submitted a change that reverted the user interface to a previous version, I realized that they were conducting their own beta tests to see which version of the user interface their customers liked best – and forcing us to pay for the results. Normally, I wouldn't mind, but their beta testing was starting to negatively affect our other Japanese customers.

In another project we needed to localize the client to support 16 languages on both the Macintosh and MS-Windows platforms. We spent about one week chasing down a variety of bugs associated with the Macintosh translation. As it turns out, our internationalization vendor used an MS-Windows character set when creating the files. The resultant bug was extraordinarily hard to track down; once found, it was an easy bug to fix.

In general, localized software almost certainly requires a strong beta program conducted with customers and/or partners who are experts in the highest priority target languages. Most development organizations simply don't have the necessary diversity to properly test the results of the translations provided by your localization vendor.

- *Canceling requests.* Requests that users make in client-server based architectures must be cancelable. One system I worked on hard a very hard time canceling requests because a given request was distributed across multiple, in-licensed components (a database from one vendor and a search engine from another). Coordinating the cancellation process across these elements proved quite challenging.

  I'm concerned that many of the published approaches to creating web services do not address the issues associated with canceling a request. Suppose, for example, I have a web service that takes a long time to compute a result or consumes a large amount of resources while computing a result. How do I "cancel" the request once it has been initiated? Creating the necessary infrastructure for canceling a request is important for any system – make certain you know how to do this in yours.

- *Timeouts.* Systems that require a session with another system or the user often have a variety of timeouts that are associated with their operation. Choosing the wrong value can lower usability, consume unnecessary system resources, and ultimately require the user to perform more work to accomplish their desired tasks. This is one area where sensibly chosen defaults can really help your application be perceived as more usable.

- *Network availability/speed.* There are lots of architectures that use networks, but don't have the luxury of an always on connection, speed guarantees, or notification of a change in these states.

- *Managing shared resources.* All systems have one or more resources that are ultimately shared. I've worked with searching systems, for example, that took control over all available processors in a multi-processor system, leaving none available for other work. A better design would enable the system administrator to tune the amount of processors the search service consumed.

- *Failure recovery.* Systems can fail in a variety of ways. Tarchitectural choices make certain kinds of failures more or less likely. Once a failure occurs, how your system handles the resultant failure directly impacts usability. Ideally, your system handle s and recovers from the failure in a reasonable manner, such as when your printer driver attempts to retries your print job on a network failure. Failure conditions also provide fodder for enhancing the system design, especially when you can convert previously unhandled failure conditions into failure conditions that become automatically handled and thereby do not disrupt the user.

  Even if you focus on handling as many failure conditions as possible, chances are good that you'll have to alert the user to some of the failure conditions associated with your system. You can inform your users to any of the following: the existence and magnitude or potential negative effects of the failure; potential and/or recommended recovery procedures; and preventative actions the user can take to prevent further occurrence of the failure.

## 10.4 THE NEED FOR SPEED

An important aspect of usability is performance, both actual performance and the performance perceived by the user. Unfortunately, this large topic area is quite misunderstood. It is beyond the scope of this book to provide detailed advice on how to create high performance systems. Instead, my focus will be on providing a precise definition of performance related terms so that the marketect and tarchitect can communicate clearly on this topic, and on showing what information regarding performance the development team needs to provide the marketect.

### 10.4.1 LET'S BE CLEAR ON WHAT WE'RE TALKING ABOUT

The terminology that I use to describe performance is presented in the following table.

| **Term** | **Definition** |
|---|---|
| Throughput | How many {bits \| transactions} per unit of time |
| Performance | Time per unit (inverse of throughput) |
| Latency | How long I'm waiting to get an answer. |
| Capacity | Number of users (entities) system can support in a given configuration at a fixed level of performance. |
| Scalability | Ability to increase capacity by "throwing hardware" at the problem. |
| Reliability | How long can this thing run without failing? |
| Response time | The total time that I perceive to be associated with processing a request. An emotional and subjective qualitative rating, it is the "human" dimension of performance. A friend defines it this way: "The time between when I tell the system to do something and the time when it admits that it's trying to do it". |

All of these terms are related to each other. To illustrate, let's assume that you want to calculate results for some of these terms for one of your systems. Your first step would be to specify, or "fix", one or more one or more configurations of your system. This could mean specifying such things as the underlying hardware, amount of disk space, memory size, operating system, database, web server, and any other applications required. As for configurations, consider specifying the following:

- Below: below the minimum specified configuration-- just for testing.
- Minimum: minimum specified configuration. This should be well below the average configuration in your target market (market configuration).
- Average: Slightly BELOW the market configuration.
- Ideal: Market configuration of early adopters/influential end users. (Demo configuration)

- Max: The best system you can (practically) assemble.

The next step is to identify the transaction or operation you wish to test, complete with reference data. If you're going to test the performance of a spreadsheet recalculation, you have to have the same data available for every test run!

In this example, I'll assume your system is server-based system, constructed in a manner similar to that described in chapter eight. Once you've got the server up and running, you can program a test or driver program to simulate a request. Let's say that when you send a single request for a specific, common operation your system responds in .1 second. This is a base level of performance. If you configured your test driver to stimulate your system with up to 10 requests per second, and it ran without failing for 24 hours while maintaining the same performance, you'd also have some base data for reliability.

As you begin to increase the number of transactions you'll probably start to notice more interesting things. You might expect the system to fail when the number of transactions exceeds 600 per minute (since 600 * 0.1 = 60 seconds). In reality, you may find that the system doesn't fail until the number of transaction exceeds (say) 700 per minute. This shows the difference between latency and throughput on the *total* system, because parts of various transactions are held in different components as they pass through the various layers.

Let's further assume that you can separate the transaction logic from the persistent store. This means that you should be able to increase capacity by adding more machines. Let's say that you do this. You add more machines, but because you have a single, shared database all of these transaction servers are pointed at the same database. You might find that you can now handle 2,500 transactions per minute before some kind of failure, such as a completely dropped request, occurs. This is a measure of the scalability of your system – the ability to add more hardware to improve various performance factors. If you find that you can't improve beyond this, then you know you've reached the maximum of the database server. Improving this may require you to optimize the specific database server in some special way, such as installing special disk processing hardware, changing database vendors, tuning or optimizing the database by doing such things as adding indices or profiling SQL queries.

Note that in this example the mechanism used to improve performance shifted as various limits were reached. This is common in complex systems. More memory might buy you better performance – to a point. Then you have to get a faster processor or perhaps faster disk hardware.

You might also find that your system does not exhibit linear performance. In fact, complex systems seldom exhibit linear performance curves. Most often, you will find have linear responses for small areas of the various curves that would be plotted from these data. Don't be surprised if your system to "falls over" or "cliffs" when certain tolerances are exceeded. Thus, the reason for having a scalable system (a system that you can "throw hardware at" to increase capacity at an acceptable level of performance).

My example thus far has assumed that the driver program sends requests in a uniform manner. This does not match reality. Most non-batch processing, server-based systems have "bursts" of activity. Since servers queue requests, it's possible to handle bursts of transactions that are significantly higher than the steady-state throughput limit without failure. For example, your original configuration may be able to handle 1400 requests that come in during the first minute, if no requests come in during the second minute. Since different servers vary in the number of requests they can successfully queue, it's also a good idea to know the limit for the queue. That will give you the maximum "burst" that can be handled by the server, which is another factor in reliability.

The perceived response time associated with the system may not actually match the performance just described. Performance is typically measured at a layer below the user interface, because the user interface layers can add any number of additional processing steps or transformations that may not be considered a realistic part of performance. Moreover, response time may be more unpredictable than performance. Still, response is important, and having a good response time really matters because it is associated with our perception of the system and our ability to use it to perform useful work. Subsecond response times are often required to create the perception that the system is responding to our needs in an "instantaneous" manner. Response times of less than 5-10 seconds are needed to enable the average user to maintain an uninterrupted flow of thought. Users may lose focus on their task if the response time is greater than 10 seconds.

Reliability also deals with the "gracefulness" of failure handling. If a server crashes absolutely when its peak load is exceeded, it's less reliable than a server that sends back an error code in response to the messages it can't handle or simply refuses to allow a user to login.

It is also important to note that it is rare for a system to be processing only one kind of transaction or operation. Since most systems support a variety of operations, each taking different amounts of time and resources, the ideal way to measure performance attributes is to define some kind of user model so that you can understand the effects of an "average" user on your system. Some authors recommend creating a stochastic model of system usage, but this is only appropriate when requests are truly random in nature. Most systems do not have random distribution of requests, so a random model can give misleading data. A better source for the data necessary to create a good user model is the log files associated with the operation of your system from a production environment.

In addition to the complexities associated with testing the system, your should also learn to be conservative in your performance estimates. Real world conditions can vary quite a bit from lab conditions, causing potentially misleading results.

### 10.4.2 What A Marketect Really Wants With Respect To Performance

It is tempting to think that what a marketect really wants is the fastest possible system. This is most obviously true: *of course* everyone associated with the project wants the fastest possible system. A good engineer hates waste, and poor performance grates on their sensibilities. Poor performance is harmful to customer relationships, as it forces them to purchase unnecessary and unwanted equipment and/or limit growth in some important manner. But the fastest possible system is not what a marketect *really* wants.

What your marketect really wants is a way to confidently, reliably, and above all, accurately answer performance related questions. *That* is more important than simply building the fastest possible system, because this is what is needed to create a winning solution.

To illustrate what I mean, here are sample questions that I've received from customers regarding various performance attributes. Recall that each of these questions are asked relative to a specific total system configuration (hardware, storage, network, and so forth).

- How many simultaneous { users | connections | sessions } will the system support? What is the load on various components?
- How many transactions can be processed with my current hardware (where the list of hardware was detailed in a very precise manner)?

The biggest reason that marketects need a way to confidently, reliably, and above all, accurately performance related questions is that most of the time customers don't ask raw performance questions. Instead, customers come with some basic understanding of their needs and their environment and ask the sales and/or services organization for help in creating the required infrastructure to support their needs. The marketect needs to provide some way to answer these questions. To get a sense for what I mean, consider the following questions, abstracted from real customer requests.

- We anticipate 200 active and up to 500 casual users. What configuration of hardware do you recommend to support these users?
- We estimate that our system will need to support 800 downloads per hour during the new product release. How should we structure our web infrastructure to support these downloads?
- We estimate that our system will initially 2,500,000 requests annually, with a projected 25% growth rate. Make certain your hardware estimates account for a three year period and that your design provides for at least 98.5% (or 99%, or 99.9%) overall system availability.
- If the database size was to grow at a faster than anticipated rate, what impact would there be to the system? Where would possible bottlenecks occur and how would you scale the proposed system to handle this issue?

Questions such as these form the foundation for a long and complex sales process. Answering these questions well usually requires additional information from the customer. Ultimately, however, the marketect must be provided with the necessary data to create accurate answers.

One of the most effective ways to communicate this information in a comprehensible way is through case studies or white papers, with an appendix outlining additional performance scenarios. Another method, suitable for use in complex performance scenarios, is to provide a program that will estimate performance under various conditions.

The numbers described above must be periodically recalculated to reflect the effect of new releases and new underlying hardware. We have been collectively conditioned to expect superior performance in every aspect of our system. Response times of three seconds in a first release may be completely unacceptable in a third or fourth release. If a customer invests in new hardware, it is entirely reasonable to expect that their application is going to run faster. Performance always matters.

### 10.4.3 Responding to The User

One of the timeless pieces of advice from the usability literature is that your system needs to find some way to provide appropriate feedback to the user. Unless the system can respond in a truly instantaneous manner, you're going to need some feedback mechanism. In general, there is a strong correlation between problem/application/system complexity and the challenges associated with creating a system that can provide good feedback to your user.

What has worked well for me is to first broadly characterize the kinds of feedback that you can provide. The following tables lists two categories.

| **Kind of Feedback** | **Examples** |
|---|---|
| Immediate<br>When the task is expected to take less than 1-2 seconds | • Visual changes to common objects, such as changing the cursor very briefly to a "mail has arrived" icon when your mail client has downloaded a new email message in the background<br>• Auditory responses, such as a beep (use sparingly and put under the control of the user)<br>• Information messages or other changes displayed in status bars<br>• Be careful of your estimates – tarchitects are notoriously bad at estimating how long something will take, and if you estimate wrong you're going to choose the wrong kind of feedback. |
| Continuous<br>When the task is going to take more than 2 seconds | • General animation in an animation "loop" is appropriate when you have no idea how long a task will take, such as the spinning globe that appears in Microsoft Internet Explorer when it is loading a web page.<br>• Percent-done progress indicators, when you can estimate time or effort to completing a task. These can be shown in a status bar (if the task cannot be cancelled) or in a separate dialog (if the task can be cancelled). |

Percent-done progress are your best choice when you need to provide continuous feedback to the user. They reassure the user that the system is processing their request. Ideally, they allow the user to optionally cancel a task if it taking too long. The very best progress indicators also provide a reasonably accurate estimate of the total amount of time needed to complete the task.

**Providing Feedback Prevents Unnecessary Work**

One client/server system I worked on didn't provide users with enough feedback regarding the processing of their requests. As a result, users would submit the same request multiple times, at times overloading the sever with redundant, useless, requests. The problem went away once we implemented appropriate feedback mechanisms. This illustrates the main point of this system: response time and feedback are *qualitative, emotional* perceptions of system performance. You are far better off creating the right emotional response of acceptable response than trying to provide cold facts and figures about system throughput.

### 10.4.4 PERFORMANCE AND TARCHITECTURAL IMPACT

The authors of *Software Architecture in Practice* note that the performance factors described above are affected by both architectural and non-architectural choices. Architectural choices that affect these factors include the allocation of functions among various components, the manner in which these components inter-operate, how they are operationally deployed, the management of state, and the management of persistent data. Different architectural choices affect different factors, and the needs of the overall system must be considered as choices are made. For example, converting a "stateful" architecture to a "stateless" architecture may increase latency but dramatically improve scalability.

Non-architectural choices that affect performance include the algorithms chosen for key operations within a single component and any number of technology idioms that are related to the specific implementation. Examples of non-architectural choices that improve performance are doing such things as implementing a more efficient sort algorithm, or slightly restructuring a database to improve query performance

Managing performance factors is a complex topic, and a detailed study of these factors is beyond the scope of this book. That said, there are some basic tools and tricks that every tarchitect should have at their disposal when considering performance factors. Here are some architectural choices that have worked well for me.

- *Throw hardware at the problem.* In many of the systems I've worked on, the easiest way to solve a performance issue was to throw some hardware at the problem. Sometimes a bit more memory will fix the problem. Sometimes an extra CPU is better than a faster CPU. The specific ways that you can throw hardware at a problem are nearly endless, so architect your system to take advantage of the ones that matter most to your customers.

  Of course, there is a fine balance. Too many engineers believe that throwing hardware at the problem is a justification for sloppy development practices. I vividly remember a conversation with an engineer that I had to ultimately fire because of poor coding practices. He worked on a system that was a collection of CGIs written in C++. His coding style produced massive memory leaks, which he justified as acceptable because all we had to do was purchase additional memory to "cover up" his mistakes. There is *no* justification for such sloppy development practices, no matter how much hardware you have at your disposal!

  Finally, this will only work if you understand the "scalability" of your tarchitecture.

- *Use large-grained transactions.* Performance is typically enhanced in distributed systems when the transactions between components are fairly large.

- *Understand the effects of threading on performance.* Multi-CPU systems are pretty much the standard for servers. Unix-based operating systems, such as Solaris, have proven that they can scale to dozens of processors. You should understand how multiple CPUs affects the performance of your architecture. It can be different than you might think. In one application we relied on a rather old searching engine. We thought that adding processors would help performance. It didn't, because this searching engine wasn't multi-threaded.

- *Use a profiler.* Performance can only be increased in a reliable manner if you know what is too slow. One technique is to use a profiler. But, be forewarned that a profiler can only identify and improve situations where non-architectural bottlenecks exist. When profiling shows an architectural problem, it often means that that you either live with it -- or rewrite.

  Another technique is to run your program on a small number of data elements and then extrapolate the results to a larger number of data elements. This technique can tell you fairly quickly if you're headed in the wrong direction.

- *Handle normal operations and failure conditions separately.* This is a variant of advice I first read in Butler Lampson's excellent paper *Hints for Computer System Design.* In general, normal operations should be fast. Failure conditions, which presumably happen much less frequently, should be handled in a way that is appropriate. There is usually very little motivation to recover quickly.

- *Cache results.* In its simplistic terms, a cache simple saves some previously computed result so that it can be reused. Caches can have an enormous impact on performance in an extraordinary number of circumstances, from operating systems to companies that improve various performance factors on the internet, such as latency, by caching web pages. If you elect to use a cache, be certain that you understand when and how a cached result should be recomputed. Failure to do so means that you will inevitably be using incorrect results. The ramifications of this vary considerably by application, but in any case you should know what can go wrong if you are using poor results. Butler Lampson refers to cached results that could be wrong as a "hint". Hints, like caches, are surprisingly useful in improving performance.

  Make sure that your architecture has a programmatic ability to turn on and off caching "on-the-fly" so that you can test the impact of caching. Cache problems are among the most insidious bugs to find and fix.

- *Perform work in the background.* In one system that I worked on, one of the most frequent requests from customers was to issue print commands as a background task. They were right, and we should have designed printing as a background task right from the start. There are always a large number of processes that can be handled as background tasks. Find them.

- *Design self service operations.* One of most striking examples of improving efficiency in non-computer systems deals with self-service. From ATMs to "pay at the pump" gas stations, consumers are constantly provided with opportunities to serve our own needs. Although not necessarily intended, self service operations can improve any number of specific performance parameters. I've found that this concept also helps in architectural design. For example, by letting client components choose how they want to process results, I've found ways to dramatically simplify client-server systems. Consider an application that enables the user to download pre-packaged data into an spreadsheet, where the user can then utilize any number of built-in graphing tools to manipulate the results. Of course, you may have legitimate concerns regarding the distribution of data in this kind of solution, but the performance benefits of self service designs cannot be ignored.

- *Learn the idioms of your implementation platform.* Your implementation platform – language, operating system, database, and so forth – all have a wide variety of idioms on how to use them efficiently. Generally techniques for improving performance, such as passing by reference in C++ or preventing the creation of unnecessary objects in Java can have a surprisingly large effect on performance. Keep in mind that I am *not* advocating that you simply design for performance. Instead, I'm merely pointing out that one of the ways to improve overall performance is to make certain you're using your implementation technologies in a sensible way. The only way to do this is to thoroughly understand your implementation platforms.

- *Reduce work*. Perhaps this advice is a bit trite, but it is surprising how often an application or system performs unnecessary work. In languages such as Java, C++, or Smalltalk, unnecessary work often takes the form of creating too many objects. In reference to persistent storage, unnecessary work often takes the form of having the application perform one or more operations in a sequential manner, when restructuring the data or the operations would enable the database to perform the work or would enable the application to produce the same result through batch processing. A related technique is to use stored procedures or database triggers. The non-database world provides such examples as pre-computed values or lazy initialization.

## Chapter Summary

- Usability is about creating systems that allow their users to accomplish necessary tasks easily, efficiently, and with a minimum number of errors. Usability enables users to achieve their goals with little, if any, frustration.
- Usability is a core feature of your product brand. Like your brand, it touches every aspect of the product.
- Winning solutions are usable; usability contributes to long-term profit.
- A *mental model* is the set of thoughts and structures that we use to explain, simulate, predict, or control objects in the world.
- A *conceptual model* is some representation of a mental model, in words or pictures, using informal or formal techniques.
- *Metaphors* are models that help us understand one thing in terms of another.
- Maintainability is enhanced by separating the user interface from the rest of the tarchitecture, even though there is no absolutely certain way to separate the influence of the user interface from the rest of the tarchitecture.
- Performance matters. This isn't a justification to pursue design decisions purely in the context of performance, just an acknowledgement that performance always matters.
- Marketects want a way to confidently, reliably, and above all, accurately answer performance related questions. This ability is especially important in enterprise software.
- There are a variety of techniques that you can use to improve performance, including caching, performing work in the background, offloading work to other processors, or avoiding work entirely.

## Check This

- ❑ We have tested the usability of key tasks within our system.
- ❑ We have captured a conceptual model (perhaps using the UML). We have used this conceptual model and our understanding of the users mental model to create a system metaphor.
- ❑ The system metaphor is readily apparent in the design and implementation the system.

- ❑ We have agreed upon the terms that define performance. We have provided a way for marketing and sales to estimate configurations.
- ❑ We know when and how to "throw hardware at the problem" and what will happen when we do!

## TRY THIS

1. Do you have a user model? Is both your team and marketing and development team familiar with it? Is it based on real users, or what people think are real users?

2. Pick a reference configuration for your current system. What are the various values for the performance factors described above?

3. What was the last article or book you read about usability?

4. Where could improved usability help you improve your product or service the most? What is the potential economic impact of this improvement?

5. What kinds of feedback does your system provide? How do you know that this feedback is helpful?

6. Ask your marketect for a copy of a recent RFP and how they responded to it.

# 11. INSTALLATION

Most software must be installed in order to run. Unfortunately, many development organizations defer dealing with installation and installation issues as long as they can. When they do get around to handling installation, the results are often perplexing and error prone. The economic implications of poorly designed installation process is very real, ranging from tens of dollars for each support call to thousands of dollars spent on professional services fees that could be avoided if the installation process was easier. Indeed, one of my reviewers pointed out that you can *lose* a sale based on an onerous evaluation installation. In this chapter I will consider some of the unique challenges associated with designing and implementing a good installation process. In the next, I will cover upgrades.

**Your Cost of Poor Installation**

Our enterprise class software worked great when the server was *finally* installed. I emphasize "finally", because our initial server installation process was so complex and error-prone that it required sending at least one, and usually two, members of our professional services team to the customer's site to install the system and verify its initial operation.

We didn't think this was a problem because we charged our customers for installation support. Once we created this "standard" charge, we didn't think about it anymore. Fortunately, our Director of Customer Support challenged our assumptions and analyzed both the real and opportunity costs associated with our poor installation process.

The real cost calculation showed that we didn't make any money by charging for the on-site professional services. Most of time we broke even. The rest of the time we lost money. Even if we didn't lose any money, however, we certainly lost an opportunity to improve customer satisfaction. The opportunity cost, which measures the alternative uses of a given resource, presented a more sobering analysis. Simply put, was it better to have a member of our extremely limited and valuable professional services team doing "routine" installations or some other activity, such as a custom integration? In this case, common sense was supported by reasonable analysis. The opportunity costs of our poor installation process were substantial. We took it upon ourselves to improve our installation to the point where a customer could install the system with no more than one phone call to our technical support team. We eventually achieved this goal, improving the satisfaction of our customers and our profitability at the same time. Perhaps, if I'm lucky, someone from Oracle will read this chapter!

## 11.1 OUT OF BOX EXPERIENCE

Usability experts have coined the term "out of box experience" ("OOBE") to describe the experience a person has using a computer or other device for the very first time. Some companies excel in creating superior "out of box experiences", mostly because they *care* about being excellent in this area. Apple computer has a well-deserved reputation for creating superior "out of the box" experiences. The first was with the original Macintosh: Simply take it out, plug it in, turn it on, and you're ready to go. This great OOBE has returned with the popular iMac series of computers.

In software, the "out of the box" experience begins when the person who is going to install the software first acquires it. This could be through a purchase at a local store, with a box containing a CD-ROM or DVD. Alternatively, and increasingly, it could be when the user downloads the software directly onto their machine, either through a more technical process, such as secure ftp, or through a more user-friendly distribution process, as through Aladdin Knowledge Systems download client.

The "out of the box" experience continues as the customer attempts to install the software. For common packaged software, such as video games or basic productivity tools, installation often happens automatically when the user inserts a CD-ROM and an autorun facility automatically begins the installation process. For high-end professional software, installation can be substantially more complex, with the installation process possibly requiring one or more programs to be executed to verify that the software can be installed properly (more on this later).

It is helpful if the marketect establish some clear goals for the tarchitect regarding the installation experience. A useful goal might be that an "average user" should be able to perform the installation process without having to make any phone calls to technical support. Of course, the definition of an "average user" varies substantially based on the kind of software that you're building. It could be a schoolteacher, with basic computer skills, or an MCSE-certified system administrator, with extensive knowledge of operating systems. For those companies that have adopted use cases, I recommend adding at least one use case capturing the installation of the software.

## 11.2 OUCH! THAT MIGHT HURT

Humans, myself included, are so pain averse that we will often attribute pain sensations to potentially painful events. This, in turn, causes us to fear the event. Even if the event does not happen, we may still claim to have experienced some degree of mental anguish. Pain indeed! Understanding this concept is helpful to understanding how your customers may be approaching the installation experience. Here are some fears that customers have expressed to me about software installations.

- *Too hard.* Many users perceive the installation process as too hard. They are justified in this belief when the installation program advises them to do things that sound dangerous ("Shut down all applications before continuing or you will irreparably damage your computer") or ask them to do things they do not understand. In older days, installation was also physically hard, especially on personal computers. I remember shuffling as many as 30 high density floppy diskettes to install a large software program! Fortunately, the physical difficulties have largely been removed as the software industry has moved to using CD-ROMs, DVDs, and the Internet to distribute software.

- *Too complex.* The simplest installation is just copying the right file to the right location on your computer. In today's modern operating systems, something as simple as this just doesn't work anymore. Proper installation of a software program usually requires a sequence of complex steps to be executed just right. The entire process can fail if any individual step is not done absolutely perfectly! Mitigate this by providing both "typical" or "standard" options as well as more advanced or "custom" options.

- *Too easy to break something.* A major installation fear concerns the state of the system if something does go wrong during the installation process. I've experienced too many software applications that leave the system in an unknown or unstable state if the install process fails to complete (for whatever reason). This is unnecessary, and is the result of a sloppy or careless installation program. Because it is unnecessary, it is unacceptable.

- *Unknown amount of time.* Users often express frustration because they don't know how long a "typical" installation will take. As a result, they can't plan various activities around the installation process. For example, if I know that installing some software will take more than 20 minutes, I will probably wait until lunch before I begin unless I need the software right away.

- *I've already entered that data!* There are few things as frustrating as an installation process that requires you to enter the same data multiple times, or one that requires you to fill out an endless number of forms only to find out later that you forgot to enter some important code, such as a product serial number. Get all of the answers you need up front. Your users and technical support organization will thank you.

## 11.3 INSTALLATION AND ARCHITECTURE

The various definitions of "software architecture" provided in chapter 1 are not strongly correlated with the concept of installation or the installation process. I don't really have a problem with this, because the primary reasons "architecture" is of concern deal with such things as efficiently organizing the development team or choosing the best tarchitectural style for a given problem or problem domain. That said, there are a variety of ways that marketural or tarchitectural forces and choices influence the installation process.

- *Managing subcomponents.* In a component-based system, you're going to have to make certain that all required components are present, in the right version, and configured the right way, in order for your software to work. Quite often this means making tough choices about whether or not you should install a component or require that the user install a component. I've done both.

- *Technology in-licensing requirements.* Some technology license agreements (see chapter 10) have terms that govern the installation of the technology. You may be contractually required to use their installer. Of course, the license agreement could state exactly the opposite, and require you to write your own installer subject to the terms of the agreement.

### We Pick Up After You're Done

The specific technologies chosen by the tarchitect can affect the installation process in a number of ways. Consider the effects of the persistent storage strategy. It is common in well-designed tarchitectures to partition persistent storage concerns into a separate subsystem, both logically and physically. If a relational database is used for the actual storage, quite often the only configuration data that be needed is the database connection string, a user id, and a password – provided the database has been properly installed on the target system, the appropriate permissions have been set, and the database is available for your application's use! If it isn't, you're going to have to figure out how to get that database installed.

In one application, in which we supported SQLServer or Oracle, we created a two-step installation process. The first step required the customer to install and configure a variety of software, including the database, before they could install our software. To help our customers complete this first step, we provided detailed instructions on how to configure each required technology.

The second step was installing and configuring our software. To make this easier we followed several steps outlined in this chapter, including creating a pre-installation program that ensured each required technology was properly installed and configured.

- *License agreements.* It is common for your legal department to want to associate some or all of the license agreement with your installation program. Commonly referred to as "EULAs", for "End User License Agreement", you need to make certain you're including these as part of the installation process in the manner your legal department requires. For example, you may be required to present a "scroll to the bottom and click" screen that presents the license agreement to the user and halts installation until the user indicates acceptance (usually by clicking on some button).

- *Business model.* Certain kinds of licensing models, such as per-user volume licensing (see chapter 9) track the "installation event" for the purpose of "counting" the number of licenses that have been consumed. Other licensing models that track access or use of a component may affect the installation process. For example, many large applications have many smaller features that are only installed when the user attempts to use the feature ("on demand installation"). More generally, your business model can affect the manner in which you've created your installer, and, your installation process can make supporting your business model easier.

- *Partitioning installation responsibilities*. A general tarchitectural concern is the partitioning of responsibilities among various system components. As the capabilities of these components change, the set of responsibilities that we may ascribe to a component may also change, which ultimately can affect the partitioning of components in the delivered system.

  The increasing sophistication of installation programs, such as InstallShield's Developer, provide a great case study. In the early days of MS-Windows, writing a really great installation program required the tarchitect to create a variety of components to do such things as check for the right versions of key files, necessary disk space, and so forth. These capabilities, and many more, are now provided by these installation programs. Learning to let these programs do as much work as possible is a practical strategy for handling many complex installation tasks.

- *Installation environment*. The environment that you're installing into, as well as the environment that you're going to support, can have a substantial impact on your installation architecture. Consider an email management tool for the growing market of knowledge workers who work from their home office. If the tool is really useful, it is inevitable that corporate users will want to adopt the tool.

  While both of these users may require the same set of core features, the context associated with each of these target markets is very different. This difference will be expressed in a variety of ways, including the installation process.

  The home office user may be content to install the software via a CD-ROM, a DVD, or through an Internet download by answering a few simple questions and storing the software in a location that works best for them. The corporate user, on the other hand, is working in a very different environment. Their machine is likely to be under the control of the corporate IT department, and the system administrator responsible for the corporate users machine will probably want to exert a great deal of control over the installation process. Among other things, they may wish to install the software from a centrally controlled server on the internal network; they may wish to control one or more of the parameters associated with the installation, such as where files are stored; and they may wish to control which specific features are installed. When designing your installation process, make certain that you account for the environment of the target market.

## 11.4 HOW TO INSTALL

A general algorithm for installing software goes something like this:

Installation data collection & pre-condition verification

Installation

Post-installation confirmation

Let's review each step in detail.

### 11.4.1 INSTALLATION DATA COLLECTION & PRE-CONDITION VERIFICATION

In this step you ask the person responsible for the installation for all the data that will be needed later in the installation. When I write "all the data that will be needed later in the installation", I really mean it. There's nothing worse than starting a 1 hr installation process, walking away for lunch, and then return to have some subcomponent asking some minor question, with 50 minutes still to go.

To make this step easier, at the very least you should ask your technical publications department to create a simple form that captures all of the information necessary in written form. An even better approach is to capture all of this information in a file that is associated with the application, for future reference by your customer and technical support organization. Augmenting these forms with case studies of installations on common configurations is also helpful. Ask for all vital configuration information and important customization data, including such things as where the product should be installed, database connection strings, proxy servers, and so forth. Storing these data in a file enables you to easily support corporate installation environments.

Once you've collected the required data, verify as much of it as possible before you begin the actual installation. Until you've verified that everything works, don't start the install. Your verification activities should include any or all of the following.

- *Check the free space.* Although storage systems continue to make astonishing improvements in capacity, ensuring that there is adequate free space is still a very important task. It doesn't matter if you're installing into a PDA, a cell phone, a laptop computer, or a server farm. Check to make certain you have enough free space for both the target application and any "scratch" space you may need during the installation process.

- *Check connections.* If your software requires any connections to other entities, check them *before* you begin the actual installation. I've gone so far as to create installation pre-condition verification programs that simulate accessing the database or an external internet server to ensure that the information provided was accurate. The installation process was put on hold until a valid connection was established.

- *Check configurations of required entities.* If you rely on a specific database to be installed, chances are you are also relying on that database to be configured in a specific manner for your application to run. To illustrate, one of my teams spent quite a bit of time debugging a nasty error in which our software wouldn't run on SQLServer. As it turns out, the root cause of the error was an improper installation of SQLServer – the default collating sequence was exactly opposite what was required by our software! Had we checked this setting before the installation we would have saved multiple phone calls and a lengthy debugging session.

- *Check access rights.* Operating systems with sophisticated rights administration systems, such as Unix-based operating systems like Solaris and OSX, or other operating systems, like Microsoft NT (and its progeny) often require that the user be granted the necessary rights to install the software. This usually happens when the installation must modify "system" or "privileged" files. I've worked on several programs where the installation process had to ensure that the user installing the software had the proper access rights.

### 11.4.2 INSTALLATION

Once you've passed the pre-installation verification step, you can begin the actual installation. This can be as simple as copying a few files to the proper locations or as complex as reconfiguring many parts of the system. Here are some things to consider during the installation process.

- *Provide some indication of progress.* The actor associated with the installation process needs to know that the installation is proceeding. In common end-user applications, this is usually done through a progress meter. In complex, enterprise applications, this is often done by displaying information in a "command line" user interface. In either case, provide some form of feedback.

- *Provide a visual map.* Very complex applications organize the installation into various steps. Provide your users with a way of tracking progress by giving them a form or a map so that they can "check off" the completion of major activities.

- *Track progress in a log file.* Log files (see chapter 13) are a useful tool to record the operations and actions that you take during the installation process. Capturing these actions within a log files makes recovery from installation failures as well as subsequent removal of the installed software substantially easier. At the very least, your customer / technical support personnel will know what happened! Don't forget to capture user responses to questions raised by the installation process.

- *Make it interruptible.* I've noticed that many enterprise application installation processes are created under the assumption that the installer, usually an IT system administrator, is going to be devoting their full and undivided attention to the installation process. This is, at best, wishful thinking. IT system administrators are extraordinarily busy people. Instead of expecting that your installer can devote their full attention to the installation process, I recommend that you create your installation process with the expectation that the person doing the installation will be interrupted *at least* once during the installation process.

  This perspective is invaluable in helping motivate the team to choose interruptible installation options. The best way to do this is to make the installation "hands free". That is, after you've gathered and verified the information you need to install the software, everything runs without intervention once it has been initiated. The next best approach is to make the installation process "self-aware", so that the system can monitor its own installation progress and can restart itself at the appropriate place as needed. Other approaches include providing an installation checklist that the person installing the software can use to keep track of each step or explicitly breaking up the installation process into a series of smaller, easily performed and verified, steps.

- *Follow platform guidelines.* Most platforms have specific guidelines that provide guidance on how to design your installation process/program. Learn them. Follow them.

**There Are No Cross-Platform Guidelines**

One exception to the admonition to follow cross-platform guidelines occurs when you're creating software that must run under various platforms. In this specific circumstance, it may be substantially easier for the development team to write a cross-platform installer. I've done this on server-side software using Perl. For client-side software, I still recommend following platform guidelines, even if it takes more time and energy. The effort is worth it.

- *Avoid forcing the user to reboot.* If you must force the user to reboot, give them a choice as to when it will happen. While I don't actually know of any reason where you simply must force a user to reboot, at least give a choice.

- *Avoid asking unnecessary questions!* If you can resolve a question or a parameter setting at run-time in a reasonable way, do it! In many circumstances, the person responsible for the installation is doing it for the first time. As such, they are probably not familiar with your program, so asking unnecessary or even esoteric questions ("Would you like your database indexed bilaterally?) are, at best, useless, and, at worst, harmful to the operation of your system. It is far better to pick sensible defaults that can be changed later, after the user has gained appropriate experience with the system.

  If you must ask the user a question, make certain you provide sufficient reference material so that they understand the question, the effect that choosing a particular answer might have, and whether or not their answer may be inconsistent with previously answered questions. Since many people simply choose the default answer anyway, make certain any default answers really are the most reasonable, least harmful values.

### 11.4.3 POST-INSTALLATION CONFIRMATION

Installation isn't complete until you've confirmed that the software was installed correctly. This may be as simple as verifying the right files were copied to the right locations. It can be as sophisticated as executing the installed software and invoking any number of manual and/or automated tests. Unlike the installation pre-condition verification, which focused on the context in which the software is installed, the post-installation verification should focus on the actual execution of the system: Is it giving the right results? If not, why not? Make certain you log the results from your post-installation confirmation, as they can be invaluable to resolving any problems.

Once the installation has been verified, *clean up!* Many installation programs create or use temporary storage. Some do a very poor job of cleaning up their work when their finished. Set a good example, and clean up any files or modifications you've made during the installation process.

Now that things are working and you've cleaned up, you can do additional things to enhance usability and your product. Consider any of the following.

- *Display "read me" or "notes" files.* Most people probably won't read them. Write them anyway. You will gain critical insights into your product by doing so.

- *User registration.* Many user registration processes are initiated after the software has been installed. You may, for example, check for an active internet connection and automatically launch a web page that would allow the user to register their product. From a marketing perspective, you should provide some reasonable incentive for registering other than Spam!

## 11.5 FINISHING TOUCHES

Here are some techniques that I have found useful in improving software installations.

### 11.5.1 They Don't Read The Manual

That's right: people who are charged with installing your software often *do not* read the manual. This is not an argument against installation manuals. Such manuals are vitally important, for no other reason than they provide the development and QA teams with a concrete description of system behavior that can be verified. Of course, I've found that the really great technical writers can substantially improve the usability of such things as installation manuals. When something is hard to describe, it is hard to use, and the great manual writers often find ways to substantially simplify the installation process.

More generally, users do not often do many of the things that we recommend. While this may be unfortunate, it is also part of the reality that you must deal with. And, while your users may not read the full manual, there are other techniques that you can use to make installation easy. One technique that I've found helpful is to provide some kind visual roadmap of the installation process. This can be combined with the checklist described earlier. It gives the person installing the software context about what is happening as well as providing them with an overall sense of what is being done at each step of the process. A really well done roadmap can also provide an estimate of how much time each step will take, which can be invaluable in helping the person installing the application manage their tasks.

### 11.5.2 Testing The Install And Uninstall

Both the installation and uninstallation processes must be thoroughly tested. Yeah, I know that this is common sense, but I've found that many QA teams do not adequately test one or both of these process. This is a very poor choice given the adverse economic effects of a bad installation or an incomplete uninstallation. Keep in mind the following when testing these options.

- *Start testing the installer in the first iteration.* Agile development processes, such as XP, Crystal, and SCRUM, and iterative-incremental development processes, share the idea that a system is developed in "chunks". A best practice is to start the installation process very early in the product development cycle, usually by the third or fourth iteration.

  The benefits are quite substantial. Done properly, the installation process can be included as a step in your automated build process. With each build, the "actual" installation program can be run, and it can be configured to install the product correctly. This allows you to begin testing the installation, using it more easily in sales demos, customer evaluations, or even alpha/beta programs. Developing the installation process early can help shake out dependencies among system components, ensure the build is working correctly, and reduce the overall complexity of creating the installer (instead of trying to do it all at once you're doing it like the rest of the system – a little at a time).

- *Try the various options*. Your users will. Your testers should. Hopefully, an understanding of the complexities associated with testing these various options will motivate the development team to not provide so many.

- *Automate*. Complex installations alter the target machine in a variety of ways, from copying files to setting various parameters (such as registry parameters in MS-Windows). The only way you'll have a chance of reliably checking the effects of the installation is through automation. Write programs that can accurately assess the system before and after the installation or uninstallation.

- *Follow platform guidelines*. As stated earlier, modern operating systems have a variety of guidelines associated with installing software. They also have similar guidelines surrounding the uninstallation process. Make certain you're following platform guidelines *and* properly removing what you added or changed.

- *Automate*! If your product will be installed multiple times by a single organization or person, provide a way to generate automation scripts for your setup.  Most installation generators allow you to feed a "script" to the setup apps they generate so that they run completely automatically.

  Even better, provide a way for a the person doing the installs to take the log from a successfully install, and generate the installation "script" for future ones.  Anyone who has to do 200 identical installs will thank you.

## CHAPTER SUMMARY

- Installation is about money. A poor installation usually costs you more than you're willing to admit or quantify. A good installation saves you more than you're able to admit or quantify.

- Many users find installation scary. They don't understand the questions and can't often make sense of what is going on.
- Your installation has to deal with the structure of your architecture. All components and component dependencies must be handled properly.
- Make certain your proposed installation process is supported by all of your license agreements.
- A general algorithm for installing software goes something like this:

  Installation data collection & pre-condition verification

  Installation

  Post-installation confirmation
- Automate your installation process for internal testing, and make it automatable for enterprise environments.
- Test. Test. Test.

## CHECK THIS

- ❑ We have defined how each subcomponent will be handled by the installation process.
- ❑ Our installation process meets all technology in-license requirements.
- ❑ We have defined the level of skill required of the person installing the software. This level of skill is reasonable for our product.
- ❑ We have a way of verifying the correctness of the installation.
- ❑ Our installation process adheres to target platform guidelines wherever possible.
- ❑ An average user can perform the installation without referring to the documentation.
- ❑ We have tested both installation and uninstallation.

## QUESTIONS FOR MONDAY MORNING

1. Starting with a fresh computer, grab a copy of your documentation and your software. Install it. Perform a standard operation or use case. How do you feel?
2. Who installs your product? What is the definition of an "average installer" of your software?
3. Where is it installed?

4. How is it installed?
5. What is the expected/required skill level?
6. How long does it take to install your software? Can lengthy portions be easily interrupted?
7. Can you undo/remove what was installed?
8. How can you diagnose install problems?

## TODO

Talk about the manner in which installation changes over the life of the product. Early adopters, in an early market segment, are more likely to accept a tough to use installation process. This needs to change over time, and is heavily influenced by your market segment.

# 12. UPGRADE

While the previous chapter covered installation, this chapter focuses on upgrades. Perhaps surprisingly, upgrades are often more problematic than installations because they have the ability to cause customers substantially more pain. This is due to a variety of factors, including the potential for the loss of data. Handling upgrades well is one important key to not just creating winning solutions, but sustaining them through multiple releases. Given that more money is made on upgrades than on initial sales, and the fact that one of the primary functions of marketing is finding and *keeping* customers, it is surprising that upgrades receive such little attention from marketects and tarchitects.

## 12.1 LIKE INSTALLATION, ONLY WORSE

In the previous chapter, I discussed some of the fears that users of your software may experience when installing your software. Upgrading a software system takes these fears and adds an entire set of new ones. Here are some of the additional fears associated with upgrades that I've had to deal with over my career.

- *The pain associated with rework.* Many times an upgrade requires the user, system administrator or IT personnel to rework one or more aspects of the system and its underlying implementation. This rework can take a variety of forms, but usually centers on how your system is integrated with other systems. For example, I've created systems that relied on data produced by various U.S. governmental agencies. In an astonishing case of poor customer management, these agencies would rather every now and then simply change the format of the data being distributed to customers such as myself, breaking the programs we had written to reprocess this data. When this happened we had to scramble to resolve the issues. Clearly the governmental agency knew of these changes before they were instituted. The pain and cost my company, and that of other companies that relied on this data, could have been minimized or avoided had these agencies simply considered our needs before changing the format. At the very least, they could have provided us with advance warning of the changes!

- *Ripple Upgrades.* I refer to a "ripple upgrade" as an upgrade that forces you to upgrade otherwise stable system components. Examples of ripple upgrades can involve either hardware or software or both. On the hardware side, it is common for a new version of application to require more memory, a faster processor, more disk space—or all of the above! Examples of software ripples occur when you want to upgrade to a new version of an application that requires a new version of an operating system. Ripple upgrades are a painful part of technology reality. If you're going to in-license technology – and you will – than you're basing part of your future offerings on one or more in-license components. In many circumstances there is simply no alternative to a ripple upgrade. Ripple upgrades are also help you manage the matrix of pain by allowing marketects to explicitly remove support for a given configuration. What you *can* do is make the ripple upgrade as painless as possible, especially as it relates to software components.

- *Data Migration.* Data created in version *n* of the system often requires some kind of transformation to be fully usable in version $n_{+1}$. A common example concerns relational databases. New features typically require new schemas. The upgrade process must be constructed in such a way that the user can relatively easily move data from the old schema to the new schema. Remember that data migration may go in the other direction. Users of version $n_{+1}$ of the software may have to create data that can be used by users of version *n*. In this case, you will have to provide a mechanism for storing or exporting data in the older format, with the potential loss of features or capabilities that this may entail.

  My friend Ron Lunde points out that careful design of the schema can dramatically reduce the effort associated with migrating data between releases. The goal is to separate those aspects of the schema that change more frequently, or that shouldn't change at all once created, from those that may change more often. For example, in many transaction-based applications it is rare to upgrade the transaction data. Thus, carefully separating the transaction data from non transaction data in the design of the schema can substantially reduce data migration efforts.

- *Data Retention.* Old data is rarely deleted. It is retained according to some kind of corporate policy. Motivations for data retention policies include such things as the fulfillment of specific legal requirements, such as tax laws. Your customers may require you to produce a valid archive copy of the data, and you may need to verify that you can access these data for anywhere from three to seven years after the upgrade.

- *Certification.* Upgrades, especially of enterprise class software systems, must pass through a stringent set of internally defined customer certifications before they can be placed into production. This process usually takes at least two months. It often takes considerably longer, which is why it is so rare to see enterprise class software upgraded more than once or twice a year.

- *New APIs.* New APIs are a special form of rework. APIs must be carefully managed to meet the needs of your customer. Changing APIs introduces a variety of pain to customers, and usually results in customers delaying the upgrade. It is best, for everyone concerned, to avoid this pain if at all possible. Refer to chapter eight for a discussion of API management.

### Once Integrated Never Upgraded

It was a tough decision, but we knew that the changes we made to our underlying server were going to invalidate the earlier APIs. In the long run, these changes, and the new APIs, would provide much more flexibility and functionality to our customers. Since the APIs were rarely used, most of our customers were not going to be affected. Unfortunately, our analysis indicated that these changes would severely negatively affect a small number of our largest and most important customers, who had done a lot of custom integration work with the old APIs. The development team found a way to create a partial compatibility layer between the previous version of the APIs and the new version of the APIs, easing, but not erasing, the burden our customers would face as they migrated their systems to take advantage of the new APIs. Ultimately, customers upgrading to this new version using these APIs would have to make substantial changes.

Most of our customers realized the benefit of the new server, the new APIs, and decided to upgrade. Some, however, held out for a surprisingly long time! One, in particular, had spent almost $40K in consulting fees to integrate our system with another. They didn’t want to spend this money again. I knew that they were serious when they spent $20K to fix a bug in the older, unsupported version of the product. Eventually, they did adopt the new system, not by upgrading, but by converting to a managed services deployment.

- *New Features.* While customers may be excited to learn that your latest release has several new features, they may not be so excited to learn how to use them. Learning takes time and effort. It requires changing behaviors. Depending on the size of your customer and the magnitude of the upgrade the cost associated with learning new features can delay or even cancel the upgrade.

- *Inaccessible System.* Unless the software application is a truly mission critical business application, the upgrade process is likely to make the software being upgraded unavailable to its user. This can range from a minor inconvenience, as when I am upgrading a desktop application, to a major inconvenience, such as when I am upgrading my operating system, to a completely unacceptable risk, such as when my business is upgrading its Customer Relationship Management (CRM) or employee benefits system.

- *Reverting to the old system.* Change management protocols for important or mission-critical applications always define how the user or team performing the upgrade will revert to the previous version of the system should *anything* go wrong in the upgrade to the new system. Your architecture can help make reverting to the old system relatively easy… or a dangerous project.

### My Pain is Different

The problem with the list that I've presented is that it is relatively abstract. It covers situations that more than likely don't correlate with your specific circumstances. I don't particularly like this, because managing the pain associated with upgrades is quite dependent on managing the *specific* kinds of pain associated with upgrading your system.

As a result, if this chapter is to have any real value, you must create your own list of the specific kinds of upgrade pain that customers may experience in your system. Each of the companies I've worked for, and each of my consulting clients, has a unique list. (Not surprisingly, each of them also had a unique architecture!) Until you've created this list, you can't critically examine what you're doing and how you can improve it to make certain you're meeting the needs of your customer.

Oftentimes, many complicated steps must be followed exactly or the overall process will fail. The order in which these steps are performed, and the management of these steps, represent a special kind of "pain" experienced by your customer. Understanding the upgrade process by mapping out each step in the process gives you a chance to simplify it.

## 12.2 MAKING UPGRADES LESS PAINFUL

Like installation, there are a variety of techniques that you can undertake to make the upgrading process less painful. For starters, review the algorithm for installing your software in the previous chapter. Upgrading software typically follows the same steps, and this algorithm will help make certain you're not missing anything important and that you're asking your customer to do things in an orderly manner. That said, upgrading software does raise some additional issues which are best handled by the entire development team working together to make the best choices possible. Consider the following.

- *Don't upgrade more frequently than your customers can absorb the upgrade.* Customers can't absorb releases that occur too quickly. Customers get frustrated and concerned if releases happen too slowly. You have to use your understanding of the events and rhythms of your market (see the appendices) to guide the frequency with which you create and distribute upgrades.

- *Assessing upgrade readiness.* All of the steps that I recommended in the previous chapter, such as checking for the necessary disk space and verifying configuration parameters still apply. In addition, and depending on the upgrade, you may also wish to examine how the customer has used the current version of the application to determine how to best upgrade the system. Upgrades are often best handled by assessing the current installation and carefully planning the upgrade.

**Excising Features**

I've earned a reputation of fairly ruthlessly assessing the features associated with an application. Simply put, if a feature isn't really needed, *take it out*. Of course, like so much other advice, this is easy to say but hard to do. Removing a feature from an application is usually a traumatic experience, because it raises all sorts of questions: Which of our customers is using this feature? How are they using it? How will they react if we take it away? Attempting to find detailed answers to all of these questions can be costly and time consuming. Even asking the question can alert customers to a potential change, and there are times when it is best to deal with customers who are affected by such a decision in a very controlled manner.

Virtually the only way to handle the removal of a feature is during the upgrade process. Features are often correlated in some way to the persistent data managed by the application. By carefully assessing the persistent data of an existing installation, you can often determine if a feature is used by customers. If the data indicates that the feature isn't being used, then you can often drop support for the feature in a relatively quiet manner.

To illustrate, in one application we allowed our customers to associate "keywords" with certain data. While we thought our implementation was "good enough", it really wasn't. It was too hard to use and the results weren't all that useful. A more thorough analysis indicating that implementing this feature well would be much more work than originally expected. This feature wasn't as important as other features, so I made the decision to remove it. The team modified the upgrade procedure, and analyzed the database. No keywords meant no use of the feature and safe removal.

More generally, it is a good idea to write a program to carefully assess the current installation to determine the best course of action to take during an upgrade. A good assessment will let you know how the system is being used (based on persistent data) and may tell you how about any custom modifications, integrations, or extensions, and will help you identify any specific ripple upgrade requirements. Log files, configuration and customization files, and even corporate settings (such as LDAP servers) are sources of potentially useful information that can help you plan a sensible upgrade and mange the removal of features.

- *Data migration process*. It sounds simple: take data from version $n_{-1}$ and migrate it to version $n$. It might even be simple, depending on the nature of the changes that are being made in the system. Unfortunately, the real world of working systems and customer environments make things surprisingly complex.

Removing relatively esoteric persistent format choices, there are two basic formats. The first is structured data stored in a database management system. The second is unstructured or semi-structured data stored in a flat file. Data migration processes for these formats differ along a number of dimensions, including when the data has to be upgraded and how much work the customer must perform during the migration process.

For data stored in a database, you have to work out the details of the data migration process, because chances are good that all of the data must be migrated at once. This *can* be pretty simple, if you're absolutely certain that the customer hasn't modified the schema in any way. If they have modified the schema, you're going to have to find a way to handle the modifications. One option is to require your customer to do all of the work (e.g., export all of their customizations and associated data and then import these data after the upgrade). Since your customers may not want to deal with this level of work, you may have to create the necessary tools to perform these steps automatically. I've had the best results with a middle-ground approach, where developers write special tools that are given to the professional services organization for actual use in the field. It is imperative that the services organization understand these tools, because there is a very real chance that they will have to modify these tools in the field to deal with unique problems that occur only in customer environments.

Data stored in a flat file can often be upgraded on demand, such as when an application opens a data file from a previous version but stores it in a new version. It is a good practice to warn the user that you are upgrading their data to the version.

- *Upgrade configuration and upgrade information.* Like many users of Microsoft Office products, I have take the time to customize these products according to my personal preferences. Indeed, I really enjoy the ability to create such things as customized menus and tool bars. Unfortunately, these settings are not preserved when I upgrade from one version of Office to another – a very frustrating experience! Don't make this mistake with your users. Make certain that you upgrade configuration and upgrade information.

- *Which version are you migrating from?* A big problem with migrating data from a previous version to the current version is the fact that it is highly unlikely that all of your existing customers are going to be running the same exact version of the old software. If you're releasing version 3.7 of your application, you may have to deal with customers who are upgrading from any arbitrary prior release. Only a careful examination of your customer base will let you know how many previous versions you need to handle.

  There are two basic strategies for dealing with older versions, with a sensible approach often being a mix of these two basic strategies. The first is a single-step migration, in which you provide a direct path from any previous version of the software to the current version. The second is a multi-step migration, in which you pick a subset of the previous versions of the software that can be upgraded to the current version. Customers who have a version that is not within this subset must first migrate to one of the versions in the supported set.

- *Be exacting in your requirements associated with ripple upgrades.* Earlier in this chapter I described the phenomenon of a ripple upgrade, in which upgrading one or more components of your system require your customer to upgrade other components. The net effect can be extremely painful, even to the most experienced customer. You can reduce this pain by making certain you are extremely clear on the specific requirements associated with the upgrade.

- *Co-exist or replace?* A special issue that concerns upgrading existing software is whether or not the new version should completely replace one or more components of the existing implement or co-exist along side them. Resolving this issue requires a detailed technical understanding of the dependencies between system components as well as marketing's input on how what they seek in the user experience. While successful systems have used both approaches, I recommend opting for the replace approach if at all possible. The lingering effects of the old software are very hard to manage. The net effect is that you confuse your users, who don't know when they should remove the old version (if it wasn't safe to remove it during the upgrade, when *will* it become safe to do so?).

## Which Version Do I Use?

As mentioned in chapter six, allowing multiple versions of the same application or product on the same system increases testing complexity (remember the matrix of pain?) and often increases support costs associated with version questions. For example, For example, I was confused by AOL when I installed an upgrade to their application on my laptop and found that my desktop had two nearly identical shortcuts. One referenced the old version. One referenced the new version. They looked identical. Even more frustrating was the fact that not all of the data that I had entered into my old version of AOL was transferred to my new version of AOL. I had to transfer some data by hand. Of course, I don't mean to unfairly single out AOL. All of the applications I use on a regular basis have a variety of upgrade related problems, ranging from data migration to unknown or unstated ripple upgrade requirements. Given that more money is made in upgrades over the life of a successful application than in initial product sales, you would think that marketects and tarchitects would take the upgrade process more seriously.

## 12.3 Market Maturity and Upgrades

In other chapters of this book I have argued that innovators and early adopters are more tolerant of a wide variety of issues that may exist in the immature product. They can deal with difficult installations. They can handle APIs that don't work very well. They often accept poor performance in the expectations that performance will improve in a future release. Given that innovators and early adopters are willing to tolerate so many shortcomings, you may be tempted to think that they will also tolerate a poor or faulty upgrade process.

My experience tells me that this is not the case. One area in which innovators are just as demanding as the late majority is in upgrading their software, *especially as it relates to data migration*. Simply put, you cannot screw up their data. Innovators may put up with a lot, but trashing their data is beyond even their considerable level of patience.

Note that this is not an argument for creating an easy to use upgrade process (although this will probably help you!). It is simply a blunt statement of fact: An upgrade must never result in a catastrophic data loss.

## Chapter Summary

- Upgrades can cause considerable customer pain in a variety of areas. Make certain yours doesn't.
- Ongoing technology evolution motivates ripple upgrades. Detail these requirements carefully.
- Don't ever screw up your customers data during an upgrade.
- Understand just how frequently your customers can absorb an upgrade. They may want an upgrade, but their operational infrastructure may not be able to absorb an upgrade as quickly as you can create one.
- It helps to create a variety of tools to assess upgrade readiness, the impact of the upgrade, and whether or not you can easily remove any unnecessary or unused features.
- All market adopter segments require a good upgrade.

## Check This

- ❑ We have identified all potential areas where an upgrade can cause pain and have tried to minimize each.
- ❑ We have tested that all data migration paths.
- ❑ We have defined the manner in which *any* customer with a previous version of the system can upgrade to the current version.
- ❑ We have defined how long the system will be "down" because of upgrade.
- ❑ We understand the process for "uninstalling" an upgrade, if a customer should discover a "matrix of pain" that can't be solved for a while.

## Try This

1. Find a computer that has your second most recent version of your software. Use it. Build at least a sample dataset. Now take the most recent version of your software and upgrade the current system. How do you feel?

# 13. CONFIGURATION AND CUSTOMIZATION

Complex systems need to be configured to be useful. Unfortunately, because "configurability" is not one of the more commonly discussed "ilities", systems end up much more difficult to configure than they need to be. Ease of configuration is a worthwhile goal for both the marketect and the tarchitect. In this chapter I will discuss configuration parameters, and some of the things you can do to make certain that the system is as easy to configure as possible. Following this, I will address customizability, which is closely related to configuration.

## 13.1 CONFIGURABILITY, AN ELEMENT OF USABILITY

The primary reason to "care" about configurability, which is a dimension of overall usability, is cost. Difficult to configure systems:

- are harder to use, increasing operating costs;
- are harder to tune, resulting in lower actual and perceived performance, which can increase costs by forcing customers to purchase unnecessary hardware;
- result in more calls to technical support, increasing technical support costs;
- require more complex professional services, increasing direct costs to the customers, because professional services takes longer than it should, and decreasing employee satisfaction, because very few professional services people I know get excited about configuring a complex system;
- increase overall customer frustration, putting you at a competitive disadvantage.

A related, but important reason to make configuration easy is based on the architectural nature of the systems you're creating. Modern systems are increasingly built using "modular" approaches. Modular systems, in turn, increase configuration requirements. This will become increasingly true in architectures based on web services. Without a strong commitment to make configuration easy, you're going to increase the likelihood that configuration costs increase over the life of your application. This is *not* a pleasant prospect.

"Ease of configuration" must be architected. Doing so is in your, and your customer's, best interests. In terms of usability, "configurability" means having the right number and kinds of configuration parameters – not too many and not too few, and the set of parameters that are selected affect the system in ways that are material to the user.

## 13.2 THE SYSTEM CONTEXT

Before worrying about the technical details of structuring configuration parameters, let’s take a step back and consider what should be configurable. The basic information that needs to be captured in configuration parameters is the *system context*, a “catch all” phrase designed to identify all aspects of the contextual information you need for a properly functioning system. Identifying this information is the job of the tarchitect, as informed by key members of the development team. In this regard, Here are some specific areas to consider.

- *Location of key files and/or directories*. Many complex systems rely on a variety of pre-defined or well-known files and/or directories. Quite often these locations are set during the installation of the system and captured in configuration files for the run-time environment.

- *Bootstrapping / initialization information.* You might argue that such things as the location of key files should be obtained from a directory service. In many cases, this is true, and it is good to see the continued growth of directory services. That said, even if you’re using some kind of a directory service, most systems need some kind of bootstrapping information to obtain a “root” or “stable” address for the bootstrapping.

- *Portability switches.* A special kind of context is any context that deals with the portability of the system. For example, a team that worked for me designed a system that could be configured to use with Oracle or SQLServer. The choice was important, because the system used slightly different SQL statements that had been tuned for the specific performance characteristics of each database.

- *Compatibility controls.* Development teams often face challenges in determining how to support older versions. Instead of arbitrarily choosing how to do this, it is often helpful to turn the decision over to your customer and let them decide via any number of configuration parameters.

- *Performance parameters.* A variety of performance parameters exist for systems, ranging from timeout values, to in-memory storage requirements, to the number of database connections the system should automatically maintain. Truly sophisticated applications can adjust these parameters based on self-monitoring system performance, but most of us can get away with allowing the user to configure their systems with simpler parameters specified at system start-up.

### Too Much Of A Good Thing

It started with a well-intentioned development team trying to find a peaceful resolution to different opinions on everything from how much default memory should be allocated to the application to the default layout of the user interface. By the time they were finished, I feared that I would need to create an expert system just to help a customer install and configure our system!

In this case, we had too many optional configuration parameters. In isolation, each of the configuration parameters made good sense. In total, they presented an almost bewildering array of choices to the system administrator. We managed to create a workable solution by creating very sensible groupings of parameters, with rock-solid in-line help and plenty of examples on how changes to the parameters would affect system performance. But I learned my lesson, and am more cautious about simply putting something into a configuration parameter.

A good rule of thumb is that something should only be a configuration parameter if the system cannot function without it being set, such as a proxy server. If something can vary in a sensible way, or if you simply want to provide your users with greater flexibility, make it a customization parameter and provide a sensible default. The difference is subtle but important. Configuration parameters have to be set correctly for the system to operate, which usually means they must be set by a person with sufficient knowledge to set the proper value (e.g., a system administrator.). Customization information, which should be an attribute of some object (discussed later in this chapter), may never be set at all. I find the default setting of “Windows Standard” for the desktop experience just fine.

By the way, in case you think I'm being overly dramatic about creating an expert system to manage configuration parameters, keep in mind that one of the most famous expert systems ever created – XCON – was created specifically for the task of configuring DEC VAX computers!

## 13.3 INITIALIZATION VS. EXECUTION

There are two basic times when a system needs to be configured: before it begins execution and during its operation. As discussed in the previous section, much, if not all, of the information associated with configuration parameters is processed during initialization. In addition to reading these data to obtain the required context information, the system must also handle gross errors and/or inconsistencie s in configuration data. A proven approach is to ignore the inconsistency, choose a sensible default, and log the error. If you simply can't ignore the error, stop processing as soon and inform the user.

One drawback of requiring configuration data to be processed during system initialization is that you may require the user to shutdown and restart a system for seemingly trivial changes. This may be acceptable for simple systems, but is likely to become intolerable for complex systems. For example, log data, which are discussed in greater detail in chapter 14, are often best constructed if they can be turned on or off while the system is running. Another example deals with configuration data associated with high-availability systems. Specifically, high-availability systems need to be designed so that there is either a way to notify the system of important changes to configuration data, or the system can self-discover important changes during its normal operation. A key consideration is any data that may be useful for problem diagnosis – it is always best to gather the data needed while the system is running, at the time the error is occurring.

A special case of complexity in configuration parameters deals with pass-thru parameters, or parameters that are set in the context of your application but are really passed through to in-licensed components. Suppose, for example, that your system relies on an third party text searching engine. The structure of its configuration parameters are going to constrain the structure of yours, including whether or not you can set its parameters while your system is running. Because of this, the tarchitect must consider which configuration data are best handled during initialization and which are best handled while the system is operating.

## 13.4 SETTING THE VALUE

Once you've determined the values that need to be configured and have thought about when they need to be configured, you need to consider who can set the value. I've found that three entities are common: A human, such a system administrator; another system, as when two systems negotiate a value or receive commands for inter-operation; or the system itself, when it is architected to auto-configure one or more values. A system can rely on any combination of the three to get the job done.

It is assumed that the entity that is setting the value has been granted the necessary authority to make changes to configuration parameters. This does not mean that every entity should be given authority to change every parameters! Make certain that each parameter, or each class of parameter, is reviewed to make certain the full ramifications of changing it are understood.

You may not want any individual user permission to change any given parameter. This is usually accomplished by requiring that configuration data be stored on a system with administrative passwords, but this is not a guarantee that the right system administrator will be changing the right value. If needed, you may want to consider creating an audit trail of these data. The easiest way to do this is to put these data into a database, since a database will provide you will all of the facilities you need to manage the audit. Unfortunately, later on this chapter I will recommend storing configuration data in easily accessible, human readable, files. These are conflicting recommendations, and you'll have to choose what is best for your situation (no one said being a tarchitect was easy).

Interesting challenges can emerge when one or more the entities described above is granted authority to change the same parameters. You'll have to choose under which circumstances one entity can override the other. This can be a sticky problem, and it is best to involve the marketect in these discussions. The best choice for a given solution can change depending on how the marketect wishes to approach their market. To illustrate, I've created applications in which the number of active database connections that could be set by either the system administrator or the system itself based on an assessment of its own performance. In our case, we allowed the system to adjust the value automatically in case the administrator set it to an absurdly low or absurdly high value.

## 13.5 SETTING THE RIGHT VALUE

In discussions regarding configuration parameters, the tarchitect and marketect must work together to meet the needs of the user who is both affected by the parameter and the needs of the person (or system) who will be setting that parameter.

In general, most users are not aware of the full range of configuration parameters that may exist for their application. Nor do they need to be. To illustrate, in writing this chapter I did a quick search on my computer for all files with the extension "INI" (I'm writing this book on my trusty laptop running MS-Windows 2000). I had expected to find two to three dozen. Imagine my surprise when I found more than two hundred! A cursory review of many of these files revealed that most of them weren't really storing configuration parameters of the kind I'm talking about in this chapter. Instead, most of them were using the facilities provided by MS-Windows as a simplistic form of persistent storage.

Some, however, were clearly designed as configuration files in the spirit of this chapter, with various settings intended to be modified to meet the needs of the user. In considering the needs of this person, remember that increasing the number of configuration parameters increases the likelihood that the user might set the values to an improper or useless value and generally detracts from usability because of the increased complexity in trying to understand every parameters.

The previous sections provide you with a guide of the information you need to provide to the entity setting the values. Simply put, they need to know what values to set, how to set them (including formatting and valid values), why to set them, and their effect on the operation of the system. Put the answers to these questions within the file, and not in some external document or hidden on some web site. Let the person who is going to use the file understand what is happening.

The `caps.ini` file, distributed as part of the Microsoft Platform SDK provides almost all of the answers that are needed by a user who wishes to modify these values. Here is an extract from this file, under the section `[CAPS FLAGS]`. The information could be substantially improved by adding some information as to effect of setting each of these parameters "on" or "off".

```
[CAP FLAGS]
# CAP accepts the following parameters:
#    *  profile        = (on/off)
#    *  dumpbinary     = (on/off)
#    *  capthread      = (on/off)
#    *  loadlibrary    = (on/off)
#    *  setjump        = (on/off)
#    *  undecoratename = (on/off)
#    *  excelaware     = (on/off)
#    *  regulardump    = (on/off)
#    *  chronocollect  = (on/off)
#    *  chronodump     = (on/off)
#    *  slowsymbols    = (on/off)
#
# Anything value other than on or off is encountered
# and the default values will kick in:
#             profile        = on
#             dumpbinary     = off
#             capthread      = on
#             setjump        = off
#             loadlibrary    = off
#             undecoratename = off
#             excelaware     = off
#             regulardump    = on
#             chronocollect  = off
#             chronodump     = off
#             slowsymbols    = on
#
# Please notice there are no spaces between the keyword
# and the value (either 'off' or 'on')
#
```

While the instructions provided in this document are clear (at least to me), they do contain some minor grammatical mistakes. Because mistakes like this do occur, it is useful to ask your technical publications department in the complete documentation of the configuration file, including its internal documentation. They should not write the initial description of these parameters – that is the job of individual developers as reviewed by the tarchitect. They should be involved in the final review process, because they are trained in how to structure and communicate important information in a precise manner.

## 13.6 Configuration Parameter Heuristics

I've found the following heuristics useful when designing configuration parameters.

- Make them easy to change, even when the rest of the system is down. This means that they must be stored externally, in a simple, easily managed, non-binary format.

- Store all of the data in one location. If you *must* have configuration data in multiple locations, go find a person that you can trust and convince them that storing the data in multiple locations is a good idea. If both of you continue to think it is a good idea, then go ahead and do it.
- Store the file in an obvious location with an obvious name. I recommend names like “SYSTEM-CONFIGURATION-DATA” store in the root directory of the installed application.
- Platform-specific file formats such as “INI” files are OK, but why not use XML? It is simple, as verbose as you need, and easily managed.
- Be careful of such things as registry entries. They aren’t portable, and can be difficult to change for non-technical users. In general, I don’t perceive a lot of value from the use of the registry.
- Make configuration data easy to capture and forward to technical support. It is amazing the number of problems a sharp technical support person can solve once they understand the how the system has been configured.
- Make it hard to get the values wrong. And if they are wrong, notify the user as soon as possible and either stop execution or continue execution with sensible defaults.

I’ve found that the most resilient designs are based on the idea that configuration parameters are persistent attributes of some entity or object. This enables other objects or subsystems to deal with the information contained within this object, and not with the details of how to manage INI or XML files. Because these attributes can often be affixed to many different objects, the tarchitect should consider that any of the following may be useful:

- A system context object, for storing information about the overall context of the system;
- "per computer", "per service instance", "per user", or even "per selectable user profile" objects;
- other objects that capture the semantics associated with the configuration data.

Just remember that while you don't have to get all of the configuration data you need right at the beginning of the system, retro-fitting can be a cumbersome process, and may require architectural changes.

## CHAPTER SUMMARY

- Configuration is an essential aspect of usability. If your system is difficult to configure, your customer won't be able to use it to its fullest capabilities.
- Configuration parameters should be designed to capture the system context, which is all aspects of the contextual information you need for a properly functioning system. Examples of this information include the location of key files and directories, portability switches, and compatibility controls.
- There are two basic times when a system needs to be configured: before it begins execution and during its operation. Creating a system that can process changes to values while it is running is often helpful during debugging.
- Customers need proper support in setting the proper value. Provide it to them.

## CHECK THIS

- ☐ We have defined all of the systems configuration parameters. For each parameter, we have defined its security and auditability requirements. For each parameter, we have defined whether it is used only during initialization or if it can be changed while the system is running.
- ☐ We have documented how to set each value and have provided guidance for setting the right value.

## TRY THIS

1. What is your process for choosing when to add new configuration parameters? Who in the team is allowed to do this?

2. Where is the documentation for your configuration parameters stored? Can your users simply open up the configuration file and obtain everything that they need?

3. How tolerant is your system when the wrong values are set? What happens?

# 14. LET THERE BE LOGS

You just hit your favorite web site. It takes a bit longer than normal to download the start page. Chances are you're wondering if the system is "hung" or if it is just responding more slowly than usual because of increased load. Chances are the system administrator of your favorite web site is wondering the same thing. And unless someone created the system so that it can log or otherwise display its operational status and related performance data she/he may never know.

Sometimes you'll find a reference in a book or paper that admonishes you to construct systems that can log their activities. This is good advice, because well designed logs can help you perform a variety of tasks crucial to the marketectural goals of your application. This chapter explores logs in a detailed manner, covering such topics as the purpose and audience of log data, the specific kinds of content usually contained within log files, and log format and management.

**Better Late Than Never**

Sometimes the schedule pressures associated with getting the release done on time can consume all of your energy. Instead of carefully thinking things through, your overwhelming motivation is to just *finish the $@#%&@ software*. Since it can be pretty hard to pull yourself out of this line of thinking, it helps to have senior developers who will.

We were just completing our first ever release of a brand new enterprise class system. I wasn't thinking about log files – at all. Fortunately for me, and our customers, one of my senior developers, Dave Smith, was. He took it upon himself to add some of the foundation needed for reasonable log files. Myron Ahn, another senior developer, extended the log file format and structure, and made sure that many parts of the system were logging a variety of useful information. I can't claim that this first release of our system was a model of "logging tarchitecture beauty", but I do know that it served our initial needs very well, and formed a very useful foundation for future logging data in future releases.

The importance of this lesson has served me well over time. I've added logging, or have substantially enhanced logging, in any number of systems since then. Each time the results were worth the effort.

## 14.1 I WANT TO KNOW WHAT'S HAPPENING

The fundamental motivation for logging data is that someone wants to know something about the system. Table 14-1 captures the bulk of what people want to know.

| Category | Purpose | Audience |
|---|---|---|
| Debugging / Error Logs | Provide information on the operation of the system as an aide in debugging. | During construction, developers are clearly the primary audience. After the system is released, technical support and professional services personnel can all use this information to resolve problems experienced in the field. |
| Error Recovery | Capture information that can be used to restore the working state of a system in case something crashes. | Almost exclusively the domain of the application. An example of such a log is a database management system transaction log. I do not advocate the creation of extensive error recovery logs, especially for persistent data. Use a database instead. |
| Performance Tuning | Provide information on one or more aspects of performance, usually with the goal of improving performance. | Professional services organizations; IT organizations within customer sites; end users. |
| Capacity Planning | Provide information on the actual amount of resources consumed during the operation of the system. As you consider capacity planning, remember to include the requirements of the logs. | IT organizations within customer sites; end users. |
| Behavior Tracking and Auditing | Provide information on how various actors (end users, system components, other systems) interact with the component in | Marketing organizations (for user/feature profiling); Security organizations (for audit trails). Note that |

| Category | Purpose | Audience |
|---|---|---|
| | question. | in these cases the content of the log may be similar but the manner in which the log is used is quite different. |
| System configuration management | Provide information on the contextual information associated with the system. These logs are often symmetric with the information contained within configuration files. | In a way, this category is a subset of debugging information, and the same audience is interested its results and uses. |
| Operational status | Provide information on the current operational status of the system (e.g., how many transaction requests are in the queue, how many users are connected to the system). | IT organizations; professional services. |

**Table 14-1**

Developers often misuse log files by storing data of one category in a log designed for another. They store debugging information within the operational status log file, or behavioral data in the system configuration log file. These mistakes, whether intentional or not, dilute the value of the data, invalidate or seriously complicate post-processing analysis, and contribute to overall system complexity.

A more serious abuse of log files is when a developer uses them to store non-logging data. The most flagrant abuse I've ever experienced was when one developer used a log file as a way to store persistent data! A good rule of thumb is that log files are generally "read only", in that one system generates log data that other systems consume. There are exceptions to this rule, such as when a system can process its own log data (e.g., I've worked on systems that can "self-tune" key performance parameters based on log file analysis), but these exceptions are fairly rare. Application processing of log files should be completely optional: if someone deletes a log files and your application crashes, you've got a design problem that needs to be fixed.

## No Categories Are Sometimes Better Than Two

The categories described above are not hard boundaries. Sometimes you're just not certain where log data should be stored, because the data could be used for more than one purpose, you're not certain who will be using the data, or how they will be using it. One option for handling this situation is to store these log data in a single log file that has appropriate identifiers for each type of log event. Post-processing tools can then read this single file, obtaining the data they need.

To illustrate, in one application I worked on the system administrator specified the locations of various required files required at initialization and using during operation in a system configuration file. As the system processed these files, it would log the results of each processing step. These log data were used in several different ways:

- Developers used them while creating and debugging the system;
- Technical support used them to help customers recovery from a system crash;
- Professional services used them to help in performance tuning and capacity planning.

We didn't spend a lot of time debating where we should store these data. Instead, we just stored these data in a single file.

## 14.2 NOT JUST THE FACTS

Within the categories described in the previous section there is a wide variety of information that can be included within a log. Table 14-2 suggests, in greater detail, some of the things that you may find helpful to add to a log file according to each category. As you review this table, keep in mind that this is one of the times that "Murphy's law" really does apply. Specifically, you may not be able to anticipate the real information you need in a log – until you need it. To combat this problem try to create log data in with sufficient contextual data to make full and complete sense of what is happening. In other words, if your logging information about a complex transaction, don't just log when the transaction started and when it stopped: log each processing step so that you can understand transaction states. Alternatively, you can adopt an approach common in rule-based systems, in which the system should be able to "explain" its behavior.

| **Category** | **Suggested Contents** |
|---|---|

| **Category** | **Suggested Contents** |
|---|---|
| *All Categories* | • Log entries should include a date/time stamp.<br>• Consider including information that can identify the user. This is usually required for logs used for auditing purposes. This may be expressly prohibited for logs used for behavioral analysis. Know the difference. |
| Debugging / Error Logs | • Error logs need quite a bit of contextual information to be truly useful. Consider the following entry, obtained from MS-Windows Dr. Watson error logging program (DrWtsn32).<br>`*----> System Information <----*`<br>`     Computer Name: LUKELAP`<br>`     User Name: Luke Hohmann`<br>`     Number of Processors: 1`<br>`     Processor Type: x86 Family 6 Model 6 Stepping 10`<br>`     Windows 2000 Version: 5.0`<br>`     Current Build: 2195`<br>`     Service Pack: None`<br>`     Current Type: Uniprocessor Free`<br>`     Registered Organization:`<br>`     Registered Owner: Luke Hohmann`<br>• Inputs and outputs to each function call or equivalent. Consider multiple levels of debugging and whether or not you should record the values of parameters. Parameters need to be recorded in a sensible manner (which may be harder than you think).<br>• Consider creating logs of all main components. For example, in a "heavy" client client/server system, both the client and the server may benefit from logs. |
| Performance Tuning | • Track each command, and how long it took to execute. If possible, track both elapsed and actual processing times. |

| **Category** | **Suggested Contents** |
|---|---|
| Capacity Planning | • Maintain an awareness of system resources. In one application I worked on, the system administrator could specify the number of initial database connections in a configuration file. This is important, because database connections consume valuable resources. During operation, additional database connections were created as necessary. These events were logged so that we could analyze the system to see if performance could be improved by specifying more database connections up front. |
| Behavior Tracking and Auditing | • One of the primary uses of these log data is to prepare various reports of the operations / features that have been used, including assessments of optional features that have *not* been used. For example, many systems structure optional features as DLLs. It is easy to add information to a log file when one of these optional features are invoked. Scanning the log to identify if these features were invoked at all is trivial.<br>These kinds of reports assist marketects in a host of important activities. Highly used features may be sources of new revenue by creative licensing programs, including feature based licensing. Unused features may become candidates for removal in a future release. Alternatively, unused or little used features may become the target of marketing and/or educational campaigns designed to increase use.<br>• Logs used for auditing purposes must be constructed so that they can be trusted. |
| System configuration input and processing | • Carefully review the data contained within the configuration parameters. If any of these data are processed by your system in some interesting and useful manner, consider adding the input and the resultant output or effect to a log file. As described earlier, many configuration parameters have values that can be overridden by the system if they are set inappropriately. Resetting the value of a configuration parameter to an appropriate value should be logged for further review and analysis. |

| Category | Suggested Contents |
|---|---|
| Operational status | • Operational status logs as being most closely associated with enterprise server applications. These applications have very specific status requirements that very considerably by application. Transaction based systems may wish to log transaction states (beginning, processing, ending), cumulative requests, outstanding request, and so forth. Enterprise applications based on concurrent user business models may wish to log whenever a user logs in/out, when a user session is gracefully or abruptly terminated, failed login attempts, total number of users, etc. In this manner, operational status logs are often correlated to the business model. |

**Table 14-2**

Any log data may be provided real-time, or near-real time, to other applications. This is helpful when you want to create such things as dynamically updated dashboards that show the status of the system in a rich graphical format.

**Logs Are Not Glorified Trace Statements!**

Once the decision has been taken to create any log files, developers can begin to stuff a variety of data into the log that is generally useless to anyone but the developer. To maintain coherence and integrity of log data, especially logs used by customers and or technical support organizations while meeting the legitimate debugging needs of the team, I recommend that the tarchitect create a separate log for developer-centric data. Once this is done the tarchitect can periodically review the contents of a sample log and work with developers to make certain the right data ends up in the right log.

## 14.3 LOG FORMAT AND MANAGEMENT

### 14.3.1 LOG FORMAT

Log data can be structured in a variety of ways, including flat files, databases, direct input from the system to a formal monitoring system, such as HP OpenView or the Windows event log.

*General comments on log format.*

- *Internationalization.* Creating log data in the target language of the user can substantially improve usability. It can also score points with customers who would prefer to use their native language for all aspects of system operation.
- *Start with a time stamp.* The first entry in a log file should be a time stamp that includes the date.
- *Always add a unique identifier, traceable to the actual source location.* Log data are most useful when you know you can return to the source code to interpret what is happening.
- *Provide a way to identify transactions.* Many log entries are associated with various kinds of transactions. Once a transaction is given an "id", put this id in the log, as it is the best way to tie together all logging data associated with a single transaction.

*Flat Files*

Flat files are probably the most popular format for creating log files. They are faster and more malleable than a database and usually trivially ported. Indeed, as in the chapter of configuration management, when I did a casual search on my laptop for .INI files, I just completed a casual search for .LOG files. Once again I was quite surprised at the results! I had expected to find, at most, a dozen or so files. Instead I found over 132 .LOG files, created by programs ranging from MS-Outlook to Adobe Acrobat and my REX6000 PDA.

What follows are some principles I've found useful in flat file design.

- *Easily parsed.* The contents of the flat file should be easily parsed. A little thought here can save a whole lot of work later. A good rule of thumb is that it should be trivial to import log data into a relational database or analysis tools like MS-Excel.
- *Easily readable.* Log files that are easily parsed are often easy to read – but not always! Again, with a little bit of planning, you can create log files that are both parsed easily by computers *and* humans.

- *Sensible location.* Don't put log files into places where a user won't expect them, such as root. Put them in a sensible location, preferably one that is associated with your application or can be configured by the user/system administrator.
- *Sensibly named.* Log files should be sensibly named. While I prefer that log file names be easily understood, sometimes this is just not possible. An example is when a separate log file is created for each invocation of an application. In these cases, you may want to generate a random name for the log file (but *do* put these into a sensible location!). It is very helpful to store the date the log was created as part of the name. Using the format of "YYYY-MM-DD-HH-MM-<log file name>" has the advantage of automatically sorting log files by date (using a 24-hour clock).
- *No garbage or special characters.* Avoid putting anything into the log file that is not a "normal" character. Use Unicode characters if you're concerned about multi-byte languages.
- *Don't forget the documentation!* Log files, like configuration files, need precise documentation to ensure they are used properly. Explain what logs are created, when they're created, how they're created, what different settings will produce, and so forth.
- *Create a reconciliation or audit id.* If you're using different log files, create some kind of identifier than can tie related entries together. If at all possible, use this identifier with third party applications.

### 14.3.2 Log Management

Log management refers to how and when logs are created, updated, and/or deleted, responses to error conditions, and so forth. Consider the following.

- *Dynamic logging.* Complex systems benefit from the ability to configure logging parameters while they are running. Unfortunately, many systems, including some that I have built, require stopping the system, changing a configuration parameter, and restarting the system to begin logging certain data. Clearly, the best time to obtain log data is only when you need it. To accommodate this need, consider making logging optional, and easily startable/stoppable while the system is running, usually through privileged commands. Even better, construct your software so that *it* can decide if you need to log some data. If your fraud detector notices that something is amiss, have it log everything for awhile…

- *Per-thread logging.* Per-thread logging can be invaluable when trying to debug problems in multi-threaded systems.
- *Logging levels.* In addition to starting/stopping logging, consider making the level of detail configurable. This is often expressed as "logging levels", where you can set a number to indicate the amount and kind of information that should be logged. For example, you might set developer-logging to the highest number, and set the "default" logging to a different value. This helps prevent your logs from filling up with data that is only needed in specific or specialized circumstances.

  A complimentary approach to logging levels is to provide labels for the various kinds of logging data. These labels can be used in conjunction with levels to create a very flexible system. Examples of labels include:

  - debug: usually reserved for extended data; turning on when conducting error diagnosis;
  - info: provides noteworthy information;
  - warning: logs an entry when a potential error condition has been detected, such as when the system has detected a lower than desired amount of a critical resource;
  - error: logs an entry when an error has occurred.

  Logging levels and labels should be designed to work in conjunction with the categories defined earlier. Thus, a "warning" entry for performance purposes might be generated when the system takes longer than expected to process an event whereas a "warning" entry based on operational status may be generated when the system becomes low on memory or disk space. Not all combinations will make sense – there is no good definition of a "warning" or "error" for behavioral tracking and usage (unless you're concerned that a user is over-using a feature).
- *Provide APIs.* Controlling the behavior of the system through logging levels and labels should be available through an appropriately designed set of APIs.
- *Configurable syntax.* There are times when a different format is easier to process. To illustrate, webserver logging facility, like those found in Apache, are completely configurable: You're in complete control of the syntax and the contents.
- *Exceptions should be easy to log.* Exceptions and logging a very closely related. Log every exception.

- *Remove when clearly not needed.* Many logs outlive their usefulness, unnecessarily cluttering machines and potentially confusing log consumers. If you store the date the log was created as part of the filename, removing unnecessary logs becomes that much easier.
- *Security.* Some log data may be sensitive. You may need to consider encoding log data or storing it in a privileged location. Depending on the specific content of the log, customers may not be willing to share it with your company.
- *Automatic forwarding.* Conversations between technical support and customers can become pretty tense when something is going wrong. Asking your customers to collect and forward log files to forward to your support organization may result in additional frustration: non-technical customers may not know where log files are located, they may not know which log files are needed, and they may not understand the proper procedures for sending these files. To make things easier for your customer, consider providing facilities to automatically capture and send log data to your technical support personnel. You'll not only get better results, you'll get happier customers.

### 14.3.3 Logging Standards & Libraries

There are a variety of logging standards that are either platform independent (such as the W3C Extended Log Format or the Common Log Format for web servers) or platform dependent (such as Windows Event Logs). Basing your log files on these standards enables you and your customers to use a wide range of freely available analysis tools. There are also several logging libraries available, such as log4j, and you should only roll your own logging facilities, formats, or external viewers after you've proven that freely available tools won't work for you.

## 14.4 Post-Processing Log Data

Simply creating the necessary log data is often insufficient to meet the needs of the users described at the start of this chapter. Many times additional post-processing tools are needed to make sense of log data. Consider any of the following tools.

- *Compaction services.* Log data can quickly consume a *lot* of disk space. Web servers on high traffic web site can easily generate hundreds of megabytes of data per hour. Sophisticated tools, such as compaction services, can substantially reduce the amount of disk space required by the log. Simpler approaches, such as a rolling log, in which data is maintained only for a specified reason and then replaced, can also be used to keep logs under control. An example of a rolling log might be only storing the last 60 minutes of data within a log.

- *Synchronization tools.* Logs created by several different applications may require synchronization in order to produce a useful result. Writing these tools can help ensure that logs will be synchronized in the proper manner.
- *Log viewers.* It often makes sense to create tools that view the contents of logs for users. Of course, if your using system management tools, such as the MS-Windows Event Manager, these tools are handled for you.

**Be Careful About Repurposing Log Data**

Users often discover a variety of uses for log files. Unfortunately, these uses can subtlety conflict with the original uses of the log file, causing any number of problems. In one system I worked on we recorded various user operations for performance purposes. It was suggested that we could also use these data for billing purposes. While it appeared that all of the data was available, this would have been a dangerous choice. Because the logs were simple flat files, they could be easily changed by the user, completely changing their bill. Clearly, this was inappropriate, and a different tracking mechanism was implemented to meet the needs of the billing requirements.

## 14.5 LOGGING SERVICES

I've found it helpful to think of logging as a "service" that is provided to the development team that abstracts the complex issues associated with implementing good logs. The actual implementation of a logging service varies from architecture to architecture, but in most cases it can be realized as some form of a Singleton. Implementing the log file as a centralized service has the following advantages.

- *Internationalization.* Within the source code you can invoke the logging service with the specific "code" that is associated with the log entry in question, passing in any additional data as necessary. The logging service can then use this code to look up any internationalization of the log entry and write these data to the log file.

- *Unified timestamps.* The logging service helps ensure that all timestamps make sense. In a multi-threaded environment, it is possible for log entries to occur “out of order”. A centralized logging service helps prevent this.
- *Flexible destinations.* The logging service can choose the destination of the log entries at run-time, freeing developers from having to make these decisions.
- *Consolidation across multiple instances or servers.* If you’re operating in an environment that could include multiple instances of your application running on one or more servers, consider augmenting log file names or log file data with additional information that would allow you to disambiguate various data. Examples of such data include the process or thread id, and the IP address of the host machine.

## CHAPTER SUMMARY

- Logs are created to help provide information or aid activities in the following areas:
    - debugging;
    - error recovery;
    - performance tuning;
    - capacity planning;
    - behavior tracking and auditing;
    - system configuration management;
    - operational status.
- Logs must be constructed from the perspective of the *consumer of the log file*. This often means you should add additional, contextual information to the log file to aide them in its use. It also means you should construct a log file so that it can be easily analyzed.
- Assess the operational impact and/or environment of your log files. Make certain you can handle and manage various error situations that may arise (like out of disk space).

## CHECK THIS

- ❑ We have defined the purpose of each log file.

- ❑ We have confirmed the utility of each log file with its intended audience (i.e., it contains the right data in the right format).
- ❑ We have taken care to remove all developer-specific debugging information from log files.
- ❑ Our log files follow the same guidelines for internationalization developed for other parts of the user interface.
- ❑ Our log files follow the same guidelines for portability developed for other parts of the system.
- ❑ Our log files are easily parsed.

## TRY THIS

1. Perform a log file assessment of your current system. What logs does it produce? Why? Who uses these logs? For what purpose? Are these logs:

   Secured? If "no", do they need to be?

   Plain text? If "no", why not?

4. How big? How much data are you generating?
5. Is any data redundantly logged (e.g. is a web server already logging some data)?
6. Are logs unique to each day? Are they rolling?
7. Obtain a sample log. Open it. Are there any "silly" entries relative to the target audience? An example of a silly entry would be including the name of a function and all of its parameters in a log file designed for behavior tracking and analysis. Such an entry really isn't useful to the marketect, unless a post-processing tool can convert it into an appropriate entry.

# 15. RELEASE MANAGEMENT

*Release management* ensures that the correct artifacts are shipped to the customers that want or need them. In order to do this, you need to identify, organize, and control these artifacts; assign them descriptive labels, and integrate these labels into the appropriate back office systems through such things as SKUs and/or part numbers.

Release management is strongly related to configuration management, which is the process of identifying, organizing, and controlling the various components and related artifacts during the creation of the system. Release management is more effective when it is based on some of the well known practices of configuration management. This chapter discusses important topics in release management, including identification, SKU's, serial numbers, and the tarchitectural implications of release management.

## 15.1 YES, YOU REALLY NEED THIS

Properly organized and executed source code and related artifact configuration management is vital to the success of any development organization, regardless of their idiosyncratic system architecture, development methodology, or implementation language. Practically, this means managing change: tracking changes to system artifacts in a coordinated way, communicating these changes to the people who need to know about these changes, and sometimes even preventing or delaying certain changes. Fortunately, we've learned many best practices in configuration management as it relates to promoting productivity within work groups (I list several good books and web sites in the bibliography).

Applications of configuration management extend beyond those associated with promoting teamwork. In a component based system, components usually don't work with every other possible version of other components. (Just ask your QA team – they'll tell you about the ones they've tested, not every possible combination). It is a lot easier for a customer or technical support to prevent and/or diagnose problems when components know their prerequisites. In message passing systems, it is helpful for messages to have version identifiers, so that changes in content or processing rules can be managed.

You need release management because your customers need it. They need to know which version of a system should be ordered, whic h version is compatible with previous versions, what patches and/or upgrades are available, which apply to their situation, in what order they should be applied, and so forth. The issues are complex, both from the standpoint of the underlying technology, and because some of the choices are made for reasons that have nothing to do with the underlying technology, as I will describe later in this chapter.

## 15.2 Establishing a Baseline

Practioners of software configuration management have defined a few basic terms that are useful in managing external deliverables. These are presented below in Table 15-1.

| **Term** | **Definition** |
|---|---|
| Program Family | The sum total of all versions of all components that comprise the product. This chapter focuses on that subset of the program family that is made available for external distribution. |
| Component / Artifact | A component or artifact is the smallest discrete entity that is identified in the system. Each component or artifact distributed to a customer must be uniquely identified and versioned. The tools the development team uses may track components at a much finer level of granularity. Some tools track each function or method as a separately versioned component. A well managed project places more than just the source code under configuration management. Interfaces, test plans, test cases, technical and end-user documentation, MRDs, and even project plans on large, important projects are all candidates for configuration management. A good rule of thumb is that any independently replaceable component should be uniquely identified. |

| **Term** | **Definition** |
|---|---|
| Version | A version is a fixed or "frozen" component or other artifact. In software, it is usually important to maintain versions of source artifacts (e.g., source code) and things that can be derived from these source artifacts (e.g., object code, API documentation, etc.), because we typically don't distribute the source artifact, but the derivations. The most important principle of a derived version is that it can be uniquely recreated from the source. You may need to establish internal version and external version identifiers. |
| Revision | A revision is a new version of a component or artifact that is intended to supersede the old. Revisions are usually linearly ordered, and are often sequentially number to reflect this ordering (e.g., as in a sequentially increasing project build number). |
| Variation | A variation is an alternative implementation of a component or other artifact. An example of a variation is a software component that is designed to perform the same identical task but on two different operating platforms, where these platforms require slightly different implementations. It is important to note that variations are *not* sequentially ordered – they are alternatives. |
| Distribution | A distribution is a version created for distribution to a set of customers. It is usually made up of multiple components and/or artifacts that have been certified in some manner. By definition, a product contains a list or configuration of the components it comprises. |

| Term | Definition |
|---|---|
| Release | A release is a named and versioned collection of components and artifacts that are generally intended for external distribution to one or more customers and that have been certified in some manner. It is what we commonly think of as the product, although recall from chapter two that marketects use a slightly different definition of a "product". A release can be simple or arbitrarily complex, and is often recursively structured. Unlike revisions, releases may not be strictly linear, especially in the case of patches to a major system. The remainder of this chapter focuses extensively on releases. Finally, I'd like to repeat the most important rule of a release: You *must* be able to perfectly recreate *anything* you send to a customer. |

**Table 15-1**

Management of dependencies or prerequisites is *central* to effective configuration management practices. Even very simple software has components that rely on specific versions of other components. The simplest way to manage dependencie s is to make certain that the release has the correct version of all of required components. In practice, this is often impossible because of licensing agreements and the fact that not every release contains all components. I'll discuss this issue again in greater detail later in this chapter.

## 15.3 RELEASE MANAGEMENT

Managing releases involves three factors: what you're releasing, who you're targeting for this release, and *their* motivation for obtaining and using the release.

- *What you're releasing.* You may be releasing something as small as a single patch to a single component or something as large as the complete product. A *full* or *complete release* is a release of the entire product, usually done in such a way that it can be installed on a fresh system or be used to upgrade an existing system. A *partial, module, update* or *fractional release* is a release of some subset of the functionality associated with the entire product that is usually designed to extend the capabilities of a base system, as when an optional module is added to an existing product. A *patch release* is a release of some subset of the product, usually designed to precisely replace one or more existing components in a working installation that have known errors. In general, patch releases should not be used to add new functionality to an existing system.

The determination of a full, partial, or patch release can be quite fluid. You might be in the midst of planning a full release with two major milestones when a competitor makes a major announcement about one of their releases. The plan is then changed to preempt your competitor by converting the first milestone into a full release and the second milestone into a partial release. These, and a host of other factors, strongly influence how a marketect decides what to release.

Full, partial, and patch releases must all be subjected to the same release processes – however you've defined them! To illustrate, most release processes require the team to do a virus scan before shipping bits to a customer. It doesn't matter if you're shipping a full, partial, or patch release – all of them must be scanned for viruses.

- *Who you're targeting.* You may be targeting internal users as an alpha release, external users in a beta, limited, or general release. A *limited*, *managed*, or *controlled release* is a release that has been targeted to a specific set of customers, such as a patch release to customers who have a bug that occurs only on a single hardware platform. A *general release* is a release that is intended for all of your customers.

  I find that marketects and development teams often confuse *who* they are targeting with *what* they are releasing. "Who you're targeting" is a about of scope. "What you're releasing" is about size. When a new virus is discovered, anti-virus vendors want to quickly update their virus definition files. What is being released is relatively small in size, but the scope is large. It might be referred to as a generally available update on the web site of the anti-virus vendor.

  I've found that many companies have trouble targeting their releases, mostly because they don't know enough about their customers. Suppose, for example, you support Solaris and MS-Windows XP and find a relatively easily fixed bug in the XP release. Unless you've kept track of which customers have Windows XP you'll have to send out the announcement to all customers. It would be far more efficient, and you could write a more compelling announcement, if you knew that the announcement was only going to be sent to the people who should get the patch.

- *Customer motivation.* Customer motivation can range from customers who actively work against accepting and installing the ("we can't install that patch into our production system ten days before the holiday shopping season – it is too risky") to customers overwhelming your web site because of the demand associated with a release ("must have it *now*"). Most releases tend to fall into a middle ground, and usually the marketect has to use a combination of carrots (new features, improved performance, reduced bugs, greater reliability) and sticks (lack of support for discontinued platforms or releases, license agreement compliance) as a way of increasing motivation to the level where the customer implements the release.

  Choosing the right set of carrots and sticks is one of the most complex tasks faced by a marketect. For example, a new release is often made backwards compatible with a previous release. But should it be made backwards compatible with software that was released three years ago? Probably not, as the aggregate costs of maintaining such backwards compatibility, including regression testing, verification of operations, and associated support, can be enormous. Of course, the risks associated with losing important, major customers who may not be interested in upgrading their software to a new release can be just as important, which makes keeping the installed base as current as possible a major job of the marketect. No matter how you keep track of these things, you're going to have "version skippers" and you're going to have to define the upgrade path from whatever version they've got to your current version, no matter how painful the individual steps associated with the upgrade.

## 15.4 RELEASE IDENTIFICATION

Unlike component identification, which may or may not be exposed to the customer, release identification concerns the manner in which a release is identified to a customer. The full identification consists of the product name (see chapter nine) and appropriate versioning information that captures both the appropriate revisions and variations of the product. The goal is to capture all of the necessary information in as few names and identifiers as possible, which helps improve overall efficiency.

Over the years, I've found that there is no single, universal algorithm for creating release identifiers. Moreover, you need slightly different algorithms based on the first two dimensions of release management: what you're releasing and who you're targeting. With these caveats in place, here are algorithms that have worked well for me, and have proven to be considerably more useful than the seemingly arbitrary identification schemes used by many vendors.

### 15.4.1 Full or Complete Releases

Regardless of who you're targeting, I've found that full releases are best identified using:

- the name of the product;
- the four-digit tuple of *x.y.z.build* to capture revision information as defined below; and,
- an arbitrary number of variation identifiers pursuant to the needs of the product.

The meaning of the four-digit tuple *x.y.z.build*, where *x*, *y*, z, and *build* are defined below in Table 15-2. Note that this scheme is designed to take advantage of the natural linear ordering associated with revisions.

| **Tuple** | **Definition** |
|---|---|

| Tuple | Definition |
|---|---|
| x | A major release.<br>One motivation to increment the major release number is when there is some extensive, customer-visible architectural or feature change. These changes, in turn, must be defined and agreed upon by the marketect. Consider a system that manages very large databases. In such a system you might define a major release as any release that:<br>• changed the structure of these databases because of the rather severe impact upgrading the system had on your customers;<br>• modified the published API in a way that makes it incompatible with previous versions;<br>• removed functionality (yes, a good marketect will remove unwanted functionality); or,<br>• added substantial new functionality, such as support for a new operating system.<br><br>In systems that rely on multiple components, incrementing 'x' on one might mean incrementing 'x' on the other. An example is a client server system, in which clients at release x.*.* are guaranteed to work with servers x.*.* and $x_{-1}$.*.*, but not servers $x_{+1}$.*.*.<br>Another important reason whereby 'x' is incremented is for purely business reasons. For example, a customer's support contract might state that that their software will be supported for 18 months after the next major release. By incrementing 'x', you put the customer on a forced path to upgrade (one of the "sticks" I mentioned earlier). In one company I worked at, we designated our first release of a major enterprise class system as "5.0", to both build on a legacy of previous releases of a related product and to help us avoid the concerns that many IT administrators have regarding a "1.0" release of the software.<br>Most marketects should establish strong goals to distribute major releases to all customers as quickly as possible. If it is a "major" release, treat it as such! |

| **Tuple** | **Definition** |
|---|---|
| y | A minor release, usually associated with desirable features or other improvements.<br>The minor release number is incremented when marketing deems that the set of features in the release justifies incrementing this number. It can appear at times that the decision to increment ‘x’ or ‘y’ is quite arbitrary. While this can be OK, I prefer that the marketect define the trigger events that force ‘x’ or ‘y’ to change. In practice, I’ve found that it is easier to define the trigger events associated with ‘x’ than ‘y’. Defining these trigger events helps your customers understand your behavior as you release new products. |
| z | A maintenance or “dot” release.<br>Maintenance releases are made available to all customers who are affected by the contents of the release. Any given dot release should be compatible with other dot releases that share the same major and minor release numbers. |
| build | The specific build number associated with the product. For languages that are compiled, it is trivially easy to compute the build number. For languages that are interpreted, the build number can be created by a simple program that labels a fully checked in code base.<br>The build number is rarely presented to the customer except when the customer is accessing some part of the application that includes the build number for precise identification purposes, usually in relation to technical support. The build number may be optional in situations in which the main component is comprised as an aggregate of subcomponents. To illustrate, suppose you have a product that is comprised of two sub-products, one with release identifier 1.3.2.29 and the other with release identifier 3.6.2.19. It might be acceptable to simply identify your release as 3.4.0.0, or some other special label that indicates the composition. |

**Table 15-2**

It usually works best for marketing to promote only the major and minor identifiers to customers. Thus, when a customer is told that they are receiving version 3.4 of the product, they might really be receiving version 3.4.2.129. The primary motivation for this is that it is simply too costly to try and manage the full tuple in promotional materials, license agreements, sales collateral, and so forth.

By definition, a full release being distributed to any given existing customer is a complete upgrade. In practice, you will find that customer satisfaction is likely improved when you only modify those components that must absolutely be modified, especially in the case of a dot release.

Some people recommend augmenting this scheme to include the target of the distribution. This can be accomplished by inserting an appropriate identifier into the *x.y.z* naming scheme. For example, you might have "A" for an alpha/internal release, "RC" for a release candidate sent to QA, "MR" for a managed release, and "GA" for a general release. These identifiers can be inserted to the right of the "y" or "z" designator ("SuperDraw 4.5A"). I prefer not to do this, as it makes the overall naming convention unnecessarily complex, and because it mixes what is being released with whom the release is targeted. I've also had situations in which a release that was originally intended only as an alpha release was later distributed to a trusted external customer. This change in scope would invalidate the release identifier, and thereby limit its usefulness.

### 15.4.2 Partial Releases

Determining the identification scheme for a partial release is mostly dependent on marketing factors. If the component or artifact can be purchased separately, or as an optional part of the main distribution, it is usually best to allow this component to evolve under its own *x.y.z.build* identification scheme, following the guidelines established in the previous section. The consistency in naming makes it easy for customers to build a mental model of the various optional components. The consistency in naming also makes it easier to construct an overall list of available products.

Partial releases that are not sold separately, as when a virus vendor updates their anti-virus files, and are not expected to be revised in the future do not have the need for the complexities associated with a naming convention that is designed around revisions. In this case, partial releases simply need a unique identifier. For most products a specially defined name and a date is usually sufficient to identify the release.

A key issue in creating partial releases is managing the dependencies that exist between the components or functionality of the partial release and the main product. These dependencies may be captured through rules that govern release identifiers or through the design of the architecture, as described later in this chapter. To illustrate the rules approach, you might require that every release of a component at version *x.y* is compatible with every version of the main system designed $x.y_n$, where

$y_n$ is greater than or equal to $y$. Thus, "SuperDraw Enhanced Rendering Tool 4.5" would be compatible with "SuperDraw 4.5", "SuperDraw 4.6", and so forth. Rules won't do you or your customers any good if you fail to follow them: if SuperDraw were to go through a major upgrade and be released as "SuperDraw 5.0", you would have to modify the release identifier of the enhanced rendering tool to match, even if the code didn't change. While this may seem like "busy work", it will save you and your customers a lot of pain.

### 15.4.3 Patch Releases

Recall that a patch release is a release of some subset of the product, usually designed to precisely replace one or more existing components in a working installation that have known errors. Identifying patch releases represent special challenges. Everyone involved usually has a strong opinion on how to identify a patch release, everyone thinks that their way is the best way, and everyone feels like arguing over each point for an endless amount of time! There are a *lot* of poor choices that can be made when identifying patch releases. This section provides guidance to easily create a sensible patch release identification scheme. Patch release are always associated with something in use, which means that they are created to deal with a customer in a potentially stressful situation. Moreover, the team that creates the patch may not be the team that created the system, and may not be familiar with previous release identification schemes.

Because patches are highly dependent on an existing release, it is usually convenient to refer to the product in the patch identifier. At the same time, you don't want to adopt the *x.y.z.build* numbering convention, because patches are rarely revised unless there was a serious mistake in the QA or release process. More importantly, the linear ordering associated with revisions implies that everything that was included in the previous release is included in the next highest release unless specifically stated otherwise. Thus, we expect that version of 4.5 of our favorite compiler includes and extends upon the functionality included in version 4.2, 4.3, and 4.4. This is *not* necessarily true with patch releases. A given patch may or may not include the modifications of a previous patch. Patches are not cumulative unless specifically designed as such.

Patches are often associated with emotionally charged events or bugs that often take on a life of their own. Since some aspect of these events usually became associated with the patch, I recommend trying to leverage this to your advantage by referring to patches via a name and possibly a date. The net result is that patches are named as *product – x.y{.z{.build}} – patch name*. Note that the maintenance release and build number are optional in this naming convention. In practice, this allows customers to easily identify the patch that they need. The external, customer facing name might be something like “SuperDraw 4.5 Repaginate Long Documents patch”, which means that this patch can be applied to any SuperDraw 4.5.* system. If the patch is focused on a specific dot release, you would refer to it in this scheme as the “SuperDraw 4.5.2 Repaginate Long Documents patch”

Especially complex products may choose to call out those areas affected by the patch, primarily because it makes it easier for the customer to identify on their own which patches they may wish to download from a self-service technical support web site. For example, let’s say that your system is a client/server system, with an optional workflow module. You might then augment the naming convention to be *product – x.y{.z{.build}} – product area – patch name*, as in “SuperDraw 4.5 Repaginate Long Documents Server patch” or “SuperDraw 4.5 email Notification Workflow patch”.

Patches that are dependent upon other patches can call out those dependencies via documentation. If there are a large number of patches associated with a product I recommend collecting all of them into a maintenance release. If this is not possible, then another approach is to adopt a service pack that serves a similar purpose. Make certain your documentation is clear on whether or not service packs are cumulative.

It is important to note that not everyone agrees with naming patches. Consider, for example, what should happen if your patch has an error (yes, it happens). This means that your patch needs to be versioned! Versions as best handled through numbering. One way to resolve this is to version the patch but not include the version identifier unless absolutely needed. So, you’d create a patch in the form *product – x.y{.z{.build}} – patch name.patch version* but only include the version identifier when needed. So, if you have a problem with the first version of the “SuperDraw 4.5 Repaginate Long Documents Server patch” you can release a second version called “SuperDraw 4.5 Repaginate Long Documents Server patch, version 2.” However you choose to resolve this, do not impose an arbitrary limit on the naming. You’ll eventually run into a situation where the limit is exceeded.

Very sophisticated architectures are smart enough to package patches together, tracking what is installed or not installed. Some companies do this as part of their software, and it allows automatic updates (think anti-virus software as a simple example). Other companies, such as ManageSoft, do this on behalf of corporate administrators, taking snapshots of the software on various desktops.

Let’s assume that you’d like to extend your architecture to include patch management. It will need to be smart enough to understand what is, and is not, installed. It will need some mechanism for communicating with a remote server, preferably over the internet, to obtain updates. It should be able to detect if pre-requisites are available, and if not, install them. It needs to be able to determine that an automatic update was installed correctly, didn’t break the system or any settings, and roll back the change if something is wrong. These are very complex requirements, and in general I don’t recommend this approach.

**Bug Fixes Don’t Have to Be Free**

When bug fixes are not included as part of the license, the marketect must make decisions about when to fix a bug. Sometimes the right choice is to fix a bug, building good will with a customer. Sometimes the right choice is to charge for the bug, which can also build good will with a customer.

In one company we had a customer with an extremely urgent request to fix a bug on an unsupported product. Specifically, they had a perpetual license to use the product, but the particular version of the product they had installed was no longer supported. In a very real sense, the customer brought this problem on themselves because they had failed to upgrade their system over the course of several releases. When they contacted us to fix the bug, I originally said “No, if they want the bug fix they can upgrade”.

As the saying goes, “money talks…”, and my original “No” turned to “Yes” once I was able to negotiate a substantial fee for fixing the bug. My team hustled and fixed the bug in record time (even I was a bit surprised at how quickly they fixed the problem!). The customer was so impressed with this service that they subsequently executed the major upgrades that they had delayed far too long.

### 15.4.4 Variations

Variations, like patches, don’t have a monotonically increasing revision number. Naming them and inserting or appending this name into the overall identification string in a way that makes sense is the best way to handle them. For example, suppose that our SuperDraw client/server system supports Linux and Solaris. The binaries for these two operating systems are functionally equivalent but different. Thus, you might call a full release of version 4.5 "SuperDraw 4.5 for Linux" and "SuperDraw 4.5 for Solaris". If you required a patch to this release for Linux you would call it the "SuperDraw 4.5 email Notification Workflow patch for Linux".

Things become more complex when the system or the component supports multiple variations, usually associated with portability, internationalization, or performance characteristics. Suppose that SuperDraw supports six languages and has two performance options: single (default) and multiple-CPU. Some of the ways that this might be named include:

- "SuperDraw 4.5, German language for Linux", for a full release of the single CPU version;
- "SuperDraw 4.5, German language for Linux, multi-CPU", for a full release of the multi-CPU option;
- "SuperDraw 4.5 email Notification Workflow patch, German language, for Linux"

As a general rule, the more options, the more complex the name. I consider this a good thing, because customers don't deal with these names every day, and they often have trouble clearly remembering what they want and/or need. The verbosity of the names helps ensure they are getting the right artifact.

## 15.5 SKUs AND SERIAL NUMBERS

Except in the very smallest of companies, products that are shipped to customers must be identified for a variety of purposes. Most companies rely on SKUs to manage their releases within these systems. Software that is sold in high volumes may also be serialized for unique identification, tracking, and some limited copy protection.

### 15.5.1 SKU MANAGEMENT

The changing nature of releases and variations, and their potentially long and descriptive names, puts a great deal of pressure on other corporate systems that must account for sales, shipments, upgrades, or other customer activities. The easiest way to manage this is to establish SKUs (Stock Keeping Units) to manage releases. A *SKU* is a unique release identifier that enables the release to be distinguished from all other releases within the various corporate systems that must keep track of these things. As an identifier, a SKU is not dependent on the various textual descriptions of releases that I've described in previous sections.

SKUs are used in a variety of circumstances. Orders for specific releases are managed through SKUs. Prices, which are not part of the release identifier or product name, must be obtained from a pricing database or pricing system. The primary key used to find the price is the SKU. Inventory levels for physical goods are usually managed through SKUs. "Inventory" for electronic software distribution doesn't make sense, but since chances are good that you'll do some physical and some electronic distribution, SKUs still have a place in your inventory management system.

Teams often struggle with knowing when to assign a SKU to a release. Creating and managing SKUs introduces a variety of additional tasks into the overall release process, and it is understandable that marketects avoid creating SKUs unless absolutely necessary. The following guidelines have worked well for me.

- Always assign a SKU to any release that can be sold. This is regardless of the scope of the release (patch, partial, or full) and target (controlled or general).
- Try to assign a SKU to any general release (a release that is targeted to all customers), regardless of scope (patch, partial, or full). This makes tracking what is "globally" available convenient.
- Always assign a SKU to any release if your primary customer tracking and distribution systems are keyed to SKUs *and* you can track who obtained the release. This provides you with great knowledge in who has obtained what release.
- Avoid creating SKUs for releases that are simply posted on self-service web sites, such as technical support or free download web sites. You're not selling these, so there is no need to create SKUs in your financial systems, and you're not keeping track of which customer has obtained these releases.

Of course, these recommendations should be followed only as they relate to your current corporate policies. If your company mandates that every release, no matter what, has a SKU, then by all means, create SKUs.

You can't mandate the format of a SKU, because it is usually under the control of other people other people (such as corporate IT or fulfillment) and systems (such as your order fulfillment and accounting system). You can, however, provide these organizations with estimates of how many SKUs you will need, and work with them in creating a format that will enable you to accomplish your objectives.

Let’s return once again to our SuperDraw example. SuperDraw is just one of four product lines at SuperSoft. Other parts of the company, such as accounting and order fulfillment, don’t care about the specific product line names or version identifiers. Accounting just wants to track sales results by product line and division, while order fulfillment just wants to make certain that the right deliverable is shipped to the customer. Instead, they care about the SKU, which is their way of keeping track of these products. For various reporting reasons, accounting has defined a SKU format as follows: NNNN-MMMM-#, where NNNN is a four character division identifier, MMMM is a four character product identifier, and # is a unique product number of arbitrary length. Accounting has defined the division identifiers; product management and accounting together have defined the product identifiers; and product management is responsible for assigning unique product numbers. Product management might define the following SKUs, as shown in Table 15-3

| **SKU** | **Release** |
|---|---|
| DRAW-SERV-0001 | SuperDraw 4.5, German language for Linux |
| DRAW-SERV-0002 | SuperDraw 4.5, Russian language for Linux |
| DRAW-SERV-0003 | SuperDraw 4.5, English language for Linux |

**Table 15-3**

As you can see, the number of SKUs associated with a released product can be estimated by adding the number of full releases multiplied by the number of full release variations, added to the number of partial releases of optional components multiplied by the number of optional component variations, added to the number of any other item that is sold not already accounted for by these estimates. In practice, this means that SKUs can quickly grow to hundreds of thousands of identifiers – make certain you have enough available.

### 15.5.2 Serial Numbers, Registration, and Activation

A SKU represents a class of products for tracking purposes. It can’t identify an individual product sold to an individual customer. A *serial number* is a unique identifier that distinguishes individual products. *Registration* is the process by which the entity that purchases a product formally associates themselves with that entity that sold it, usually with the serial number acting as a key. *Activation* is forced registration process in which the product doesn't work, or may not work completely, or will only work under certain circumstances, unless the customer registers themselves to the company. Note that activation and registration are related to license enforcement (see chapter four), as one of the goals of activation is to ensure that only properly licensed software is allowed to work.

In the physical world, serial numbers range from the lot numbers and associated codes printed on vitamin bottles to the identification tag affixed to my PDA. As a digital good, it is difficult to uniquely identify software with a serial number. Unlike physical goods, digital goods are often trivially copied, and embedding a serial number within the object code at production is an often expensive change to internal processes. Moreover, unless you've adopted one of the license enforcement schemes described in chapter four to prevent copying or modification of your software, serial numbers can be easily changed.

Although serial numbers can be a bother, there are real benefits to their usage. By associating a serial number with a product and asking the user to register that serial number with your company, you can collect vital demographic statistics as well as tailor marketing campaigns to meet user needs. Consider that once your customers have registered their serial number, you can collect data that lets you notify them of product upgrades, bug fixes, and other offers of additional products and services that may be of interest. Registered customers may also be willing to provide you with additional valuable information, including their preferences for new features or their willingness to participate in beta programs.

Properly registered serial numbers can help reduce piracy. In the past, when serial numbers were printed on CD sleeves, the sheer size of the program and the difficulty in duplicating CDs were deterrents to copying the software. Technological advancements have made such deterrents ineffective, and software developers are continually looking for ways to reduce piracy.

Software activation is one technique that is effective at deterring piracy while also increasing the number of users that register their software. The specific processes employed by various companies vary according to need or by the software activation vendor chosen to provide for the activation. The general process works something like this:

1. The software publisher prepares their software for distribution. A serial number may be associated with the software at this time, although this is not required, as serial numbers

can be generated dynamically. The software is protected in some way to prevent execution until it has been given the proper activation code.

2. The entity purchasing the software, the consumer or an enterprise, installs the software. The installation process binds the software to the machine. The binding process is accomplished by taking a unique "fingerprint" of the machine on which the software has been installed, such as the processor id, motherboard serial number, or MAC address of the primary Ethernet card. This information is usually stored in a secure location to help prevent illegal copying of the software.
3. To use the software, the purchasing entity contacts the publisher with either the serial number, the machine fingerprint, or some combination thereof and requests an activation code. This process may also force the entity to register themselves with the publisher. It is convenient if the publisher provides several channels for acquiring the activation code, such as the internet, email, phone, and fax. During this process, a serial number is created or stored within various corporate databases and marked as "activated". Registering the serial number helps ensure it is uniquely identifying a product, while storing the machine fingerprint and binding the software to the machine helps prevent piracy.
4. The activation code is given to the software and stored in a secured location. The activation code, or license, depending on the technology that was chosen, allows the software to be used – but only on the designated machine.

Choosing to adopt a software activation process is a strategic decision. Most companies don't have the resources to create an effective activation system. Several vendors provide software activation solutions. Each proposed offering must be evaluated relative to existing and planned backend systems and workflows to make certain needs are met. Managing the backend systems is likely to be a much larger task than choosing a vendor to provide software activation services.

## 15.6 Release Management Influences On Tarchitecture

Prior knowledge of release management requirements can improve your tarchitecture by encouraging you to make tarchitectural choices that make release management easier. Consider the following.

- *Use existing solution and infrastructure wherever possible.* Chances are good that one or more aspects of your deployment architecture has been designed to handle any number of a variety of issues associated with technical configuration management. Microsoft, for example, has an extensive (some would say “nightmarish”) infrastructure for managing components packed as DLLs and their associated dependencies. Learn these infrastructures and leverage them when you can.

- *Put version information into your tarchitecture as early as possible.* Once you’ve identified the need for version information in your tarchitecture, put it in. Don’t delay. Retrofitting version information is expensive and painful.

- *Components should know their dependencies.* In component-based systems, it is helpful when components know which version of the other components they can work well with. It is easiest if this is done in a data-driven approach, perhaps through a special dependency checking component that processes dependency information stored in a configuration file. A practical application of this is when a client is directed at a server that can’t support it. Instead of failing outright, the client could inform the user that the server does not support this client and ideally provide the user with some information on how to rectify the situation.

- *Messages and protocols should be versioned.* Messages sent between components should be versioned so that changes in content or processing rules can be managed.

- *Databases and tables within databases should have versions.* One way to do this is to create a system table that simply contains the a version identifier of each table in the schema. This system table can be extended to provide release information for specific columns within tables, as needed by your application. This makes upgrading schemas in the field, detecting changes made to released schemas, and making certain components that read from and/or write to the database do this properly much easier.

- *Any component that can be updated should be versioned.* This makes certain that your technical support organization can quickly assess whether a component should be updated in response to a customer problem. It also simplifies the installation of partial and patch releases.

- *Internal components should understand the versions of external components they require.* If you're system *requires* Solaris 2.8, check for it. If you *know* your system has problems running on Windows XP, check for it.

- *There must be a way of obtaining the version of all versioned artifacts.* This enables technical support to help the customer, and is the foundation for customer self-service or automatically updating software.

- *Beware the testing and support implications of patches.* Creating more components, and providing support and/or backwards compatibility for previous releases, can increase testing and verification complexity at an exponential rate. Just because you can provide backwards compatibility support doesn't mean that you must.

## CHAPTER SUMMARY

- *Release management* ensures that the correct artifacts are shipped to the customers that want or need them. It is based on the following concepts:
    - program families;
    - components and artifacts;
    - versions – a fixed or frozen component or other artifact;
    - revision – a new version intended to supercede the old;
    - variation – an alternative implementation;
    - distribution – a set version created for distribution to a set of customers
    - release – a named distribution.
- Managing releases involves three factors: what you're releasing, who you're targeting for this release, and their motivation for obtaining and using the release.

- Releases must be identified. The four-digit tuple of *x.y.z.build* provides a proven foundation for creating release identifiers.
- SKUs are used to manage releases within back office systems such as accounting and order fulfillment.

## CHECK THIS

- ❑ For each release, we have defined what we're releasing and who we're targeting. We have also estimated the customers motivation for obtaining the release.
- ❑ We have a defined system for identifying releases.
- ❑ Each release that needs a SKU has one.
- ❑ Each release that needs a serial number has one.

## TRY THIS

1. What are the corporate systems that track releases and SKUs? How do these systems inter-operate?

2. Does your system require serial numbers? Should it?

3. Should you require your users to register their software? Why or why not? Should you require your customers to activate their software? Why or why not?

# 16. SECURITY

*— with Ron Lunde*
*System Architect*
*Aladdin Knowledge Systems, Inc.*

Most of what you're trying to accomplish with your tarchitecture is to make things easy. You want your products and systems to be easy for your users – easy to install, easy to configure, easy to use, and easy to learn. You want them to be easy for your developers – easy to understand, easy to change, easy to extend, and easy to re-purpose. If problems occur, you want the problems to be easy to detect, easy to diagnose, and easy to fix. And, you want them to be easy for the ecosystem that inevitably develops around a winning solution – easy for solution providers to extend, easy for system integrators to integrate, and easy for operations to install, maintain, and extend as necessary.

The main thing that is different about software security is that it's not about making things easy. It's about making things hard. You want your software to be hard to steal, hard to misuse, and hard to fool. You want to make certain that no one is cheating your business model or using your software against the terms of the license agreement.

Security is an essential part of a winning solution, yet it is often overlooked until the system is nearing completion. Just like an effective error or exception handling scheme, security must be taken into consideration during the design of tarchitecture. Security isn't icing on a cake. Security is "eggs in the batter", and if it isn't in there from the start you can't go back and add it when you take it out of the oven and serve it to your customers.

In this chapter we'll explore the ways in which software and the data that it manages can be misused, and some of the techniques and technologies used to prevent this potential misuse.

Keep in mind that security is a huge topic. Many excellent books have been written on security, and on specific aspects of security, such as cryptography. You should find this chapter useful even if you've read those books, however, since our focus is on how to create a winning solution using the technologies available to you, rather than the details of the technologies themselves.

## 16.1 VIRUSES, HACKERS, AND PIRATES – OH MY!

There are four main types of security that you need to consider.

*Digital Identity Management.* Most enterprise systems provide services to either humans or other systems in fulfillment of a larger transaction. If you're building an enterprise application, some of the things you're going to do include defining different capabilities for different users and the roles that they assume, formally tracking their actions, and verifying that a given user is who they claim to be.

*Transaction Security*. The communication between the various parts of your system must be secure, so that portions of your system cannot be replaced by unauthorized components, and so that messages cannot be intercepted, altered or hijacked. Hackers can exploit holes in transaction security, or simply prevent your transactions from occurring, using denial-of-service attacks, and you'll also need to protect against these.

*Software Security*. This involves protecting your software from viruses or hackers. Nobody should be able to alter your software except those you specifically authorize, and nobody should be able to exploit holes in your security to gain unauthorized access to one of your customers' systems. Software security protects your work from viruses, software pirates, and some types of hacking. One aspect of software security, software piracy (the illegal copying of software) was covered in chapter four.

*Information Security*. The databases and information repositories used by your system need to be secured against unauthorized access or use. In many circumstances, the real target isn't the software that manages the web site, but the detailed transaction history, including credit card account numbers, that the software has stored in a database. Unless you take explicit steps to prevent unauthorized access, these data may be at risk.

Each type of security requires its own tools and techniques, which we'll cover in greater detail later in this chapter. Some of these techniques, such as maintaining confidentiality through encryption, can be used to achieve higher security in more than one area. For example, to maintain confidentiality of messages you can encrypt the message; to maintain confidentiality of database records you can encrypt the data.

Sometimes you may need to pay very little attention to one of the types of security and focus all of your attention elsewhere. At other times, you're going to have to explicitly account for every kind of security. While information security may not be a strong requirement for a family financial management application running on a personal computer, it is likely to be of paramount importance to an enterprise application that maintains payroll records. While digital identity management isn't needed for a game that you play by yourself, it is of critical importance when playing a multi-player game for a monthly fee over the Internet! Getting clear on those elements of security that are important for your application is critical for creating a winning solution.

### 16.1.1 MANAGING RISK

The first thing most security experts will tell you is that there is no such thing as a "secure" system. You can increase the level of security, so that you can talk about having a "very secure" system, but you can never be 100 percent sure that your system is safe from all types of attack. To illustrate, firewalls and intrusion detection tools are commonly used to prevent unauthorized access, but hackers are not the only concern – disgruntled employees and others with physical access to your computers can create serious problems, while completely bypassing the firewalls that are meant to keep hackers out! In fact, risk management consulting firm Kroll estimates that 80% of all attacks happen from inside the firewall (you did perform a full background check of the temporary secretary before you let him or her borrow your corporate id, didn't you?). Just like it is physically impossible to cool matter to absolute zero degrees, it also becomes exponentially harder to make your system increasingly secure, and you'll never be able to say that you are absolutely secure: You can't prevent every element of "crazy" human behavior.

You can't "eliminate" risk. You can manage it. Usually, it isn't difficult to make the cost of hacking your system prohibitively expensive, so that no real hacker will have the time or money to successfully do so. Consider software piracy. When large programs were first distributed on CD-ROM, piracy wasn't that bad. You couldn't copy a CD-ROM, and no one wanted to download 100Mb on a 56Kb modem. Those times are long gone, and copying software onto a variety of media or downloading it on high speed Internet connections has lead to an explosion of pirated software.

It's possible to go too far with security, just like anything else. Storing all of your data in an encrypted format might make your system more secure, but it will certainly prevent reasonable integrations with other corporate systems. Security is an area where the marketect and the tarchitect must work together. The market must lead the assessments of risk by determining what harm would befall the customer, the company, or other entities if one or more elements of security are compromised. The tarchitect must inform the marketect of the possible ways to handle these problems, as well as make his or her own independent assessment of risk (not too many marketects are going to worry about transaction security). Once the perceived risks and mechanisms to handle them are known, the marketect and the tarchitect can then make the tough calls on how to deal with the potential problems.

### 16.1.2 SEE NO EVIL, SPEAK NO EVIL

How big of a problem is security, anyway? After all, it's actually fairly rare to hear of a major corporation having a large security problem. If that's true, then investing in security is not a wise investment, and everyone would be better off putting their time and efforts towards improving other aspects of the system.

According to the seventh annual joint FBI/Computer Security Institute (CSI) Computer Crime and Security Survey[1], 75 percent of the companies surveyed do not report security problems to law enforcement agencies, because of negative publicity or fear of giving their competitors an advantage as a result.

The same report showed that of the 500 corporations surveyed, 40 percent reported denial of service attacks, 20 percent reported theft of proprietary information, 12 percent reported financial fraud, and 8 percent reported sabotage. In fact, lapses of security result in hundreds of millions of dollars lost annually. The time to start considering security is right at the start, when you're designing the tarchitecture.

If you're working in an environment that doesn't worry about security, start shouting.

[1] http://zdnet.com.com/2100-1105-877606.html

## 16.2 DIGITAL IDENTITY MANAGEMENT

This section explores the most important building blocks of digital identity management, including authorization and authentication.

### 16.2.1 AUTHORIZATION – DEFINING WHO CAN DO WHAT

Like many elements of security, there is a static and dynamic aspect to authorization. The static portion deals with defining which entities get rights to perform an operation, access part of the system, or access part of the database. The most common approach is to have a trusted user, such as the system administrator, define the rights of other users, by individual or by class. For example, if you were designing a medical records system, you might want to permit a patient to get at all of his own patient history, but not allow him to get at any data about anyone else.

The dynamic aspect of authorization concerns the check that is made to ensure that a given user or entity has the necessary rights to perform the operation or access a certain part of the system or database. In sophisticated systems these checks need to happen at run-time because the rules that govern authorization may be based on a variety of run-time parameters, including the role the user has assumed while accessing the system, their prior behavior, the state of the system, or even the behavior of others using the system at the same time.

There are a number of technologies and rights management systems that you can use, or tie in to, to provide authorization and access control. Lightweight Directory Access Protocol (LDAP) is one commonly used system. Role Based Access Control (RBAC)[2] is a generic name for technologies used to provide authorization based upon the “role” of a user. If you need to provide file system authorization, ACLs (Access Control Lists) may be all that you need. It’s a good idea to isolate the systems you use for authorization by wrapping them with your own authorization layer, so that you can swap out one type for another as the requirements of the system change over time.

### 16.2.2 AUTHENTICATION – PROOF OF IDENTIFY

Authentication is the process a system uses to ensure that an entity is who they claim. This can be important as the precursor to authorization, or it may be required to simply engage in a trusted

[2] http://csrc.nist.gov/rbac/

transaction. The authentication technology that is most appropriate is based on whether you have a closed or an open system and whether or not you require third party certification of identity.

## CLOSED SYSTEMS

In closed systems, there is little or no need for an independent third party to certify an identity. The system itself, by its structure and/or operation, provides for acceptable levels of authentication. Many enterprise applications that are based on granting specific rights to authenticated users are closed systems. Once the system administrator has registered a user id with the system (either directly or through a directory) and provided this user with the necessary means to access the system (e.g., a password and a token), whenever they successfully log into the system they are now recognized as an *authenticated*, certified user.

Authentication in closed systems is usually based on one or more of the following three types of things:

1. Something you know – e.g. a password;
2. Something you have – e.g. a smart card or a computer that has been uniquely identified, either through a machine fingerprint or an actual unique identifier stored in the processor or the motherboard;
3. Something you are (biometric) – e.g. a thumb print or a voice print.

A combination of any two of these is normally considered strong authentication. Extremely secure environments may require all three, or multiple applications of each kind of authentication. Of course, by far the most common, and certainly most insecure, of these items is a simple password. Users often choose passwords that are easily guessed. When system administrators force them to choose more secure passwords, users often make the system even less secure by writing them down on a scrap of paper or allowing their favorite web browser to "save" the password for easy reference. Be careful of being lulled into a false sense of security because your system requires a password!

From a tarchitecture perspective, we recommend abstracting the approach that you use for authentication. Many systems start with a simply password-level authorization and evolve through use or customer demand to require stronger forms of authentication.

## OPEN SYSTEMS

Unlike closed systems, open systems require an independent third party to authenticate a user. Examples of open systems include sending a secure email between two parties, or trying to establish a secure communication link between two applications on the web. In this case we need some kind of certification by a third party that the communicating entities are whom they claim.

We've faced the problems of open systems before, and we've solved them in similar ways. In the 18th century, if you were a gentleman scientist and you wanted to visit a member of the Royal Society whom you had not met, to study in his library for example, you would bring a "letter of introduction" from a colleague known to both of you, to show that you could be trusted. If the fellow you were visiting wanted to be extra sure that you were who you claimed to be, he would compare the signature of the letter with the signature on correspondence from your mutual colleague, in order to make sure that you hadn't simply forged the letter. We do the same thing today, except that we now use certificate authorities and digital certificates instead of venerable gentlemen and parchment and ink. It is these certificates and the trusted third party that provides for the necessary levels of authentication for open transactions.

The idea is that you prove you are who you claim to be to a trusted third party, and they give you a digital certificate signed with their private key. Anyone can use the trusted third party's public key to verify the certificate, and since they trust the signer, and the signer is saying that you are who you claim to be, they can now trust you. This hierarchy of trust can extend in a certificate chain – each certificate signed by an agency that vouches for them, until it reaches the top – the "certificate authority".

Certificate authorities support "certificate revocation" – which means that if you lose your private key, or someone up the certificate chain has their security compromised, the certificate corresponding to the lost key can be revoked. Then you get another certificate, and everything continues as usual – except that anyone presenting the revoked certificate will not be authenticated.

The problem with certificate revocation is that it means you sometimes have to be able to talk to the certificate authority in real time, to make sure a certificate you want to authenticate has not been revoked. For isolated subnets or disconnected applications, this isn't possible. It also means that if you choose to run your own certificate authority, you must ensure that your system is reliable – 24x7 operation.

For this reason, you should carefully consider whether your system needs revocable authentication – sometimes, if you combine certificate-based authentication with other information, you may not need it. For example, in one system Ron worked on, the server checked the IP address of the caller, as well as the caller's certificate. Even if the client's certificate and private key were stolen, if the originating IP address was incorrect, the server would log and disallow the transaction.

There are several challenges associated with certificate-based authentication. Let's start with the first step, proving you are who you claim to be to the trusted third party. Some certificate authorities have a very poor record of verifying these claims, and there have been several multi-million dollar lawsuits filed over inappropriate verification procedures. In addition, some certificate authorities have hurt themselves through deceptive marketing practices. Simply put, how can you "trust" a certificate authority who deceives customers? There is no undisputed world-wide leader in certificate management. There are several for-profit, not-for-profit, quasi-governmental, and governmental certificate authorities, all of which hinders the creation of global solutions.

In addition to some of these technical and operational challenges, we are also deeply dismayed at the level of complexity exposed to average users who may wish to use, or who are forced to use, digital certificates. The level of usability associated with acquiring, managing, and using certificates is abysmal. Until usability improves, the use of certificates will continue to remain relatively low.

### HYBRID SYSTEMS

It is entirely possible that you application may have involve aspects of an open system with aspects of a closed system. Consider a corporate email system. It is closed, in that a user can probably be acceptably authenticated with the same corporate user id and password that gave them access to the corporate network when they first logged in. When receiving an email from another internal user, the user can be fairly confident that the person who sent the mail did, in fact, send it.

The system is open, in that this user may have the need to send and receive secure emails with entities that are external. Suppose this same user receives a distressing email from their outside attorney. It would be good if there was a way to certify the identity of the sender.

## 16.3 TRANSACTION SECURITY

Transaction security is of most importance if you're making a client/server application or a web service. The objectives of transaction security are to provide support for Auditability, Integrity, Confidentiality, and Accountability. It is also likely that you're going to rely on the authentication and authorization approaches discussed in the previous section to work in concert with these additional security techniques.

### 16.3.1 AUDITABILITY – PROOF OF ACTIVITY

An auditable system requires authentication and a system designed to track behavior. Together, these enable the system to generate credible reports of activity. These reports can be used for a variety of reasons. Consider financial applications that use proof of activity to spot suspicious behavior while it is occurring. For example, a "velocity check" will spot a sudden higher-than-expected transaction volume coming from the same source, which may be an indication of fraud. Often, audit data is copied to an off-site location that is managed by a trusted third party, so that the audit data can't be tampered with in order to remove evidence of transactions.

The primary motivation for creating an auditable systems is that governmental or industry group has created guidelines, rules, or other regulations that require some kind of auditability in the operation of the system. Secondary motivations include the desire to provide additional services over historical data or analyze large bodies of data to spot hidden trends, such as in the data warehousing market.

### 16.3.2 INTEGRITY – PREVENTING TAMPERING AND ALTERATION OF DATA

A "digest" function (also known as a one-way hash) takes a block of data and returns a small byte sequence that represents it. Good digest functions are very random, so that any change to the original data results in a different digest value, and so that it is very difficult to find a different block of data that has the same value. Algorithms such as SHA1 and MD5 are well known digest functions, and they've been exposed to public scrutiny to make sure they meet the above requirements.

Digests are normally encrypted with a private key, to create a digital signature. Anyone with the public key can decrypt the digest. They can then run the digest algorithm on the block of data producing a new digest. If the digests match, the source data has not been altered. The reason we don't simply encrypt the data itself with the private key is that private key encryption is a very expensive operation, and would take far too long to encrypt large blocks of data. The digest algorithm is very fast, and produces a small result, which is easy to encrypt. The signature for a block of data need not be attached to the data, or even stored with it.

Generally speaking, integrity is easy to provide – the code that implements the algorithms used is readily available, and much of it is open source – so there's little excuse not to use it if there's a chance it will be needed. Integrity is often correlated with auditability, as a way of proving that the record of system behavior has not changed.

### 16.3.3 CONFIDENTIALITY – KEEPING DATA AWAY FROM THOSE NOT ENTITLED TO IT

Encryption also comes to the rescue, when what you want to do is make sure that nobody can intercept your messages or the data stored in your system and gain some advantage by learning their contents or reading these data. Consider the difference between a log file that tracks system activity for billing purposes verses a message sent between two systems or a record that contains a credit card. In the case of a log file, you may only need to implement an integrity mechanism to prove that the log file was not altered. In the case of the credit card, you not only need to prevent a devious person from altering a message or record that contains a credit card number – you also need to prevent them from being able to read or access it in the first place!

The most common approach to encrypting data, and one used by systems such as Secure Sockets Layer (SSL) is a two step process. The first step relies on symmetric key encryption, which is very fast. The sender of the message generates a symmetric key and then uses these with an algorithm like RC4 to encrypt the data. In the second step the sender encrypts the symmetric key used in the first step with the public key of the receiver. Public key encryption is much slower than symmetric encryption, but because the data that is being encrypted is so small, it happens very quickly. The sender then sends both the encrypted data and the encrypted key to the receiver. The receiver decrypts the key using their private key and then decrypts the data. The overall process is a good balance of speed and cryptographic strength. Since SSL is very widespread, we recommend that you use it whenever possible. We also expect that implementers of SSL will adopt even stronger cryptographic algorithms, such as Rijndael instead of RC4. Using the standard means that you'll get the benefits of the standard as it improves (as well as the drawbacks if is cracked; although this is possible, we're comfortable in recommending the use of this standard).

### 16.3.4 ACCOUNTABILITY – HOLDING PEOPLE RESPONSIBLE FOR THEIR ACTIONS

One of the objectives of transaction security is to provide a mechanism for "non-repudiation". That means that if you send a message to me, I can later prove that you sent it, and that only you could have sent it. Only operations that involve public/private key operations permit non-repudiation, but any operation that does rely on public/private key operations will supply that capability. Read this sentence again, slowly. Suppose, for example, that a user logs into a multi-user system using a two-factor authorization scheme (such as with a Smart Card and password). Even though you have authorized this user, you may not be able to hold them accountable for their actions—unless you've designed the system to that these actions require strong authentication.

Strong accountability requires strong authentication – if all you need to send a message is a smart card containing a digital certificate, someone else may have stolen your smart card and used it. On the other hand, if you need to enter a password as well, the chance that someone else could have sent the message becomes much lower.

Digital signatures are used along with strong authentication to provide strong accountability. Nobody without your private key can send a message signed by it since all they have is your public key. That means that by saving a digitally signed message, a service can later prove that you sent it.

## 16.4 SOFTWARE SECURITY

Some of the issues associated with software security were raised in chapter four, in the discussion of how license managers can be used to enforce the terms and conditions of business and licensing models. In this section we'll cover the topic more thoroughly.

The first thing most software developers think of when it comes to software security is preventing software piracy. This is a good thing, because no matter which reports you read about software piracy, it a multi-billion dollar problem. Several techniques have emerged to try and make software more secure, each a bit more complicated, and each a bit more effective at deterring thieves.

- *Serial numbers and activations.* Recall from chapter fifteen that a *serial number* is a unique identifier that distinguishes individual products, and that through an activation process you can bind a legal copy of the software to a specific machine. A more advanced technique to accomplish the same thing is to use a digitally signed license with software, so that an application or a service will not run unless a valid license with a valid signature is present.

  Most people aren't hackers, and most people do not frequent hacker pages on the internet. Most of the illegally copied software in the USA is passed on by friends and relatives, who may have few qualms about passing along a serial number or a license as well. Serial numbers, software activations, and digitally signed licenses all cut down on "casual copying" – after all, a serial number can be traced back to its official owner, if it is posted on the internet or otherwise "gets away".

  Hackers attack these schemes in a number of different ways. The most damaging attack is when they reverse engineer the serial number generation algorithm, and write their own serial number generator. That opens the door for actual software piracy – the selling of illegal copies.

  Digitally signed licenses are much better – the private key needed to sign the licenses will not be available, so the hacker must either modify the program to bypass the software security, or else replace the embedded public key with a different key, for which the pirate has a private key. That way, the pirate can generate his own licenses.

- *Protecting the validation code.* All of the schemes we've discussed thus far are based on embedding one or more checks in your code that confirm the presence of a valid license. A typical code fragment that does this might look like the following:

```
if (!LicenseIsValid(licenseFileName))
{
   ComplainAndExit();
}
```

  Surprisingly, it's often trivial for a hacker or a software pirate to bypass this kind of security check, even if you have a highly secure, digitally signed license and even if your license validation routine is difficult to reverse engineer. All the *software* hacker has to do is use a disassembler, find the code corresponding to the "if" statement, and insert a jump around the test or replace the test with no-ops. That means that you can't just use a Boolean return to check your license – you have to be a lot trickier than that, to foil a hacker.

Instead of just asking a simplistic "yes/no" question, more secure approaches are based on actually storing something that your application needs to run as encrypted data within the signed license. This could be a critical function, like an initialization routine, or a function that registers subcomponents within the application. The application then verifies the license and decrypts this data, which would in turn control the behavior of the software. You'll then have to protect the software that performs the license validation and decryption… which is when you'll realize that the professional license managers described in chapter four are actually pretty hard to write!

- *Hardware binding.* Hardware binding, as discussed in chapter 15, is the process of associating or binding information about the user's hardware or other hardware device (often referred to as a "dongle") into the software license. This prevents a license that was generated for one machine from working on another or it prevents the software from operating unless the hardware device is physically attached to the computer (usually via a serial or USB port).

  While that's a good way to cut down on "casual copying", it is not without risk for the publisher. Every time an end user buys a new computer or upgrades his hardware, the software will stop working. This normally creates an unhappy customer, and will result in at least one expensive customer support call. All other things being (relatively) equal, a potential customer is likely to choose a product that is not bound to the computer, especially if he is burned once or more when he tries to run software he paid for, and can't.

An important thing to remember about piracy prevention is this: many, if not most, of the people who run illegal copies of software would not have bought the software if they couldn't get a free copy. You wouldn't want to make life difficult for your legitimate users, possibly driving them away, in a futile attempt to prevent people from using your software who would never actually buy it.

Software security can also add significantly to the cost of developing, maintaining, and supporting your software. "Obfuscation" is often helpful in foiling attempts by hackers, but it can make your programs extremely difficult to debug. Even running a certificate server as a certificate authority can add tremendous cost, given the 24 x 7 operational requirements.

*This does not mean that we endorse software piracy.* The facts are that casually copying an application from your work computer to your home computer, or purchasing one copy of an operating system and installing it on more than one computer with the necessary license rights, or posting an application on a web site or making it available via a P2P network are all **illegal**. The best way to address software piracy is to examine the risks associated with implementing strong anti-piracy tactics with the potential for lost revenue. If you're selling enterprise class software, or providing your software as a service via an xSP, software piracy is not likely to be a problem, partly because of the intense integration and support requirements, and partly because of the ease with which piracy can be determined through non-technical means. If your software requires regular updates of code or data to be useful, software piracy may not be that much of a problem.

But, if you're losing thousands or millions of dollars because of software piracy, then do something about it. Explore a new business model, such as a rental. Implement a lightweight protection mechanism and see who breaks it. If you look at the hacker web sites or Usenet lists; use Google "groups" to view alt.2600.*] and find your software freely available, implement increasing forms of protection. You may even be justified in using a hardware based software protection device.

## 16.5 INFORMATION SECURITY

Many of the same techniques that are used for transaction security can also be adapted for information security, although they are far less effective.

To understand information security, ask yourself the question "what if someone had open access to all my files and databases?" If your data is encrypted, and you've stored the encryption keys in the database or a file in the same system, then the hacker has the keys needed to decrypt it. If your data is signed, the hacker can alter it since he has the keys needed to sign it again after it is altered. You can, and should, store these things outside of the database or file system managed by the application, but this is an operations nightmare. In addition, it is common for other systems to want to manipulate these data, and encrypting these data makes them useless for this purpose.

As a result, the primary approach to information security is not to protect the information from the bad guys once they've gotten access to it, but to instead prevent them from getting access to it in the first place. The main tools in your toolkit for doing this are network tools such as firewalls and intrusion detection software, and user management tools such as password policy checkers.

Password security is an interesting area in itself. Many companies do not ever store passwords in their database – instead, they run a digest algorithm over the password and store only the digest. That means that no hacker can ever steal a whole database full of passwords, since the passwords themselves are never stored. Unfortunately, it also means that if a user loses a password, the best you can do it generate them a new one – you can't derive the password from the result of the digest function any better than a hacker can. You have probably seen many web sites that operate that way, especially those that ask you for a special question that is used to help authenticate you should you lose your password and require a new one.

Another technique is to store passwords on an internal system that is not directly reachable from the Internet, but only from a server on a local file system. That way, if a hacker is going to get passwords, he must first hack into the outer system, and then use that as a way to get at the inner system. Intrusion detection systems can often foil such attempts.

One of the common things that we think of in terms of information security is information theft. In the digital world, theft boils down to illegally copying bits from one computer to another. This may not be the most serious threat to your information. Semantic attacks, in which the bad guys alter data stored in a database to gain an advantage, are on the increase. For some reason, these attacks seem less harmful, perhaps because Hollywood has created too many cute movies in which the nerdy hacker changes his grades. When that nerdy hacker is changing bank accounts, altering credit histories, or changing voting records, things aren't so funny.

Information security should guide your system architecture, but you should not attempt to write your own tools – there are many excellent tools available for that already available.

## 16.6 SECRET ALGORITHMS OR SECRET KEYS?

The techniques associated with achieving security are usually based on a mathematically complex algorithm in the public domain, or one whose underlying logic is in the public domain. This may not matter much to you – unless you're the one who is picking the algorithm! In fact, the task of picking an algorithm can feel so daunting that you might not want to pick one at all. Instead, you might think that a secret algorithm would be the ultimate in security. After all, one of the best kept secrets of World War II was the German Enigma machine, which they used to encrypt and decrypt wartime information. It was only after an enigma machine was captured intact that the Allies were finally able to start decoding messages.

Which is why you shouldn't use a secret algorithm. More generally, "security through obscurity" is one of the weakest forms of security available. Your "secret" algorithm can be found, reverse-engineered, or leaked. Given the tremendous and ongoing improvements in raw computing power, no algorithm is safe from a brute force approach to cracking it.

A better choice than a secret algorithm is any of the excellent public algorithms that are available for whatever implementation you feel is important. Public algorithms are under continuous scrutiny, and many mathematicians are hoping to make a name for themselves by finding a method of cracking the algorithm that doesn't require a brute force (i.e. "try every key") attack.

Smart security managers will avoid using any product or service that uses a "secret" security algorithm, since doing so provides no guarantee of security. On the other hand, secret keys are used all the time, and nobody has any issue with them.

It can be tempting to come up with your own algorithms for encryption, but it is pointless – even if your algorithm is significantly better than others, nobody is going to be willing to spend the time and money it takes to verify that. Use standard algorithms and secret keys – until quantum computers are readily available, you'll be perfectly safe.

## 16.7 BACK DOORS

It can be tempting to put in "back doors" in your code, so that a customer service representative (for example) can get at secure data, even if the customer has lost a password or a certificate. You might think that you can appear as a shining hero, saving the customer from their own foolishness, with just a little extra effort to put in a carefully controlled back door.

That's possibly the case. On the other hand, back doors are often poorly imple mented, and can give way to easy hacks, resulting in a lot of wasted time and effort on security.

To illustrate why, we'll use the "100th Window Problem". Let's say your company has a large building with 100 windows that are all open, and you rush about locking them all before you leave for the day. Unfortunately, it turns out that you actually only lock 99 of the 100 windows – you overlook one. Even if you have excellent locks and burglar alarms on all the locked windows, a thief can slip in the 100th window and make off with all your corporate goods. The more windows you add, the harder it is to make sure that you have left things secure.

Generally speaking, there's no point in securing less than 100 percent of the relevant portion of your system. If you're going to do it at all, you must be committed to securing everything. It's better to offer your customers secure offsite escrow of passwords and keys, than it is to build in back doors. That way, if they don't take you up on your service, and they lose their password or certificate, they can only blame themselves.

That way you're not saying "By the way, our software is really secure, but we can break the security if you need us to", you're saying "We'll try to help you avoid costly mistakes, but our software is so secure that even we can't break the security." This is a better technical message, and a *much* better marketing message.

## 16.8 SECURITY AND MARKETECTURE

The marketing implications of choosing appropriate levels of security are far-reaching. Companies get hacked. They, and their customers, suffer real losses. In fact, in certain domains, security can be a significant perceived competitive advantage (just ask Sun's marketing department to tell you about the security of Windows). Here are some of the areas in which the security issues discussed above most directly interact with marketecture.

- *Authentication, business models, and operations.* Two key areas in which strong two-factor authentication can have a significant impact in your marketecture are your business model and your operations model. Business models based on named users should consider strong authentication; when users share user ids or passwords you're losing money. xSP operations personnel, such as an xSP system or network administrator, often complete tremendous access to sensitive data. To ensure that you're creating an environment that you're customers can trust, make certain they know that all activities on their systems are protected through strong, two or three-factor authentication.

- *Regulatory impact.* Applications in many domains are either regulated by specific standards or required to adhere to them (such as the U.S. Federal Information Processing Standards, or FIPS, for many kinds of applications). Clearly, you have to know them. Of course, it is possible that you can exceed the minimum legal requirements for a standard, which means you may be subject to technology export regulations.

- *Industry growth.* One of the biggest reasons for the success of the Internet was that it was based on open standards such as TCP/IP, HTTP, and SMTP. Over the next several years the security industry is going to see a proliferation of standards. By proactively adhering to these key standards, you're going to give your solution a better chance at being adopted by customers, primarily enterprises, who are beginning to demand standards based approaches to managing various security approaches.

- *Trust.* While complying with regulatory requirements may be required, doing so may not give your application a true competitive advantage. Your competitors are also subject to these requirements, and mere compliance provides few sustainable competitive advantages. Beyond compliance, which can be thought of as the "minimum" that must be done to earn a distinction of competence, lies *trust*: the feelings of confidence that your customers have in your character, integrity, and the ongoing quality of the relationship you've established with your customers.

  You've got a competitive advantage when your customers feel that they can entrust their data to you, secure in the knowledge that you'll maintain it safely, without allowing inappropriate access or disclosure. You've got a competitive advantage when system administrators can establish and provision user rights in such a way that sensitive corporate information is made available to only those individuals who should have such access. You've got a competitive advantage when your application seamlessly and usably integrates with digital certificate infrastructures in such a way that users can rely on them without becoming bogged down in a stream of incomprehensible technical jargon.

All of these, and more, are elements of trust. Trust is an elusive, but extraordinarily powerful element of your corporate brand. When you've got your customers trust, you have a powerful competitive advantage. Approaching security with care and building a strong, secure solution, only enhances these feelings of trust.

- *Dispute resolution.* Disputes are common in business, and software systems are known to be involved with or even cause them. Several security techniques, such as integrity and accountability, help ensure that disputes are resolved in a timely manner. Examining your business model, licensing model, and technology in-license agreements can provide you with additional ideas on how the proper application of security techniques can help you resolve disputes. For example, providing digest functions on log file entries can help resolve fighting among various technical support teams ("No, it's your bug, and we can prove it – here is our log file!). Ask your legal team for help in identifying areas in which applications of security technologies can help prevent future problems.

## Chapter Summary

- An overall security plan for your application must take into account appropriate risk factors.
- There are four main types of security are:
  - Digital Identity Management
  - Transaction Security
  - Software Security
  - Information Security
- The primary tools and techniques associated with digital identity management include:
  - Authorization – defining who can do what

    - Authentication – proof of identity. Authentication approaches vary tremendously based on whether you're dealing with an open or a closed system.
- The primary tools and techniques associated with transaction security include:
    - Auditability – proof of activity
    - Integrity – preventing tampering and alteration of data
    - Confidentiality – keeping data away from those not entitled to it
    - Accountability – holding people responsible for their actions
- The primary tools and techniques associated with software security include:
    - Tools for preventing software piracy and enforcing license compliance (see also chapters 4 and 15)
    - Binding the software to a machine or a hardware token
- The primary tools and techniques associated with information security include the same tools and techniques used for transaction security, but are far less effective. The primary technique is to make certain that the environment in which the data is placed is properly secured.
- Don't invent your own security algorithm. Chances are it will get cracked. Use a publicly available algorithm with a well-formed key.
- Never, ever, put in a back door.
- Use security to your advantage. The ultimate branding is trust, and security can help you get there.

## CHECK THIS

❑ Table 16-1 provides you with a way of checking the various kinds of security discussed in this chapter against solution. How do you fare?

| **Type of Security** | **Assessment** |
|---|---|
| Digital Identity Management | |
| How much security is needed? | |

| Type of Security | Assessment |
|---|---|
| Why do you think you need it? | |
| How have you addressed this need? | |
| Transaction Security | |
| How much security is needed? | |
| Why do you think you need it? | |
| How have you addressed this need? | |
| Software Security | |
| How much security is needed? | |
| Why do you think you need it? | |
| How have you addressed this need? | |
| Information Security | |
| How much security is needed? | |
| Why do you think you need it? | |
| How have you addressed this need? | |

**Table 16-1: Security Information Assessment**

## TRY THIS

1. Try to imagine who might attempt to defeat the security of your software. Is it a software pirate? An information thief? A temporary worker for which no background checking was performed? A competitor after company secrets? A vandal?  Write a one or two sentence biography of each. Take them one at a time and imagine that you are them. How would you attack your software?

2. Look for mention of your software and your competitors software in the hacker Usenet groups. Are there serial number generators, or "cracks", available? Are people asking for them?

3. Ask yourself “what is the worst that could happen”? Imagine the worst case scenarios. Try to put a dollar value on the damage that could be done, and try to estimate the likelihood that the worst case scenario will arise. If you’ve got a high likelihood of very expensive damage, it’s time to increase your security efforts.

4. Using at least five “typical” installations of your software, create a picture of how your customers secure your system. How do they manage access to your system? Can you help your customers improve security? How are other elements of your system managed? Even if your application is secure, there may be other routes to your data through 3rd party tools, and hackers can exploit them.

5. How secure are the components you’ve licensed?

## ABOUT RON LUNDE

Ron Lunde is an amateur orchid grower, a humorist, and an inventor of strange and typically useless things, who currently pays his bills by being employed as a System Architect for Aladdin Knowledge Systems. Ron has 19 years of software development experience as a software architect, software manager, or senior engineer for companies involved in electronic software distribution and license management, digital video editing, source level debuggers for in-circuit emulation, automatic ASIC and circuit board test generation, as well as consulting work in many other areas.

# Annotated Bibliography

CHAPTER 3:

Journey of the Software
Cockburn
Highsmith
More formal approaches, such as SEI CMM
Swiki/Twiki/CoWeb web sites.

CHAPTER 4:

EXPERIENTIAL REQUIREMENTS

CHAPTER 5:

Portions of the section *In Process We Trust* were based on similar arguments first presented in my article *In Methods We Trust*, IEEE Computer Vol. 30, No. 10 (Oct. 1997).

CHAPTER 6:

http://www.koek.net/pubs/fsl/proj.html
Email from Mike Lee

CHAPTER 11:

RIES, L. AND R. RIES *The 22 Immutable Laws of Branding: How to Build a Product or Service into a World-Class Brand*. HarperCollins. (October 1998).
This little book has a wealth of good advice on how to manage brands. If you want to know a lot about how marketing folks approach brand management, this is the book to read.

CHAPTER 17:

Brad Appleton's SCM links: http://www.enteract.com/~bradapp/acme/

GENERAL REFERENCES

CODE COMPLETE

PRACTICAL PROGRAMMER

SOFTWARE ARCHITECTURE IN PRACTICE

SOFTWARE ARCHITURE (SHAW)

More information on how architecture affects usability can be found at http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr005.pdf

Cooper – product development
Marketing texts – for general marketing stuff.

Alistair's book

Chapter 1
Cooper
Rational 4+1 View: http://www.rational.com/products/whitepapers/350.jsp

CHAPTER 10 – I FIRST WROTE ABOUT SPIKING THE ARCHITECTUER IN AUG. 1999 IN CUTTER IT JOURNAL

ABDEL-HAMID, T. K. AND S. E. MADNICK "The Elusive Silver Lining: How We Fail to Learn from Software Development Failures." *Sloan Management Review,* Vol. 32 No. 1 pp. 39–48 (Fall 1990).

ABDEL-HAMID, T. K. AND S. E. MADNICK *Software Project Dynamics: An Integrated Approach.* Englewood Cliffs, NJ: Prentice Hall, 1991.

ADELSON, B. "When Novices Surpass Experts: The Difficulty of a Task May Increase with Expertise." *Journal of Experimental Psychology: Learning, Memory, and Cognition,* Vol. 10, No. 3 pp. 483 – 95 (1984)

ADELSON, B. AND E. SOLOWAY. "The Role of Domain Experience in Software Design." *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11 pp. 1351–1360 (Nov. 1985)

ALLEN, T. J. "Organizational Structure, Information Technology, and R&D Productivity." *IEEE Transcations on Engineering Management* Vol. EM-33 No. 4 pp. 212–217 (Nov. 1986)

AMERICAN COLLEGE OF SPORTS MEDICINE *Guidelines for Exercise Testing and Prescription,* 4th ed. Malvern, PA: Lea & Febiger, 1991.

ASTLEY, W. G AND A. H. VAN DE VEN. "Central Perspectives and Debates in Organization Theory" *Administrative Science Quarterly* Vol. 28 pp. 245-273 (1983)

BABICH, W. *Software Configuration Management.* Reading, MA: Addison-Wesley, 1986

BARNES, L. B. "Managing the Paradox of Organizational Trust" *Harvard Business Review* pp.107–116 (March-April 1981)

BARNEY, J. B. "Organizational Culture: Can It Be a Source of Sustained Competitive Advantage?" *Academy of Management Review* Vol. 11 No. 3 pp. 656-665 (1986)

BARR, A., AND E. FEIGENBAUM. *The Handbook of Artificial Intelligence, Vol. 1.* Reading, MA: Addison-Wesley, 1981.

BASHSHUR, N. *Personal Communication* (1995)

BASILI, V., A. BAILEY AND F. YOUSSEFI. "Optimizing the Utilization of Human Resources: a Framework for Research" in *Software Engineering: Practice and Experience. Proceedings of the 2nd Software Engineering Conference, Nice, France* (June 1984)

BASTEIN, D. T. AND T. J. HOSTAGER. "Jazz Concert." *Communication Studies,* Vol. 43, No. 2 (Summer 1992)

BAVELAS, J. B. "Effects of the Temporal Context of Information" *Psychological Reports,* Vol. 31 pp. 695-698 (1973)

BECKER, F.D. "Technologic al Innovation and Organizational Ecology." in *Handbook of Human-Computer Interaction*. M. Helander (ed.) Eksevier Science Publishers B.V., New York

BELADY, L.A. AND M. M. LEHMAN. "A Model of Large Program Development" *IBM Systems Journal,* Vol.15, No. 3 pp. 225–52 (1976)

BENNIS, W. *On Becoming a Leader* Reading, MA: Addison-Wesley, 1989.

BENTLY, J. AND D. KNUTH. "Literate Programming" *Comm. of the ACM*, Vol. 29, No. 5 (May 1986): 364–69

BERARD, E. V. "Selecting and Using Consultants for Object-Oriented Technology" *Journal of Object-Oriented Programming* Vol. 6, No. 5 (Sept. 1993) pp. 48 – 53.

BERSOFF, E. H. AND A. M. DAVIS. "Impacts of Life Cycle Models on Software Configuration Management" *Comm. of the ACM* Vol. 34, No. 8 pp. 104– 118 (August 1991)

BERTELS, K., P. VANNESTE, AND C. DE BACKER. "A Cognitive Model of Programming Knowledge for Procedural Languages" in *Computer Assisted Learning,* Springer-Verlag, New York. 4th International Conference, ICCAL, 1992.

BLUM, B. "A Taxonomy of Software Development Methods" *Comm. of the ACM*, Vol. 37, No. 11 pp. 82–94 (November 1994)

BOEHM, B.W. "Software and Its Impact: A Quantative Assessment," *Datamation,* Vol. 19, No. 5 (May 1973)

BOEHM, B. W. *Software Engineering Economics* Englewood Cliffs, NJ: Prentice Hall, 1981.

BOEHM, B.W. "Verifying and Validating Software Requirements and Design Specifications" *IEEE Software* Vol. 1 No. 1 (Jan. 1984) pp. 75–88

BOEHM, B. W., T. E. GARY, AND T. SEEWALDT "Prototyping versus Specifying a Multi-Project Experiment" *IEEE Trans Soft Eng* Vol. 10 No.3, 1984. pp. 290-303,

BOOCH, G. *Object-Oriented Analysis and Design with Applications* 2d ed. Redwood City, CA: Benjamin/Cummings, 1994

BOOCH, G., AND J. RUMBAUGH. *Unified Method for Object-Oriented Development Version 0.8.* Santa Clara, CA: Rational Software Corporation, 1995

BOTT, H. S. "The Personnel Crunch." *Perspectives on Information Mangement,* J. B. Rochester (ed.). New York: John Wiley & Sons, 1982.

BOUGON, M.G. “Congregate Cognitive Maps: A Unified Dynamic Theory of Organization and Strategy” *Journal of Management Review* Vol. 29 No. 4 pp. 369–389 (May 1992)

BOYATZIS, R. E. AND F. E. SKELLY. “The Impact of Changing Values on Organizational Life.” in Kolb, J., Rubin I. M., and Osland, J. S. (eds) *The Organizational Reader.* Englewood Cliffs, NJ: Prentice Hall, 1991.

BRADFORD, D. AND A. COHEN. “Overarching Goals” In *Managing for Excellence* New York: John Wiley and Sons, Inc., 1984.

BRANSFORD, J. D. AND M. K. JOHNSON. “Contextual Prerequisites for Understanding: Some Investigations of Comprehension and Recall” *Journal of Verbal Learning and Verbal Behavior,* Vol. 11, pp. 717-726 (1972)

BROOKS, F. P., JR. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

BROOKS, F. P., JR. *The Mythical Man-Month: Essays on Software Engineering.* Anniversary ed. Reading, MA: Addison-Wesley, 1995.

BROWN, K. A., T. D. KLASTORIN, AND J. L. VALLUZZI. “Project Performance and the Liability of Group Harmony” *IEEE Transactions on Engineering Management* Vol. 37 No. 2 pp. 117–125 (May 1990)

CARD, S. K., T. P. MORAN, AND A. NEWELL. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

CARGILL, T. *C++ Programming Style*. Reading, MA: Addison-Wesley, 1992.

CARROLL, J., AND C. CARRITHERS. “Training Wheels in a User Interface.” *Comm. of the ACM*, Vol. 27, No. 8 (August 1984).

CARROLL J., AND A. AARONSON. “Learning by Doing with Simulated Intelligent Help.” *Comm. of the ACM*, Vol. 31, No. 9 (September 1988)

CHASE, W. G. AND H. SIMON. “Perception in Chess” *Cognitive Psychology* 4:55–81 (1973)

CHANDLER, D. C. “The Sapir-Whorf Hypothesis” http://www.aber.ac.uk/~dgc/whorf.html (1994)

CHRISTIANSEN, D. “On Good Designers.” *IEEE Spectrum,* Vol. 24, No. 5 (May 1987)

COAD, P., D. NORTH, AND M. MAYFIELD. *Object Models: Strategies, Patterns, and Applications.* Englewood Cliffs, NJ: Prentice Hall, 1995.

COLEMAN, D., P. ARNOLD, S. BODOFF, C. DOLLIN, H. GILCHRIST, F. HAYES, P. JEREMAES. *Object-Oriented Development: The Fusion Method.* Englewood Cliffs, NJ: Prentice Hall, 1994.

CONWAY, M. E. “How Do Committees Invent” *Datamation* Vol. 14 No. 10 pp. 28-31 (Oct. 1968)

CONSTANTINE, L. “Teamwork Paradigms and the Structured Open Team” In *Proceedings of Software Development ’90* San Francisco: CA, Miller Freeman Publishing. 1990.

CONSTANTINE, L. “Work Organization: Paradigms for Project Management and Organization” *Comm. of the ACM*, Vol. 36, No. 10 pp. 34–43 (Oct. 1993).

CONSTANTINE, L. *Constantine of Peopleware* Englewood Cliffs, NJ: Prentice Hall, 1995.

COPLIEN, J. AND D. C. SCHMIDT *Pattern Languages of Program Design* Reading, MA: Addison-Wesley, 1995.

CORTES-COMERER, N. “Organizing the Design Team -- Motto for Specialists: Give some, Get Some” *IEEE Spectrum*, pp. 41–46 (May 1987)

CROUCH, C. J. AND D. B. CROUCH. “The Impact of External Factors on Productivity in an Engineering Support Organization” *IEEE Transactions on Engineering Management* Vol. 35 No. 3 pp. 147–157 (Aug 1988)

CUFF, D. *Architecture: The Story of Practice.* Cambridge, MA: MIT Press, 1992.

CURTIS, B. “Substantiating Programmer Variability.” *Proceedings of the IEEE,* Vol. 69, No. 7, (1981)

CURTIS, B., R. GUINDON, H. KRASNER, D. WALZ, J. ELAM, AND N. ISCOE. “Empirical Studies of the Design Process: Papers for the Second Workshop on Empirical Studies of Programmers.” MCC Technical Report STP-260-87 (Sep. 1987)

CURTIS, B., H. KRASNER, AND N. ISCOE. “A Field Study of the Software Design Process for Large Systems.” *Comm. of the ACM,* Vol. 31, No. 11 (Nov. 1988)

DEMARCO, T., AND T. LISTER. *Peopleware: Productive People and Teams.* New York: Dorset House, 1987.

DEMARCO, T. *Controlling Software Projects.* Englewood Cliffs, NJ: Prentice Hall, 1982.

DEMING, W. E. *Out of the Crisis* Cambridge, MA: MIT, Center for Advanced Engineering Study, 1986.

DENIS, H. “Matrix Structures, Quality of Working Life, and Engineering Productivity” *IEEE Transactions on Engineering Management* Vol. EM-33, No. 3 pp. 148–156 (Aug. 1986)

DIJSKTRA, E.W. “The Structure of the "THE"-multiprogramming System.” *Comm. of the ACM,* Vol. 11, No. 5. pp. 341-346 (May 1968)

DIJKSTRA, E. W. “The Humble Programmer”(1972) in *ACM Turing Award Lectures*, ACM Press, New York 1987

DONABEDIAN, A. *The Definition of Quality and Approaches to Its Assessment.* Ann Arbor, MI: Health Administration Press, 1981.

DONNELLON, A., B. GRAY, AND M. G. BOUGON. “Communication, Meaning, and Organized Action” *Administrative Science Quaterly* Vol. 31, pp. 43-55 (1986)

DONNITHORNE, L. *The West Point Way of Leadership: From Learning Principled Leadership to Practicing It.* New York: Doubleday, 1993.

DRISCOLL, J. W. “Trust and Participation in Organizational Decision Making as Predictors of Satisfaction” *Academy of Management Journal* Vol. 21 No. 1 pp. 44–56 (1978)

ENRIGHT, J. *Change and Resiliance* Eden Prairie, MN: Wilson Learning Corp., 1984.

FAFCHAMPS, D. “Organizational Factors and Reuse.” *IEEE Software,* Vol. 11, No. 5, (Sep. 1994) pp. 31–41.

FAGAN, M.E. “Design and Code Inspections to Reduce Errors in Program Development” *IBM Systems Journal,* Vol. 15, No. 3 pp. 182–211 (Feb. 1976)

FAIRLEY, R. *Software Engineering Concepts.* New York: McGraw-Hill, 1985.

FOXALL, G. R. “Consumer Initiators: Both Innovators and Adaptors!” in *Adapters and Innovators,* M. Kirton (ed.) New York: NY. Routledge, 1994.

FREEMAN-BELL, G. AND J. BALKWILL. *Management in Engineering: Principles and Practice.* Hertfordshire, UK: Prentice Hall International, 1993.

FREEDMAN, D., AND G. WEINBERG. *Handbook of Walkthoughs, Inspections, and Technical Reviews.* New York: Dorset House, 1990.

GALBRAITH, J. *Designing Complex Organizations.* Reading, MA: Addison-Wesley, 1979.

GAMMA, E., R. HELM, R. JOHNSON, AND J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

GAUSE G., AND G. WEINBERG. *Are Your Lights On?.* New York: Dorset House, 1990.

GAUSE G., AND G. WEINBERG. *Exploring Requirements Quality Before Design.* New York: Dorset House, 1989.

GELLERMAN, S. W. “In Organizations, as in Architecture, Form Follows Function” *Organizational Dynamics* Vol. 18, No. 3 pp. 57–68 (Winter 1990)

GILB, T. *Principles of Software Engineering Management.* Avon, Great Britain: The Bath Press, 1988.

GLASS, R. L. AND R. A. NOISEUX. *Software Maintenance Guidebook*. Englewood Cliffs, NJ: Prentice Hall, 1981.

GLASS, R. L. *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall, 1995.

GLEICK, J. *Chaos: Making a New Science.* New York: Penguin Books, 1987.

GROVE, A. *High Output Management.* United Kingdom: Random House and Souvenir Press, 1983.

GOLDBERG, A., AND K. RUBIN *Succeeding With Objects: Decision Frameworks For Project Management.* Reading, MA: Addison-Wesley, 1995.

GOLDSMITH, R. E. “Creative Style and Personality Theory.” in *Adapters and Innovators,* M. Kirton (ed.) New York: NY. Routledge, 1994.

GOLSON, H. L. “The Technically-Oriented Personality in Management.” *IEEE Transactions on Engineering Management,* Vol. EM-32, No. 1 pp. 105–110 (Feb. 1985)

GOULD, J. D. “How to Design Usable Systems.” in *Handbook of Human-Computer Interaction,* M. Helander (ed.) Eksevier Science Publishers B.V., New York (1988)

GRISS, M. L. “Software Reuse: From Library to Factory.” *IBM Systems Journal,* Vol. 32 No. 4 pp. 548–566 (1993)

GROVE, A. “Management: Task-Relevant Maturity.” *Modern Office Technology,* Vol. 29 No. 8 pp. 16–20 (Aug 1984)

GUZDIAL, M. *Emile: Software-Realized Scaffolding for Science Leaners Programming in Mixed Media*. PhD Thesis. Ann Arbor, MI: University of Michigan, 1993.

HALEY, U. C. V. AND S. A. STUMPF. “Cognitive Trails in Strategic Decision-Making: Linking Theories of Personalities and Cognitions.” *Journal of Management Studies,* Vol. 26 No. 5 pp. 477–497 (Sept. 1989).

HALL, J. “Communication Revisited.” *California Management Review,* Vol. 15, No. 3 (1973)

HAMMER, M., AND J. CHAMPY. *Reengineering the Corporation: A Manifesto For Business Revolution.* New York: HarperCollins Publishers, Inc., 1993.

HANKS, K., AND J. PARRY. *Wake Up Your Creative Genius.* Los Altos, CA: Crisp Publications, Inc., 1991.

HAREL, D. *Algorithmics: The Spirit of Computing.* Wokingham, England: Addison-Wesley, 1987.

HAREL, D. “Biting the Silver Bullet: Toward a Brighter Future for System Development” *Computer* pp. 8–20 (Jan. 1992).

HARRIS, R. J., AND W. F. BREWER. “Deixis in Memory for Verb Tense.” *Journal of Verbal Learning and Verbal Behavior,* Vol. 12 pp. 590-597 (1973).

HATCH, M. J. “The Dynamics of Organizational Culture.” *Academy of Management Review,* Vol. 18 No. 4 pp. 657-693 (1993).

HATCHL, B. *Maps and More: Your Guide to Census Bureau Geography.* Washington, DC: U.S. Bureau of the Census, 1992.

HAYWORD, G., AND C. EVERETT. “Adapters and Innovators: Data From The Kirton Adaption-Innovation Inventory in a Local Authority Setting.” *Journal of Occupational Psychology,* Vol. 56 pp. 339-42 (1983)

HENDERSON, J. C., AND P. C. NUTT “The Influence of Decision Style on Decision Making Behavior.” *Management Science,* Vol. 26, No. 4 pp. 371–386 (April 1980)

HENDERSON-SELLERS, B. “The Economics of Reusing Library Classes.” *Journal of Object-Oriented Programming,* Vol. 6, No. 4 (Jul-Aug 1993)

HOARE, C. “The Emperors Old Clothes” (1981) in *ACM Turing Award Lectures.* New York: ACM Press, 1987.

HOARE, C. “An Overview of Some Formal Methods for Program Design.” *IEEE Computer,* Vol. 20, No. 9 (Sept. 1987) pp. 85–91.

HOFSTEDE, G. “Motivation, Leadership, and Organization: Do American Theories Apply Abroad?” *Organizational Dynamics,* Summer vol. 9 no. 1 pp. 42–63 (Summer 1980).

HOHMANN, L. “An Object-Oriented Data Model of Programming Plans.” Software Engineering Research Paper, Univ. Of Mich. unpublished

HOHMANN, L., M. GUZDIAL, AND E. SOLOWAY. “SODA: A Computer-Aided Design Environment for the Doing and Learning of Software Design” in *Computer Assisted Learning,* Springer-Verlag, New York. 4th International Conference on Computers and Learning (1992)

HOLDSWORTH, J. *Software Process Design.* McGraw-Hill International (UK), 1994.

HOPKINS, T. *How to Master the Art of Selling.* New York: Warner Books, 1982.

HOWLEY, E. AND D.B. FRANKS. *Health Fitness Instructor’s Handbook.* Champaign, IL: Human Kinetic Books, 1992.

HUMPHREY, W. S. “Characterizing the Software Process: A Maturity Framework.” *IEEE Software,* Vol. 5, No. 2 pp. 73-79 (March 1988)

HUMPHREY, W. S. *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

HUNT, J. W. *The Restless Organization.* Sydney: Wiley and Sons Australia Pty. Ltd. 1972.

IDEA FOUNDATION *Aerobic Dance-Exercise Instructor Manual.* San Diego, CA: IDEA Foundation, 1987.

*IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 729-1983

IOOSS, W. *RareAir: Michael on Michael.* Collins Publishers

JAY, A. “How to Run a Meeting.” *Harvard Business Review*

JACOBSON, I., M. CHRISTERSON, P. JONSSON, AND G. ÖVERGAARD *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading: MA Addison-Wesley, 1992.

JANIS, I. L. “Groupthink.” *Psychology Today,* (Nov. 1971).

JENNER, M. G. *Software Quality Management and ISO 9001.* New York: John Wiley & Sons, 1995.

JONES, C. “How Software Personnel Learn New Skills.” *IEEE Computer*, Vol. 28 No. 12 pp. 88–9 (Dec. 1995).

JOHNSON, J., R. SKOGLUND AND J. WISNIEWSKI. *Program Smarter, Not Harder.* New York: McGeaw-Hill, Inc., 1995.

JOHNSON, P. *Personal Communication* (1995)

JURAN, J. M. *Out of the Crisis.* Cambridge, MA: MIT, Center for Advanced Engineering Study, 1986.

KAHN, J. A. *Failure Construction in Organizations: Exploring the Effects of Failure Norms.* PhD Thesis. Ann Arbor, MI: University of Michigan, 1994.

KANTER, R. M. *The Changemasters.* New York: Simon and Schuster, 1983.

KAWASAKI, G. *The Macintosh Way.* San Francisco: HarperCollins Publishers, 1990.

KEIRSEY, D. AND M. BATES. *Please Understand Me: Character and Temperment Types.* Del Mar, CA: Prometheus Nemesis Book Company, 1984.

KELLEY, R. E. “In Praise of Followers.” in Gabarro, J. *Managing People and Organizations.* Boston, MA: Harvard Business School Publications, 1992.

KERNAGHAN, J. A. AND R. A. COOKE. “The Contribution of the Group Process to Successful Project Planning in R&D Settings.” *IEEE Transactions on Engineering Management,* Vol. EM-33 No. 3 pp. 134–140 (Aug. 1986)

KIRTON, M. *Adapters and Innovators*, New York: NY. Routledge, 1994.

KNOWLES, M. *The Adult Learner: A Neglected Species* 3rd ed. Houston, TX: Gulf Publishing, 1984.

LAMMERS, S. *Programmers at Work.* Redmond, WA: Microsoft Press, 1989.

LAMPSON, B. “Hints for Computer System Design.” *IEEE Software,* (Jan. 1984). pp. 11–30.

LIEBLEIN, E. “The Department of Defense Software Initiative — A Status Report.” *Comm. of the ACM,* Vol. 29 No. 8 (Aug. 1986)

LINN, M.C., AND M. J. CLANCY. “The Case for Case Studies of Programming Problems” *Comm. of ACM*, Vol. 35, No. 3, (Mar. 1992)

LITTMAN, D. C., J. PINTO, S. LETOVSKY, AND E. SOLOWAY. “Mental Models and Software Maintenaince.” *Jrnl. of Systems and Software,* Vol. 7, No. 4 pp. 341–355 (Dec. 1987)

LORENZ, M. “A Return on Your Consulting Investment: How to Hire and Outside Consultant.” *Journal of Object-Oriented Programming,* Vol. 6, No. 5 (Sept. 1993)

MACKIE, R. R. AND C. D. WYLIE. “Factors Influencing Acceptance of Computer-Based Innovations” in *Handbook of Human-Computer Interaction,* M. Helander (ed.) Eksevier Science Publishers B.V., New York (1988).

MAGUIRE, S. *Writing Solid Code.* Redmond, WA: Microsoft Press, 1993.

MAGUIRE, S. *Debugging the Development Process.* Redmond, WA: Microsoft Press, 1994.

MARTIN, J. AND W. T. TSAI. “N-Fold Inspection: A Requirements Analysis Technique.” *Comm. of the ACM,* Vol. 32 No. 2 (Feb. 1990)

MARTIN, R. M. *The Meaning of Language.* Cambridge, MA: MIT Press, 1989.

MATHIS, R.F. “The Last 10 Percent.” *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 6, pp. 705–712 (June 1986)

MCCABE, T.J. “A Complexity Measure.” *IEEE Tran. on Soft. Eng.*, Vol. 2, No. 6, (Dec. 1976)

MCCASKEY, M. B. “A Framework for Analyzing Work Groups.” in Gabarro, J. *Managing People and Organizations* Boston, MA: Harvard Business School Publications, 1992.

MCCONNELL, S. *Code Complete.* Redmond, WA: Microsoft Press, 1993.

MCCARTHY, S. *Dynamics of Software Development.* Redmond, WA: Microsoft Press, 1995.

MCCOLLUM, J. K. AND J. D. SHERMAN. “The Effects of Matrix Organization Size and Number of Project Assignments on Performance.” *IEEE Transactions on Engineering Management,* Vol. 38, No. 1 pp. 75–78 (Feb. 1991)

MEYER, B. *Object-Oriented Software Construction.* Englewood Cliffs, NJ: Prentice Hall, 1988.

MEYERS, S. *Effective C++.* Reading, MA: Addison-Wesley, 1992.

MILLER, M. L. “A Structured Planning and Debugging Environment for Elementary Programming.” in *Intelligent Tutoring Systems*. San Dieago, CA: Academic Press, 1982.

MINTZBERG, H., AND J. B.QUINN. *The Strategy Process—Concepts and Contexts*. Englewood Cliffs, NJ: Prentice Hall, 1992.

MORROW, D. G. “Prepositions and Verb Aspect in Narrative Understanding.” *Journal of Memory and Language*, Vol. 24, pp. 390-404 (1985).

NADLER, D. A., ET, AL. *Organizational Architecture: Designs for Changing Organizations.* San Francisco: Jossey-Bass, 1992.

NEWELL, A. AND H. A. SIMON. “Computer Science as Empircal Inquiry: Symbols and Search.” in *ACM Turing Award Lectures*. New York: ACM Press, 1987.

NIELSEN, J. *Usability Engineering.* New York: Harcourt Brace & Company, Publishers, 1993.

NOE, J.R. *Peak Performance Principles for High Achievers.* New York: Frederick Fell Publishers, 1984.

NORDEN, P. “Curve Fitting for a Model of Applied Research and Development Scheduling.” *IBM J. Rsch. Dev.,* Vol. 2, No. 3 (1958).

NUTT, P. C. “Influence of Decision Styles on use of Decision Models.” *Technological Forecasting and Social Change,* Vol 14, pp. 77-93 (1979)

O’CONNELL, F. *How to Run Successful Projects.* Hertfordshire, UK: Prentice Hall, 1994.

ORPEN, C. “Individual Needs, Organizational Rewards, and Job Satisfaction Among Professional Engineers.” *IEEE Trans. on Eng. Mgt.,* Vol. EM-32, No. 4 (Nov. 1985)

ORTON, J. D. AND K. E. WEICK. “Loosely Coupled Systems: a Reconceptualization.” *Acadamy of Management Review*, Vol. 15 No. 2, pp 203-223 (1990)

OZ, EZZY “When Professional Standards are Lax: The CONFIRM Failure and its Lessons.” *Comm. of the ACM,* Vol. 37 Number 10 pp. 29-36 (Oct. 1994)

PAGE-JONES, M. *Practical Project Management.* New York: Dorset House, 1985.

PAGE-JONES, M. *The Practical Guide to Structured Systems Design, 2nd ed.* Englewood Cliffs, NJ: Prentice Hall, 1988.

PAGE-JONES, M. “The Seven Stages in Software Engineering.” *American Programmer,* (July-Aug., 1990)

[PARNAS 1972A] PARNAS, D.L. “A Technique for Software Module Specification with Examples.” *Comm. of the ACM*, Vol. 15, No. 5 pp. 330–36 (May. 1972)

[PARNAS 1972B] PARNAS, D.L. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Comm. of the ACM*, Vol. 15, No. 12 pp. 1053-1058 (Dec. 1972)

PARNAS, D.L. “On the Design and Development of Program Families.” *IEEE Trans. On Soft. Eng.*, Vol. SE-2, No. 1 pp. 1-9 (March 1976)

PARNAS, D.L. “Designing Software For Ease of Extension and Contraction.” *IEEE Trans. On Soft. Eng.*, Vol. SE-5, No. 2 pp. 128-138 (March 1979)

PARNAS, D.L., AND P. C. CLEMENTS. “A Rational Design Process: How and Why to Fake It.” *IEEE Trans. on Soft. Eng.*, Vol. SE-12, No. 2 pp. 251–57 (Feb. 1986)

PASCALE, R. T. AND A. G. ATHOS. “Great Companies Make Meaning.” in Williamson, J. N. (ed.) *The Leader Manager.* New York: John Wiley and Sons, 1981.

PERRY, D.E., N. A. STAUDENMAYER, AND L. G. VOTTA. “People, Organizations, and Process Improvement.” *IEEE Software,* (July 1994). pp. 36–45.

PETERS, T., AND R. WATERMAN. *In Search of Excellence.* New York: Harper and Row, 1982.

PETROSKI, H. *To Engineer is Human: The Role of Failure in Successful Design.* New York: St. Martin’s Press, 1985.

PHILLIPS, J. R. AND A. A. KENNEDY. “Shaping and Managing Shared Values.” in Williamson, J. N. (ed.) *The Leader Manager*. New York: John Wiley and Sons, 1981.

PIRSIG, R. M. *Zen and the Art of Motorcycle Maintenance.* Bantam Books, New York: Bantam 1975.

PIRSIG, R. M. *Lila.* New York: Bantam Books, 1991.

PITTMAN, M. “Lessons Learned in Managing Object-Oriented Development.” *IEEE Software,* (Jan. 1993)

PORTER, M. E. *Competitive Strategy.* Macmillan, Inc.

POYLA, G. *How to Solve It: A New Aspect of Mathematical Method*, 2d ed. Princeton, NJ:Princeton University Press, 1957.

PRATT, J. H. "Socio-Issues Related to Home-Based Work." in *Handbook of Human-Computer Interaction,* M. Helander (ed.) New York: Elsevier Science Publishers, 1988.

PRESSMAN, R. *A Manager's Guide to Software Engineering.* McGraw-Hill, 1993.

PUTNAM, L. "A Macro-Estimating Methodology for Software Development." *Proc. IEEE COMPCON '76* (Sep. 1976)

PUTNAM, L. "A General Empircal Solution to the Macro Software Sizing and Estimating Problem." *IEEE Trans. On Soft. Eng.*, vol. SE-4, no. 4 pp. 345–361 (July 1978)

QUINN, J. B. "Managing Innovation: Controlled Chaos." *Harvard Business Review* (July – Aug. 1975).

QUINN, R. S. *Beyond Rational Management.* San Francisco, CA: Jossey-Bass, 1988.

QUINN, R. S., FAERMAN, M., THOMPSON, AND M. MCGRATH. *Becoming a Master Manager: A Competency Framework.* New York : John Wiley and Sons, 1990.

RITTEL, H., AND M. WEBBER. "Dilemmas in a General Theory of Planning." *Policy Sciences,* Vol. 4 pp. 155–69. (1973)

ROGERS, E. M. *Diffusion of Innovations* 3rd ed. New York: The Free Press, 1983.

ROWEN, R. B. "Software Project Management Under Incomplete and Ambiguous Specifications." *IEEE Tran. on Eng. Mgt.,* Vol. 37 No. 1 pp. 10–21 (Feb. 1990)

RUMBAUGH, J., M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

SACKMAN, H., W. J. ERIKSON, AND E. E. GRANT. "Exploratory Experimental Studies Comparing Online and Offline Programming Performance." *Comm. of the ACM*, Vol. 11, No. 1 pp. 3 – 11 (Jan. 1968)

SALANCIK, G. R. AND J. PFEFFER. "Who Gets Power — And How They Hold On To It: A Stratgic Contigency Model of Power." *Organizational Dynamics,* Winter, 1977

SCHEIN, E. H., "Coming to a New Awareness of Organizational Culture." *Sloan Management Review,* Vol. 25 No. 2 pp. 3–16 (Winter 1984)

SCHNEIDER, G. M, J. MARTIN, AND W. T. TSAI. "An Experimental Study of Fault Detection in User Requirements Documents." *ACM Trans. on Soft. Eng. and Methodology* Vol. 1, No. 2. (April 1992).

SCHNEIDEWIND, N.F. "The State of Software Maintenance." *IEEE Trans. on Soft. Eng.,* Vol. SE-13, No. 3 pp.303–310 (Mar. 1987)

SCHOLTES, P. R. *The Team Handbook*. Madison, WI: Joiner Associates, 1988.

SCHON, D. A. "Deutero-Learning in Organizations: Learning for Increased Effectiveness." in Williamson, J. N. (ed.) *The Leader Manager.* New York: John Wiley and Sons, 1986.

SCHULTZ, R. *Object-Oriented Project Management.* OOPSLA-95 Austin: TX, 1995.

SECMM-94-04 *A Systems Engineering Capability Maturity Model, Version 1.0.* Software Engineering Institute, (Dec. 1994).

SHNEIDERMAN, B. *Software Psychology.* Cambridge, MA: Winthrop Press, 1980.

SMITH, D. H. "A Parsimonious Definition of "Group": Toward Conceptual Clarity and Scientific Utility" *Sociological Inquiry* Vol. 37, No. 2 (Spring 1967).

SILVERMAN, B. G. "Expert Intuition and Ill-Structured Problem Solving." *IEEE Trans. on Eng. Mgt.* Vol. EM-32, No. 1 (Feb. 1985)

SIMON, H. A. *Administrative Behavior.* New York: Free Press, 1957.

SIMON, H. A. *The Sciences of the Artifical* 2nd ed. Cambridge, MA: MIT Press, 1981

SELLERS, D. *Zap! How Your Computer Can Hurt You—And What You Can Do About It.* Berkely, CA: Peachpit Press, 1994.

SOLOWAY, E., J. BONAR, AND K. EHRLICH. "Cognitive Strategies and Looping Constructs." *Comm. of the ACM*, Vol. 26, No. 11. 1983

SOLOWAY, E. AND K. EHRLICH. "Empirical Studies of Programming Knowledge." *IEEE Trans. on Soft. Eng.*, Vol. 10, No. 5 (Sept. 1984) pp. 595–609.

SOLOWAY, E., L. PINTO, S. LETOVSKY, D. LITTMAN, AND R. LAMPERT. "Designing Documentation to Compensate for Delocalized Plans." *Comm. of the ACM* Vol. 31 No. 11 (Nov 1988)

SPANGENBERG, G. *Personal Communication*, 1995.

STARBUCK, W. H. "Keeping a Butterfly and an Elephant in a House of Cards: The Elements of Exceptional Success." *Journal of Management Studies*, Vol. 30 No. 6 pp. 885–921 (Nov. 1993)

STEVENS, W., G. MYERS, AND L. CONSTANTINE. "Structured Design." *IBM Systems Journal*, Vol. 13, No. 2, 1974.

STROUSTRUP B. *The C++ Programming Language, 2nd Edition* Reading, MA: Addison-Wesley Publishing Company, 1991

STROUSTRUP B. *The Design and Evolution of C++.* Reading, MA: Addison-Wesley Publishing Company, 1994.

SWARTOUT, W. AND R. BALZER. "On the Inevitable Intertwining of Specification and Implementation." *Comm. of the ACM*, (Jul. 1982).

SZILAGYI, A. D. AND J. M. IVANCEVICH. *Organizational Behavior and Performance.* Santa Monica, CA: Goodyear Publishing Company,

TANNIRU, M. R. "Causes of Turnover Among DP Professionals." *Proceedings of the 8th Annual Computer Perosnne Research Conference*, Miami, FL, (June 1981)

THAMHAIN, H. J. AND D. L. WILEMON. “Building High Performance Engineering Project Teams.” *IEEE Transactions on Engineering Management,* Vol. EM-34 No. 3 pp. 130–137 (Aug. 1987)

THIELEN, D. *No Bugs.* Santa Monica, CA: Addison-Wesley Publishing Company,

THOMSETT, R. *Third Wave Project Management: A Handbook for Managing the Complex Information Systems of the 1990s.* Englewood Cliffs, NJ: Prentice Hall, .

THOMPSON, K. “Refelections on Trusting Trust.” *Comm. of the ACM* ,Vol. 27. No. 9 pp. 761 – 64, 1984.

TROPMAN, J. E. AND G. MORNINGTAR. *Entrepreneurial Systems for the 90’s: Their Creation, Structure, and Management.* New York: Quorum Books. 1989.

TUCKMAN, B. W. “Developmental Sequence in Small Groups.” *Psychological Bulletin* 63, pp. 384 – 399. (1965).

VINTON, D. E. “A New Look At Time, Speed, and the Manager.” *Acadamy of Management Executive,* Vol. 6 No. 4 pp. 7–16 (Nov. 1992)

WALTON, M. *The Deming Management Method.* New York: Perigree Books,

WARE, J. “How To Run a Meeting” *Harvard Business Review*

WATERMAN, R. H. JR., T. J. PETERS, AND J. R. PHILLIPS. “Structure is Not Organization.” *Business Horizons,* Vol. 23 No. 3 pp. 14–26 (June 1980)

WATERS, R. C. “KBEmacs: A Step Toward the Programmer’s Apprentice” *MIT Technical Report AI-TR 753* (1985)

WEGNER, D. M. “Transactive Memory: A Contemporary Analysis of the Group Mind.” in Mullen, B. and Goethals, G. R. (eds.) *Theories of Group Behavior,* pp 185-208 Springer-Verlag, NY (187)

WEICK, K. *The Social Psychology of Organizing* 2nd ed. New York: Random House, 1979.

WEICK, K. E. “Educational Organizations as Loosely Coupled Systems.” *Administrative Science Quaterly,vol.21 no. 1 March* 1976: 1–19.

WEICK, K. E. “Organized Improvisation: 20 Years of Organizing.” *Communication Studies,* Vol. 40. No. 4 (Winter 1989)

WEICK, K. E. “Organizational Culture as a Source of High Reliability.” *California Management Review,* Vol. 29. No. 2 pp. 112–127 (Winter 1987)

[WEICK 1993A] WEICK, K. E. AND K. H. ROBERTS. “Collective Mind in Organizations: Heedful Interrelating on Flight Decks.” *Administrative Science Quarterly,* Vol. 38 No. 3 pp. 357–381 (June 1993)

[WEICK 1993B] WEICK, K. E. “The Collapse of Sensemaking in Organizations: The Mann Gulch Disaster.” *Administrative Science Quarterly,* Vol. 38 No. 4 pp. 628–652 (Dec. 1993)

WEINBERG, G. M. *The Pyschology of Computer Programming.* New York: Van Nostrand Reinhold, 1971.

WEINBERG, G. M. AND E. L. SCHULMAN "Goals and Performance in Computer Programming." *Human Factors,* Vol. 16. No. 1 (1974) pp. 70– 77

WEINBERG, G. M. "Overstructured Management of Software Engineering." *Proceedings of the Sixth International Conference on Software Engineering,* Sept. 13-16, pp. 2–8 Tokyo, Japan (1982)

WEINBERG, G. M. *Quality Software Management Volume 1: Systems Thinking.* New York: Dorset Hourse Publishing, 1992.

WEINBERG, G. M. *Quality Software Management Volume 2: First-Order Measurement.* New York: Dorset Hourse Publishing, 1993.

WEINBERG, G. M. *Quality Software Management Volume 3: Congruent Action.* New York: Dorset Hourse Publishing, 1994.

WEISBAND, S. P. AND B. A. REINIG "Managing User Perceptions of Email Privacy." Comm. of the ACM, pp. 40-47. (Dec. 1995)

WHITESIDE, J. "Usability Engieering: Our Experience and Evolution." in *Handbook of Human-Computer Interaction*. M. Helander (ed.) New York: Elsevier Science Publishers, 1988.

WHITMAN, R.F. AND P. H. BOASE *Speech Communication.* New York: Macmillan Publishing Co., Inc, 1983.

WHORF, B. L. *Language, Thought, and Reality* (ed. J. B. Carroll). Cambridge, MA: MIT, 1956.

WILLIAMS, C. L. "Managing People, Process, and Place in High-Technology Industries." in *Managing and Designing for High-Technology Workplaces.* Van Nostrand Reinhold, 1989.

WILF, H.S. *Algorithms and Comlpexity.* Englewood Cliffs, NJ: Prentice Hall, 1986.

WINOGRAD, T. "Beyond Programming Languages." *Comm. of the ACM*, Vol. 22, No. 7 (July 1979)

WIRTH, N. "Program Development by Stepwise Refinement." *Comm. of the ACM*, Vol. 14, No. 4 pp. 221–27 (April 1971)

WOOD, R. E. AND E. A. LOCKE "Goal Setting and Strategy Effects on Complex Tasks." *Research in Organizational Behavior* Vol. 12, pp. 73-109 (1990)

YOURDON, E. AND L. CONSTANTINE *Structured Design.* Englewood Cliffs, NJ: Prentice Hall, 1979.

YOURDON, E. *Decline and Fall of the American Programmer.* Englewood Cliffs, NJ: Prentice Hall, 1992.

# APPENDIX A: RELEASE CHECKLIST

A simple checklist, such as the one contained in this appendix, can help you make certain you're not overlooking any important items associated with the product release. You should feel free to add or remove items, or phrase the questions differently, as required for your environment.

**Tracking Information**

|  |  |
|---|---|
| Product: | <insert name> |
| Version: | <x.y.z numbering> |
| FCS Date: | <FCS = First Customer Shipment> |

**Engineering / Development**

- ☐ Are version strings updated with the final version information?
- ☐ Is all debugging and test code removed from the software?
- ☐ Are all seeded defects removed from the software?

**Quality Assurance Activities**

- ☐ Is the final disposition and/or resolution of all defects complete? (NO bugs with the status of open, unassigned, fixed but not yet verified, or non-deferred analyze)
- ☐ Is the appropriate testing on the final build complete? (full regression testing, customer specified tests, smoke test, etc.)
- ☐ Is a program install from the release media onto a clean target machine complete? (CD ROM, web site, etc.)
- ☐ Is the program successfully installable? (files, registry updates, etc.)
- ☐ Is a program uninstall from the target machine complete?
- ☐ If appropriate, is an upgrade install complete?
- ☐ Are the final help files included?
- ☐ Are the final readme files included?

- ☐ Do all migration scripts pass?
- ☐ Have all supported platforms been tested and verified?

**Technical Publications**

- ☐ Release Notes
- ☐ In-line help in products – Documented in Readme, Release Note and Quickstart.
- ☐ Updated training materials reviewed by the tarchitect.

**Core Product Management Activities**

- ☐ Press release?
- ☐ email campaign to existing customers?
- ☐ Sales training?
- ☐ Pricing?
- ☐ Launch plan?
- ☐ Sales collateral (white papers, glossies, web site)?

**Knowledge Transfer – Professional Services**

- ☐ Can they install, upgrade, and integrate the system?
- ☐ How long will it take to install, upgrade, and/or integrate the system?
- ☐ Do any integrations need to redone for the installation to work?
- ☐ Are all training materials current with the release?

**Knowledge Transfer – Sales and the Channel**

- ☐ Can your sales team articulate the benefits of the new release to their existing customers?
- ☐ Can your sales team explain the advantages of the new product to new customers?
- ☐ Do they understand the complete business model, including any changes to this business model from the previous release (such as price changes, promotional discounts, and so forth)?
- ☐ Can they explain the product in the context of the overall set of solutions offered by the company.

## Knowledge Transfer – Technical Support

- ☐ Is technical support ready to support the product?

## Release Activities

- ☐ Is a final list of the files in the released product available?
- ☐ Are the date and time stamps on the released files synchronized?
- ☐ Is the final (gold) distribution media available?
- ☐ Is a virus scan of the distribution media complete?
- ☐ Is a final verification of the correct files on the distribution media complete?
- ☐ Is a backup of the build development environment under change control?
- ☐ Web site updated

# A Pattern Language For Strategic Product Management

This appendix introduces a pattern language for strategic product management (see Table P-1). What is unique about these patterns is that the allow the marketect and the tarchitect to bridge the gap that can exist between their respective disciplines.

| **Problem** | **Solution** | **Pattern** |
|---|---|---|
| How do you segment your market? | Create a visual representation of the market(s) you're targeting. | Market Map |
| What is the right frame of reference for time in your problem domain? | Identify the events and rhythms of your market/market segments. | Market Events / Market Rhythms |
| How do you ensure the right features and benefits are being created for your target market(s)? | Create a map of the proposed features and their benefits. Tie these to the market(s) you're targeting. | Feature / Benefit Map |
| How do you manage the evolution of your technical architecture? | Create a roadmap of known technology trends. Include specific and well-known changes you want to make to your architecture so that everyone can plan for them. | Tarchitecture Roadmap |

**Table P-1**

## 1.1 Applying The Patterns

Figure P-1 captures the relationship between these patterns as these patterns are applied. The order shown is an order that has worked for me. Like most diagrams that suggest an ordering, this diagram fails to show the dynamic approach to building these maps taken by successful development teams. In most cases, I recommend starting with a Market Map, primarily because there is so much confusion among product development teams regarding their specific target market. However, if you're working in a mature market, or if your marketing communication department already has a precise calendar, you may find that starting with Market Events and Market Rhythms is the best way to begin.

The most important point is that instead of arguing which pattern should be first, simply pick one and get started building out all of the patterns. This is because these patterns are part of a system of patterns. Like all complex systems, these patterns are characterized by feedback loops, shown in light gray in Figure P-1. It is almost guaranteed that the application of any given pattern will slightly modify the earlier application of a given pattern. Instead of focusing your efforts on making a "single" diagram correct try starting by identifying one market segment – just about any will do. Then add some market events or features. Try to increase your comfort that the data you need to make sound decisions will emerge, provided you keep on iterating over the results.

Discontinuous events, like a competitor releasing a new version faster than expected, a new conference forming around a new industry, or the introduction or maturation of an exciting technology are likely to motivate a series of coordinated changes in all of these diagrams. For example, in applications that rely on voice technology, the maturation of Voice XML or SALT that is first captured on the tarchitecture map may cause upward ripples in all of the other maps: You might be able to reach a new market with an awareness of these new technologies or you may be able to provide some "killer feature" by adding new support for these standards.

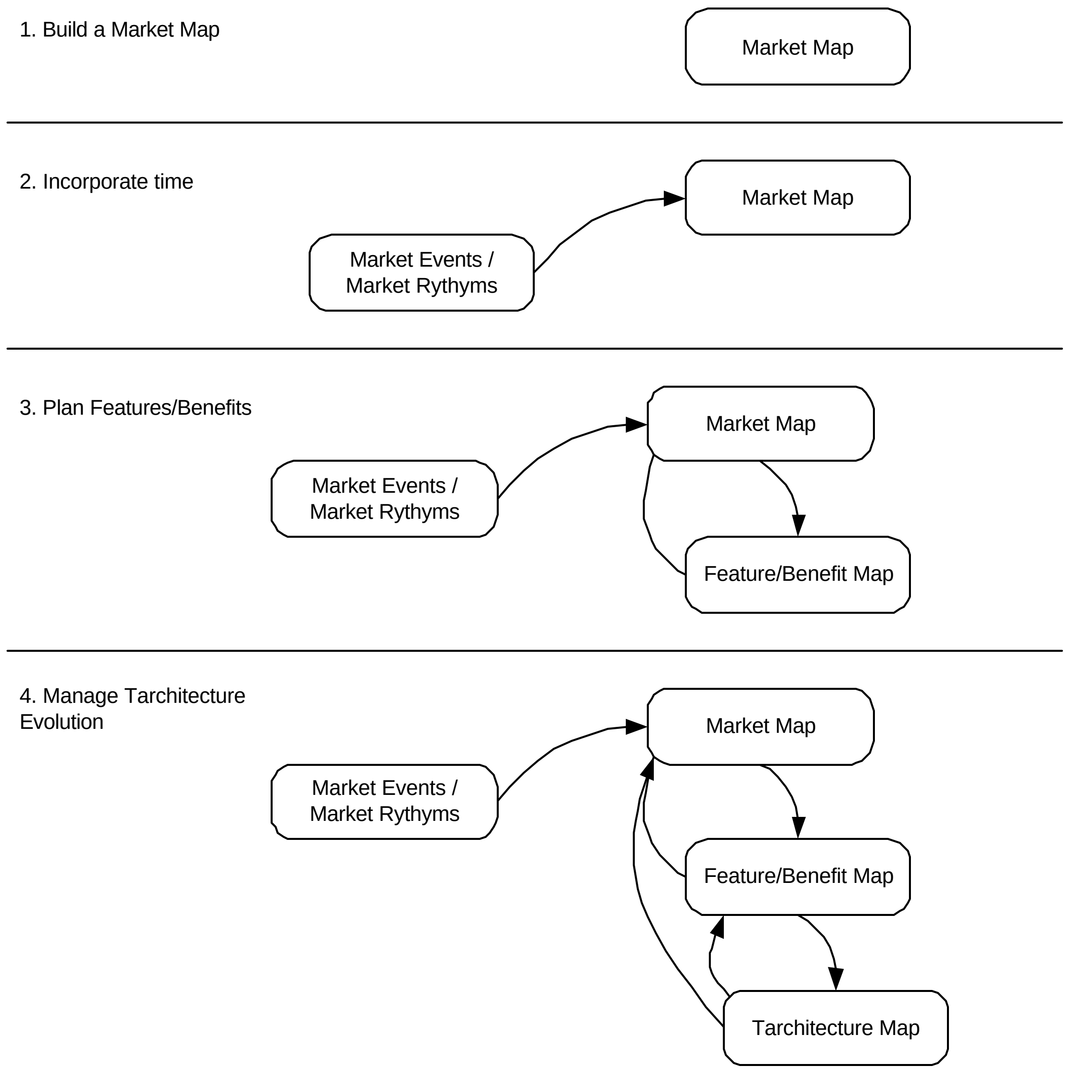

1. Build a Market Map
Market Map
2. Incorporate time
Market Events / Market Rythyms
Market Map
3. Plan Features/Benefits
Market Events / Market Rythyms
Market Map
Feature/Benefit Map
4. Manage Tarchitecture Evolution
Market Events / Market Rythyms
Market Map
Feature/Benefit Map
Tarchitecture Map

**Figure P-1**

## 1.2 CAPTURING THE RESULT

I've found that the best way to capture the result of applying these patterns is through a series of very large charts located in a publicly accessible space that is dedicated to this purpose. A stylized, condensed picture of such a display is shown in Figure P-2. The large question mark is an example of what happens when marketing identifies an unmet need that requires some new kind of technology or capability. The final row of the grid is the addition of the real schedule, or that schedule that has been communicated to customers and salespeople through such things as the product roadmap. Placing the real schedule along the bottom of the diagram makes certain everything that is being discussed in being considered from a pragmatic point of view.

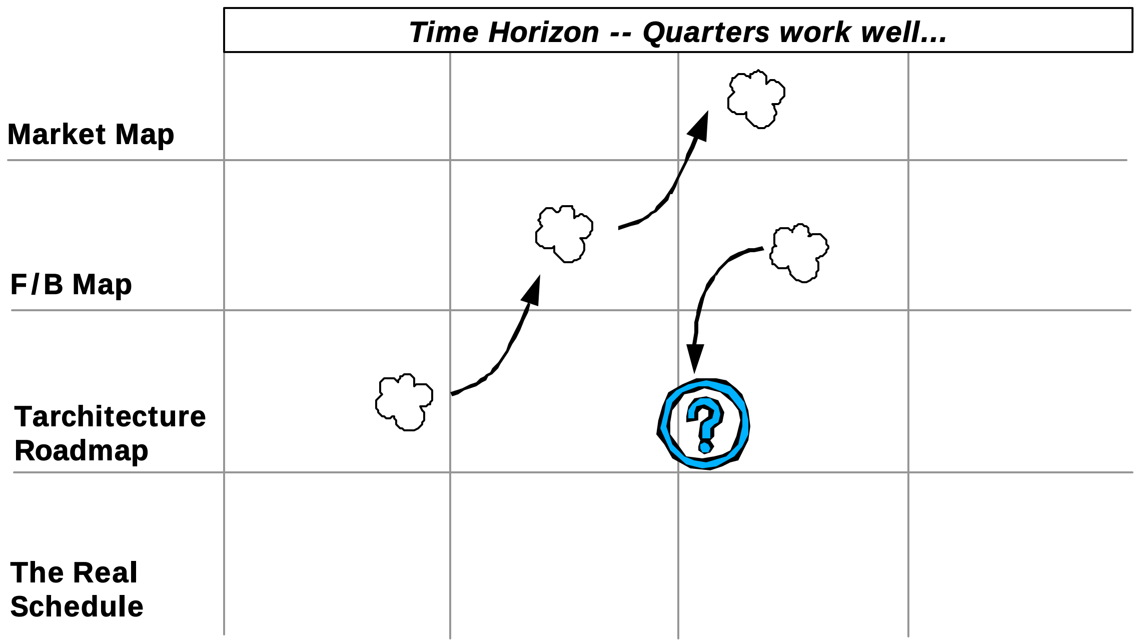

Time Horizon -- Quarters work well...
Market Map
F/B Map
Tarchitecture Roadmap
The Real Schedule

**Figure P-2**

## MARKET MAP

### CONTEXT

You have an idea for a new product and you're trying to understand the markets to which it applies.

You have an existing product and you want to make certain you're marketing it in this most effective manner possible.

### PROBLEM

How do you segment your market?

### FORCES

- Market segmentation is hard because:
    - existing markets change
    - predicting emerging markets is as much art as it is science (who knew lasers were for playing music?)
- Market segmentation is critically important.
    - If you don't segment your market you run the risk of trying to serve every market which is almost certain failure.
    - Different market segments require different solutions. You need to focus to win.
- You can't identify the most profitable segments if you don't segment well.
- You can't meet the needs of every market.
- Usability requires an understanding of the market you're trying to serve.

## SOLUTION

Segment the market by creating "classes" or "groupings" of users who share similar characteristics and/or attributes. These characteristics include critical needs, buying patterns, and various attributes that are important to you. Attributes in a consumer market might be age, household income, internet connectivity, technical literacy. Attributes in the business market might be revenue, number of employees, geography, and so forth. Name the segment on a piece of paper and then write down the most important descriptive characteristics of this segment. Large Post-It notes work well because they can be easily ordered.

Concentrate first on the actual users of your currently shipping product. If the users are not the customers (people who have the authority to purchase the product) you can address this at a later date. In this process you will identify common "points of pain" or problems these groups of users are facing. Write these down, as solving them provides input to your Feature/Benefit Map. More importantly, you must solve the problems your customers are facing.

When you begin this process try to make your segments as well-defined as possible. This will help you focus your efforts. As you examine each segment to make certain it is a viable (profitable) target, you may choose to combine it with other segments. A fine-grained approach to market segmentation will give you more flexibility in combing segments should this be needed.

Once you have created a reasonable number of segments (usually between 6 and 12) begin to order them in terms of which segment you will be addressing first relative to the actual and/or contemplated features of your product. Even before the product is finished some segments will naturally emerge as "easier" to address than others. This may be because of existing relationships (such as channel and/or customer relationships) or because it may be simply easier to build a product that pleases a certain segment. As you complete the other maps in this pattern language you can adjust the timeframes associated with the target market segments.

Provide the market map to all members of the team, especially user interface designers and quality assurance professionals. User interface designers require the Market Map to understand the needs of the customer they're trying to serve. Quality assurance professionals require the Market Map so that they can organize the testing process to make certain they are organizing the testing efforts according to key customer priorities.

## Resulting Context

Your market is segmented at a sufficient level of detail to support strategic planning. As the needs of one segment are addressed the next segment can be more precisely analyzed in preparation of the product cycle.

## Related Patterns

- Market Events/Market Rhythms
- Feature/Benefits Map

## MARKET EVENTS / MARKET RHYTHMS

### CONTEXT

You are trying to establish the market window for a given release OR an ongoing series of releases. You have a Market Map to help guide you in exploring specific market segments.

### PROBLEM

How do you choose a good target release date?

How do you establish an ongoing cycle of releases?

### FORCES

- Customers cannot usually accept a release at any time of year. For example, retailers won't accept releases of new software during the holiday selling season.
- Customers can't absorb releases that occur too quickly. Customers get frustrated and concerned if releases happen too slowly.
- Developers hate arbitrary release dates.
- Sales activities require plenty of lead time.
- Releases that are too short are almost always bad -- low quality or incomplete.
- Releases that are too long are almost always bad -- too many features, lost revenue and market opportunities.
- Organizations (developers, support, QA, release manufacturing, and others) that go too long without releasing product lose valuable skills.
- Organizations that go too long without releasing aren't fun places to work.

## SOLUTION

Identify and record the key events and rhythms that drive your market. Every domain has these events. For example, Comdex and CEBIT are important international conferences that drives many end-consumer computing devices. Consider the following when you're searching for important events:

- Conferences (technical, user group meetings and so forth);
- Competitor's press releases;
- Topics of special issues in publications that specialize in your domain and relate to your product or service.

Events that are held periodically, such as Comdex, also serve to create and establish the rhythm of the market. If you're a successful consumer electronics vendor, Comdex and CEBIT form the foundation of a yearly rhythm. This rhythm is known by all market participants and serves to drive all activities of the company. Other examples of marketplace rhythms include:

- The end-of-year Holiday season;
- In the US, the timing and structure of the school year;
- In Europe, the timing and structure of summer vacations.

Once you have identified the events and rhythms of your marketplace use them to create the timing and rhythm of ongoing releases. I've had good luck with regular release cycles of between nine and twelve months. Broad categories of software often have their own expected release cycles. High end and enterprise software systems are often designed to have major releases on twelve month boundaries with "dot" or "maintenance" releases following three to four months thereafter. Some categories of software, such as operating systems, don't really seem to have a rhythm.

The maturity of the market segment also affects the release cycle. Immature markets tend to have shorter release cycles and more events, a reflection of the learning that is happening among all market participants. Mature markets tend to have longer release cycles and more established rhythms.

Remember that you can motivate faster releases through creative licensing terms. <Need to write something about this; perhaps the "Carrots, Sticks, and Upgrades"?)

## Resulting Context

Developers are happier because they know that marketing isn't making a date out of thin air.

Commonly known dates have an energizing and engaging effect on the entire development organization.

Customers are happier because they can engage in realistic strategic planning processes. They know that "sometime" in the "third quarter" they will be receiving a new release and can plan accordingly.

## Related Patterns

- Market Map

## FEATURE/BENEFIT MAP

### CONTEXT

You want to make certain key marketing objectives match key development efforts. You have applied Market Map to identify the key market segments you're targeting.

### PROBLEM

What is the best way to capture the compelling features and their benefits for each market segment?

### FORCES

- People often think they understand the key features and benefits for a given market segment -- when they really don't.
- A feature may be a benefit to more than one market segment.
- Features that apply to multiple market segments may not provide the same perceived benefits to each.
- Developers tend to think of features (cool) while marketing people tend to think in terms of benefits (compelling advantages, reasons to purchase). This gap often results in poorly designed products and/or products that can't win in the market.
- Developers need to understand the nature and intent of *future* benefits so that they can be certain the tarchitecture is designed to meet them.

### SOLUTION

For each market segment capture the key features and benefits that will motivate them to purchase the product. Display these under the Market Map (need to provide an example).

It is crucial that you list features and benefits together. Omitting one gives the other inappropriate influence.

Choose an ordering for your map that makes most sense. I've had good results with ordering first by time and then by difficulty/complexity. Others find good results by ordering by what the market wants or what sales can really sell. Paul Germeraad from Aurigin Systems, Inc. has organized features into a "Product Tree" where the edges of the tree are the features contained in the last release. The features of the next release are placed around the edges of the tree. Paul drew lines around the features proposed for the next release. One advantage of this visualization is that the entire company can feel good about the growth of their product. The other is very practical: the expanding size of the perimeter correlates with the growth of the product development organization. As the product tree grows, so to do the needs of the team in caring and feeding the tree (including the maintenance team).

## RESULTING CONTEXT

You have a representation of the key features and their associated benefits needed to attack target market segments. This will provide the technical team with the data they need to update their tarchitecture map so that they can realize these features.

## RELATED PATTERNS

- Market Map
- Tarchitecture Roadmap

# TARCHITECTURE ROADMAP

## CONTEXT

You are building an application that is expected to have multiple, ongoing product releases. You have applied Market Map and Feature/Benefit Map to identify specific markets and the features/benefits that these markets will desire. You may have an existing architecture that supports one or more markets.

## PROBLEM

How do you manage/leverage technological change?

## FORCES

- No matter how well an application has been architectured, changes in technology can invalidate prior assumptions.
- Technologies usually appear on our horizon with enough time to accommodate them if we plan for them.
- Developers like to understand where they are headed.
- Developers like to learn new things
- Technology can enable new features that marketing may want.
- Marketing may demand features that can only be supported by adopting a new technology.
- Competitors adoption of a new technology may put you in a disadvantageous, reactive state.
- Technical people will argue over emerging technologies. Sometimes the arguments are a way of discussing the issues to learn more about them. Most of the time the only way to reach consensus is to give them plenty of time to discuss the issues.

## Solution

Create a technology roadmap that shows how your architecture will evolve. It should relate to the Market Map and Feature / Benefit Map by showing how specific technologies produce benefits that are desired by key market segments.

Review this map whenever important milestones are realized and no less than once every six months. Examples of important internal milestones include code freeze, product shipment, or when 50% of the current customers have upgraded to the most recent version. Examples of important external milestones can be a competitor releasing a new version of their product, new patent discoveries, whenever a member of the technical staff identifies a significant discontinuous technology, or whenever Market Events occur.

The creation and management of the tarchitecture roadmap requires that at least one member of the team be scanning the external environment for new developments in our field.

In the scenario where marketing has identified a feature that cannot be supported by existing technologies (either directly or because of performance/cost curves) the tarchitecture roadmap can help the team maintain their focus on periodically scanning their environment to see if new technologies have emerged that meet this need.

## Resulting Context

The good news is that you will have identified promising technical futures for your product. The bad news is that unless your team has sufficient discipline they will want to explore every possible future -- the dreaded "shiny new object" syndrome.

## Related Patterns

- Market Map
- Feature/Benefits Map