



The Addison-Wesley Signature Series

BEYOND SOFTWARE ARCHITECTURE

CREATING AND SUSTAINING
WINNING SOLUTIONS

LUKE HOHMANN



*Forewords by Martin Fowler
and Guy Kawasaki*

A MARTIN FOWLER SIGNATURE
BOOK
Martin

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the United States, please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com
Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Hohmann, Luke.

Beyond software architecture: creating and sustaining winning solutions / Luke Hohmann.

p. cm.

ISBN 0-201-77594-8 (pbk. : alk. paper)

1. Computer software. 2. Computer architecture. I. Title.

QA76.754.H65 2003

005.1—dc21

2002152562

Copyright © 2003 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0-201-77594-8

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

9th Printing May 2008

Contents

<i>Foreword by Martin Fowler</i>	<i>xxi</i>
<i>Foreword by Guy Kawasaki</i>	<i>xxiii</i>
<i>Preface</i>	<i>xxv</i>
Chapter 1 Software Architecture	1
Defining Software Architecture	1
Alternative Thoughts on Software Architecture	2
Subsystems Are Designed to Manage Dependencies	2
Subsystems Are Designed According to Human Motivations and Desires	2
Give in to Great Architectures	3
Beauty Is in the Eye of the Beholder!	4
Why Software Architecture Matters	4
Longevity	4
Stability	4
Degree and Nature of Change	5
Profitability	5
Social Structure	5
Boundaries Defined	7
Sustainable, Unfair Advantage	7
Creating an Architecture	7
Patterns and Architecture	9
Architectural Evolution and Maturation: Features versus Capabilities	9
Architectural Care and Feeding	16
Technological Currency	16
Technological Debt	16
Known Bugs	17
License Compliance	17

Principles First, Second, and Third	17
Encapsulation	18
Interfaces	18
Loose Coupling	18
Appropriate Granularity	19
High Cohesion	19
Parameterization	19
Deferral	19
Creating Architectural Understanding	20
The Team	21
Chapter Summary	22
Check This	23
Try This	23
Chapter 2 Product Development Primer	25
What Is Product Management?	25
Why Product Management Matters	26
Product Development Processes: Creating Release 1.0	26
Concept Proposal	28
Product Proposal/Business Plan	28
Development Plan	28
Development	30
Final Quality Assurance	30
Prelaunch	32
Launch	33
It Isn't Like That	33
It Is a Waterfall Process and Those Don't Work	33
It Presents All Stages as If They Were of Equal Importance	33
It Doesn't Detail Any Time	34
Where Is the Iteration?	34
It Doesn't Prescribe a Development Process	35
It Doesn't Identify the Level of Collaboration Between Groups within Stages	35
The Business Plan	35
Product Development Processes: Creating Release n.n.n	37
Augmenting the Product Development Process	37
Successive Freezing	38
Change Management Protocols	38
Recycle Bin	40
Crucial Product Management Concepts	40
The Four Ps of Marketing	40
Total Available Market, Total Addressable Market, and Market Segmentation	41
The S-Shaped Curve of Adoption	42
The Whole Product	44
Technical versus Market Superiority	45
Position and Positioning	45

Brand	47
The Main Message	47
Chapter Summary	48
Check This	49
Try This	49
Chapter 3 The Difference between Marketecture and Tarchitecture	51
Who Is Responsible for What?	51
Early Forces in Solution Development	53
Creating Results in the Short Run while Working in the Long Run	57
Projecting the Future	58
Harnessing Feedback	59
Generating Clarity	60
Working in Unison	62
Reaching Agreements	63
Making Data Available	63
Context Diagrams and Target Products	64
Chapter Summary	65
Check This	65
Try This	66
Chapter 4 Business and License Model Symbiosis	67
Common Software Business Models	69
Time-Based Access or Usage	71
Transaction	74
Metering	76
Hardware	78
Services	79
Revenue Obtained/Costs Saved	79
Rights Associated with Business Models	81
Tarchitectural Support for the Business Model	82
General Issues	83
Time-Based Access or Usage	84
Transaction	85
Metering	86
Hardware	87
Enforcing Licensing Models	87
The Honor System	88
Home-Grown License Managers	88
Third-Party or Professional License Managers	89
The Client	89
Market Maturity Influences on the Business Model	93
Choosing a Business Model	93
Chapter Summary	94

Check This	95
Try This	95
Chapter 5 Technology In-Licensing	97
Licensing Risks/Rewards	98
Contracts—Where the Action Is	101
Contract Basics	101
License Terms	102
When Business Models Collide, Negotiations Ensur	106
Honoring License Agreements	107
Managing In-Licensed Technology	108
Open-Source Licensing	108
License Fees	109
Licensing Economics	111
Chapter Summary	112
Check This	113
Try This	113
Chapter 6 Portability	115
The Perceived Advantages of Portability	115
The Business Case for Portability	116
Creating Portable Applications	119
Use an Interpreted Language	119
Use Standards-Based Persistent Storage	120
Make Business Logic Portable	120
Closer to the User Means Less Portability	120
Use XML for Standardized, Interoperable Communications between Subsystems	120
Avoid Hiding The Power of a Specific Platform in the Name of Portability	121
The Matrix of <i>Pain</i>	121
Step 1: Remove Configurations	122
Step 2: Rank-Order Configurations	123
Step 3: Make the Final Cut	123
Beware the Promises You Make	126
Chapter Summary	126
Check This	127
Try This	127
Chapter 7 Deployment Architecture	129
Deployment Choices	130
Customer Site	130
Application Service Provider	130
Managed Service Provider	130
Transactional (Web Service)	131

Customer Influences on Deployment Architectures	132
Control and Integration	132
Data Security/Privacy and Peak Loads	133
Costs and Vendor Confidence	133
Customer Skills and Experiences and Geographic Distribution	134
Corporate Influences on Deployment Architecture	136
Sales Cycle	136
Infrastructure Investment	136
Cash Flow	137
Flexibility	137
Geographic Distribution	137
Service, Not Price	138
Choosing a Software Deployment Architecture	138
Deployment Architectures and the Distribution of Work	138
The Information Appliance	140
Deployment Choice Influences on Software Architecture	140
Flexible, Parameterized, or No Integration Options	141
Upgrade Policies	141
Data Protection and Access	141
Migration Options	141
The Future of Consumer Software	142
Chapter Summary	142
Check This	143
Try This	143
Chapter 8 Integration and Extension	145
Customer Control—The Driving Force	145
Motivations for Integration/Extension	145
Layered Business Architectures: Logical Structures	147
The User Interface Layer	147
The Services Layer	149
The Domain Model Layer	149
The Persistent Data Layer	150
Variations on a Theme	151
Creating Layered Business Architectures	151
Integration and Extension at the Business Logic Layers	154
Technologies and Locus of Control	155
Integration through APIs	155
Extension through Registration	158
Integration and Extension of Persistent Data	159
Views	159
User Fields	160
Hook Tables	161
Spreadsheet Pivot Tables	162

Extract, Transform, and Load Scripts	162
Tell Them What's Going On	163
Business Ramifications	163
Professional Services	164
Training Programs	165
Certification	166
User Community	167
License Agreements	168
Managing APIs Over Multiple Releases	169
Techniques	170
Chapter Summary	170
Check This	171
Try This	172
Chapter 9 Brand and Brand Elements	173
Brand Elements	173
Names	173
Graphics, Slogans, and Other Brand Elements	177
When to Use the Trademark (™) Symbol	177
Managing In-License Brands	178
Brand Element Customizations	179
Changing Brand Elements	179
Product Areas to Change	180
QA and Change	181
Chapter Summary	181
Check This	181
Try This	182
Chapter 10 Usability	183
Usability Is about Money	183
Mental Models, Metaphors, and Usability	185
Tarchitectural Influences on User Interface Design	187
Areas of Influence	187
The Need for Speed	193
Let's Be Clear on What We're Talking About	193
What a Marketect Really Wants with Respect to Performance	195
Responding to the User	197
Performance And Tarchitectural Impact	198
Chapter Summary	201
Check This	201
Try This	202

Chapter 11	Installation	203
	The Out of Box Experience	203
	Ouch! That Might Hurt	205
	Customer Fears	205
	Installation and Architecture	206
	Forces and Choices	206
	How to Install	209
	Installation Data Collection and Precondition Verification	209
	Installation	210
	Postinstallation Confirmation	212
	Finishing Touches	212
	They Don't Read the Manual	212
	Test the Install and Uninstall	213
	Chapter Summary	214
	Check This	214
	Try This	215
Chapter 12	Upgrade	217
	Like Installation, Only Worse	217
	Upgrade Fears	217
	Making Upgrades Less Painful	221
	Choices for Painless Upgrades	221
	Market Maturity and Upgrades	224
	Chapter Summary	225
	Check This	225
	Try This	225
Chapter 13	Configuration	227
	Configurability—An Element of Usability	227
	The System Context	228
	Contextual Information	228
	Initialization versus Execution	230
	Setting the Value	230
	Setting the Right Value	231
	Configuration Parameter Heuristics	232
	Chapter Summary	233
	Check This	234
	Try This	234
Chapter 14	Logs	235
	I Want to Know What's Happening	236
	Not Just the Facts	237

Log Format and Management	239
Log Format	239
Log Management	241
Logging Standards and Libraries	242
Postprocessing Log Data	243
Logging Services	243
Chapter Summary	244
Check This	245
Try This	245
Chapter 15 Release Management	247
Yes, You Really Need This	247
Establishing a Baseline	248
Release Management	249
What You're Releasing	249
Who You're Targeting	250
Why They Want It	250
Release Identification	251
Full or Complete Releases	251
Partial Releases	252
Patch Releases	253
Variations	255
SKUs and Serial Numbers	256
SKU Management	256
Serial Numbers, Registration, and Activation	258
Release Management Influences on Tarchitecture	259
Chapter Summary	261
Check This	261
Try This	262
Chapter 16 Security	263
Viruses, Hackers, and Pirates	264
Managing Risk	265
See No Evil, Speak No Evil	265
Digital Identity Management	266
Authorization—Defining Who Can Do What	266
Authentication—Proof of Identity	266
Transaction Security	269
Auditability—Proof of Activity	269
Integrity—Preventing Tampering and Alteration of Data	269
Confidentiality—Keeping Data Away from Those Not Entitled to It	270
Accountability—Holding People Responsible for Their Actions	270

Software Security	271
Software Security Techniques	271
Software Security Costs/Benefits	273
Information Security	273
Secret Algorithms or Secret Keys?	274
Back Doors	275
Security and Marketecture	276
Areas of Interaction	276
Chapter Summary	278
Check This	279
Try This	280
Appendix A Release Checklist	281
Appendix B A Pattern Language for Strategic Product Management	285
Applying The Patterns	286
Capturing the Result	287
Market Map	288
Market Events/Market Rhythms	289
Feature/Benefit Map	291
The Tarchitecture Roadmap	292
References	295
Bibliography	297
About the Author	301
Index	303

This page intentionally left blank

Foreword by Martin Fowler

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

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

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

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

Martin Fowler,
Series Editor

This page intentionally left blank

Foreword by Guy Kawasaki

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

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

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

Guy Kawasaki,
CEO
Garage Technology Ventures

This page intentionally left blank

Preface

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

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

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

quality assurance, technical publications, and first- and second-line support organizations. I've managed teams and projects across multiple cities and continents.

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

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

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

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

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

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

These barriers prevent both groups from creating winning solutions. I cringe when engineers take only a *technology* view without due consideration of *business*

issues, or when marketing people make “get-me-this-feature” demands without due consideration of their underlying technical ramifications. When either side takes a position without due consideration of its impact, the likelihood of creating and sustaining a winning solution drops dramatically.

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

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

Acknowledgments

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

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

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

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

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

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

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

Luke Hohmann
luke@lukehohmann.com

Chapter 3

The Difference between Marketecture and Tarchitecture

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

Who Is Responsible for What?

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

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

Marketecture is the business perspective of the system’s architecture. It embodies the complete business model, including the licensing and selling models, value propositions, technical details relevant to the customer, data sheets, competitive differentiation, brand elements, the mental model marketing is attempting to create for the customer,

and the system's specific business objectives. Marketecture includes—as a necessary component for shared collaboration between the marketects, tarchitects, and developers—descriptions of functionality that are commonly included in marketing requirements documents (MRDs), use cases, and so forth. Many times the term *whole product* is used to mean marketecture.

The \$50,000 Boolean Flag

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

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

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

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

Early Forces in Solution Development

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

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

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

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

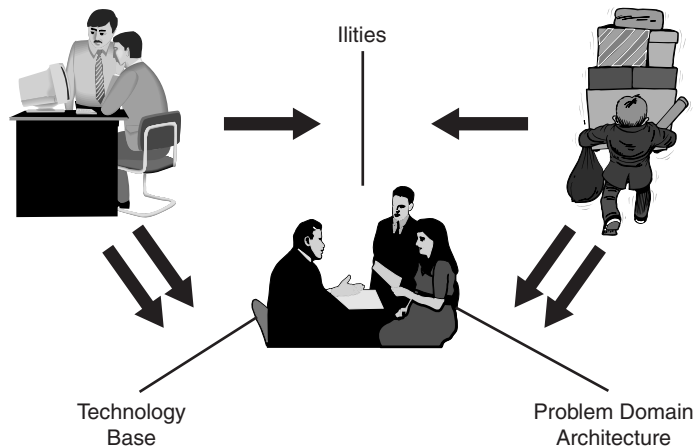


FIGURE 3-1 Forces shaping software architectures

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

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

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

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

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

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

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

Sometimes “The Hard Way” Is the Only Way

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

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

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

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

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

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

Bug Severities, Priorities, and Normalization

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

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

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

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

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

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

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

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

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

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

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

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

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

World-class marketects approach their task from a perspective of time that easily distinguishes them from those less skilled. Instead of listening to what customers want now (easy), they extrapolate multiple streams of data, including current requests, to

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

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

Projecting the Future

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

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

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

Examples of discontinuities abound. Consider an application originally designed for a single language. If this language becomes successful, the marketect may include

internationalization in her map, but the corresponding entry in the tarchitecture map is often a major discontinuity, especially if the team is not experienced in building such applications. Another example is new business models envisioned by the marketecture. It is doubtful that the original tarchitecture was planned with them in mind, so they should be noted as tarchitectural discontinuities. In a similar vein, known problems with the tarchitecture that grate against developer sensibilities should be identified so that they can be addressed in future revisions.

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

Harnessing Feedback

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

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

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

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

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

What if They Say Something They Shouldn't?

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

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

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

Generating Clarity

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

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

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

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

Managing Cultural Differences in Software Development

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

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

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

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

Working in Unison

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

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

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

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

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

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

Reaching Agreements

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

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

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

Making Data Available

Visibility to maps and features is crucial. None of the approaches I've described for capturing and planning for the future are much good if the data are hidden. Get this information into a forum where everyone can share it. Some teams accomplish this

through an intranet or a Lotus Notes database. Other teams are experimenting with Swikis, Twikis, or CoWebs with good results, although my own experience with these tools has been mixed and is heavily influenced by team culture. Other teams simply make lots of posters available to everyone. Visibility, in turn, is built on top of a corporate culture founded on trust and openness. Putting up posters merely to look good won't fool anyone. Making a real commitment to visibility—and dealing with the inevitable issues your project team members will raise—is a powerful tool to ensure marketect and tarchitect cooperation.

Context Diagrams and Target Products

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

Context diagrams are useful for a number of reasons.

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

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

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



Chapter Summary

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

Check This

- We have a marketect.
- We have a tarchitect.
- We have a bug database that classifies each bug according to severity and priority.

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

Try This

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

Index

A

Accountability, 270–271
Activation, software, 258, 271–272
Adoption, S-shaped curve of, 42–44
Advertising, 41
Ahn, Myron, 1, 235
Annual software license, 71–72
 characteristics of, 82
 enforcement, 88–93
America Online (AOL), 224
APIs
 exposure of, 156–157
 integration through, 155–158
 and license management, 91
 managing over multiple releases, 169–170
 stabilization of, 158
 upgrading of, 219
Application service provider (ASP), 130, 133
Architecture
 advantage conveyed by, 7
 alternative thoughts on, 2–4
 creating, 7–9
 criteria for replacement of, 6, 8
 definitions of, 1–2
 deployment, 129–144
 evaluation of, 8–9
 evolution and maturation of, 9–16

 and installation, 206–209
 marketing. *See* Marketecture
 modular, 227–228
 patterns, 9
 principles, 17–19
 subsystems, 2–3
 team for, 21–22
 technical. *See* Tarchitecture
 tips for, 16–17
 views of, 20–21
Artifact, defined, 248
Audit trails, 85
Auditability, in secure system, 269
Augmented product, 44, 45
Authentication, 86
 in closed systems, 267
 in hybrid systems, 268–269
 and marketecture, 276
 in open systems, 267–268
Authorization, 266
 and installation, 210
Automatic forwarding, of logs, 242

B

Bach, James, 56, 126
Back doors, 275–276
Backend integration, 92

- Background processing, 200
- Backward compatibility, 170
- Behavior tracking and auditing logs, 236
 - suggested content for, 239
- Bootstrapping, 228
- Boundaries, defining, 7
- Brand
 - changing elements of, 179–181
 - customization of, 179
 - elements of, 173–178
 - importance of, 47
 - in-license, 178
 - legal issues, 177–178
- Bugs
 - dealing with, 17
 - fixes, charging for, 255
 - prioritizing, 56–57
- Build number, 252, 253
- Bundles, in business model, 69
- Business model, 40
 - in business plan, 36
 - choosing, 93–94
 - common types of, 69–81
 - defined, 67
 - enforcement issues, 83
 - ideal, 68
 - and installation, 207
 - and license model, 67
 - market maturity and, 93–94
 - multiple, 70
 - and pricing model, 75–76
 - rights associated with, 81–82
 - tarchitectural support for, 82–87
 - verification, 84
- Business plan, 28
 - components of, 35–36
 - importance of, 35
 - updating of, 37
- C**
- Caches, 199–200
- Capabilities
 - defined, 12
 - vs. features, 11
- Capacity, defined, 193
- Capacity planning logs, 236
 - suggested content for, 238
- Caps . ini file, 231–232
- Cardinality, 187
- Cash flow, modeling of, 137
- Certificate revocation, 268
- Certification, 166–167
 - digital, 268
 - upgrade, 219
- Change protocols, 38–39
- Channel strategy, in business plan, 36
- Code tags, 15
- Cohesion, 19
- Colors, brand identification by, 177
- Common Log Format, 242
- Compaction, of logs, 243
- Compatibility, configuration for, 229
- Compensating transactions, 192
- Competitive analysis, in business plan, 36
- Complete release, 249
- Component
 - defined, 248
 - name of, 174–176
 - physical location of, 174
 - prerequisite relationships, 248
 - ripple upgrade, 218
- Concept proposal, 28, 37
- Conceptual models, 186
- Concurrent resource management, 76–77
 - characteristics of, 82
- Confidentiality, in secure system, 270
- Configuration parameters, 229
 - heuristics for, 232–233
 - setting, 230–232
- Configuration
 - during execution, 230
 - during upgrades, 223
 - during initialization, 230
 - streamlining of, 229
 - system context of, 228–230
 - usability and, 227–228
 - value setting, 230–232
- Consumptive resource management, 77–78
 - characteristics of, 82

- Context diagrams, 64–65
- Contracts
 - basics of, 101
 - and conflicts, 106–107
 - honoring, 107–108
 - license terms, 102–106
 - payment issues, 109–111
- Contractual volume licensing, 73–74
- Controlled release, 250
- Copy protection, 84
- Cost analysis, in business plan, 36
- Costs saved, business model based on, 79–80
- Coupling, 18
- Crackers, counteracting, 92
- Critical risks, in business plan, 36
- Cross-platform code. *See* Portability
- CRUD operations, 149
- Cultural differences, and architecture, 61
- Cunningham, Ward, 13
- Customers
 - characteristics of, 134–135
 - and deployment architecture, 132–135
 - feedback from, 59–60, 168
 - and installation, 205–206
 - and integration and extension, 163
 - message to, 47–48
 - relations with, 51–59
 - software control by, 132–133, 145–147
 - types of, 42–44
 - and upgrades, 219, 221
 - in user community, 167–168
 - vendor confidence of, 133–134
 - working with, 12–13
- Customization information, upgrades and, 223

- D**
- Data capture, for business models, 83
- Data migration
 - options for, 141
 - procedure for, 221, 223
 - in upgrading process, 218–219, 224
- Data retention, in upgrading process, 219
- Data security, 133, 141
 - case study of, 135
- Databases, software upgrades and, 221, 223
- Debug label, for log files, 241
- Debugging logs, 236
 - suggested content for, 238
- Degradation, sources of, 13
- Delivery
 - promptness of, 16
 - of usable product, 15
- Deployment architecture
 - case study of, 135
 - choices of, 130–131
 - choosing, 138, 139
 - defined, 129
 - customer influences on, 132–135
 - corporate influences on, 136–138
 - geographical issues, 134, 137–138
 - hybrid, 132
 - information appliance and, 140
 - maintenance of, 138–139
 - and software architecture, 140–142
 - variables in, 141
- Design documents, 29
- Development
 - plan for, 28–29
 - of product, 30
 - parallel development, 153
 - rhythm of, 15
- Digest function, 269
- Digital certificates, 268
- Digital identity management, 264
 - authentication, 266–269
 - authorization, 266
- Digital signature, 270, 271
 - for software license, 89, 272
- Directories, locating, 228
- Dispute resolution, security and, 277–278
- Distribution
 - defined, 249
 - channels, 41
 - electronic, 256
 - and software activation, 258
- Domain model, 149–150
- Dongle, 273
- Dot release, 252
- Dynamic logging, 241

E

E-mail, security for, 268–269

Early adopters
 and business model, 93
 in adoption cycle, 43
 and upgrades, 224

Early majority
 in adoption cycle, 43

Encapsulation, 18

Encryption, 269–270

Enterprise application providers (EAPs), 131

Entropy reduction, 14
 techniques for, 15

Error label, 241

Error logs, 236
 suggested content for, 238

Error recovery logs, 236
 suggested content for, 238

Error response, 189

ETL (extract, transform, load) scripts, 162–163
 charging for, 163
 in upgrades, 221

Exceptions, logging of, 242

Execution, configuration during, 230

Executive overview, in business plan, 35

Expected product, 44, 45

Extension, 36
 at business logic layers, 154–159
 layered business architectures and, 147–154
 and locus of control, 155
 motivations for, 145–147
 and persistent data, 159–163
 through registration, 158–159

F

Failure, dealing with, 8

Failure recovery, 192
 and performance, 199

Feature/benefit map, 58, 291–292

Features, 69
 vs. capabilities, 11
 customer feedback in determining, 12–13
 documenting, 10, 12

learning curve for, 220
 as marketing tool, 10
 removal of, 222
 and technological debt, 13

Feedback, 59
 and features, 12–13
 unexpected, 60
 to user, 187–188, 197–198

Fees
 negotiation of, 111
 types of, 109–111

Files, keeping track of, 228

Financial analysis, in business plan, 36

FIPS (U.S. Federal Information Processing Standards), 277

Flat files, design of, 240

Flexibility
 customer requests for, 11
 in deployment architecture, 137
 reducing through successive freezing, 38

4 P's of Marketing, 40

Fractional release, 249

Freezing, successive, 38

Full release, 249
 identification of, 251

Future of commercial software, 142

G

General release, 250

Generic product, 44

Germeraad, Paul, 292

Globally unique identifiers (GUIDs), 85

Granularity, choosing, 19

Graphics, brand identification by, 177

H

Hackers, 271

Hardware
 business models based on, 78–79, 82, 87

Hardware binding, 259, 272–273

Hardware upgrades, 218

Honor system, 88

Hook tables, 161–162

I

Icons, brand identification by, 177
Identified resource management, 77
 characteristics of, 82
Ilities, 53, 54
 dealing with, 56, 83–84
Impact analysis, in business plan, 36
Info label, 241
Information appliance, 140
Information security, 133, 135, 141, 264
 importance of, 273–274
 techniques of, 274
Infrastructure investment, effect on deployment
 architecture, 136–137
Initialization, configuration during, 230
Injection approach, to license management, 90–91
Innovations, adoption of, 42
Innovators, in adoption cycle, 43
Installation
 and architecture, 206–209
 and business model, 207
 case study of, 207
 caveats regarding, 210–212
 confirmation of, 212
 customer information for, 210–211, 212–213
 customer issues with, 205–206
 dangers of, 204
 developers and, 208
 environment issues, 208
 interruptibility of, 211
 licensing issues, 207
 out-of-box, 203–205
 preconditions for, 209–210
 procedures for, 209–212
 responsibilities for, 207–208
 roles in, 208
 testing, 213–214
Integration
 APIs and, 155–158
 backend, 92
 at business logic layers, 154–159
 example of, 147
 layered business architectures and, 147–154
 and locus of control, 155

 motivations for, 145–147
 and persistent data, 159–163
Integrity, in secure system, 269–270
Interfaces, 18
Internationalization, 189–190
 case study of, 190
Internet business service providers (IBSPs), 131
Interpreted language, for portable applications,
 119–120

K

Krutchen, Philippe, 20

L

Labels, in logs, 241
Laggards, in adoption cycle, 43
Lampson, Butler, 199
Late majority, in adoption cycle, 43
Latency, defined, 193
Layered business architectures
 creation of, 151–154
 domain model layer, 149–150
 illustrated, 148
 parallel development of, 153
 persistent data layer, 150–151
 services layer, 149
 spiking and, 151–152
 user interface layer, 147–149
License agreements, 167–168
License generator, 89
License managers
 branding and user interface issues, 92
 business model support of, 91
 capabilities of, 92–93
 client, 89–90
 home-grown, 88–89
 license format and content, 92–93
 license distribution, 93
 operational environment issues 92
 platform and operating system support of, 92
 security issues of, 92
 server component of, 90
 third-party, 89, 90

- License model, 67
 - enforcement of, 87–93
 - types of, 71–80
 - verifying parameters of, 84
 - Licenses
 - from component vendors. *See* Technology in-licensing
 - digitally signed, 272
 - Lightweight Directory Access Protocol (LDAP), 266
 - Limited release, 250
 - Localization, as user issue, 190
 - log4j, 242
 - Logging levels, 241–242
 - Logical view, of system, 20
 - Logs
 - APIs for, 242
 - audiences for, 234
 - automatic forwarding of, 242
 - case study of, 278
 - data categories in, 237
 - of exceptions, 242
 - format of, 239–240
 - information contained in, 237–239
 - installation, 210–211
 - logging libraries, 242
 - logging services, 243–244
 - management of, 241–242
 - postprocessing data for, 243
 - purposes for, 235–237
 - removable, 242
 - repurposing, 243
 - security of, 242
 - standards for, 242
 - vs. trace statements, 239
 - Longevity, of architecture, 4
 - Loose coupling, 18–19
 - Lunde, Ron, 218, 280
- M**
- Machine binding, 272
 - Main message, 47–48
 - Maintenance release, 252
 - Major release, 252
 - Managed release, 250
 - Managed service provider (MSP), 130–131, 133
 - Market analysis, in business plan, 36
 - Market events/market rhythms, 58, 289–291
 - Market map, 58, 288–289
 - Market segment
 - defined, 41
 - niches within, 42
 - Market segment preferences, 156
 - Marketecture
 - case study of, 52, 55
 - and certification, 166–167
 - clarity in, 60–62
 - defined, 51–52
 - and licensing agreements, 168–169
 - and professional services, 164
 - projecting the future, 58–59
 - skills involved in, 57–58
 - and tarchitecture, 54, 55, 57, 68
 - and training programs, 166
 - and user community, 167–168
 - Marketing communication, 41
 - Marketing requirement documents (MRDs), 26, 29
 - and configuration management, 248
 - formality, 29, 37, 39, 60
 - importance of, 37
 - Marketing strategy, in business plan, 36
 - MD5 digest function, 269
 - Mental models, 185–186
 - Metaphor
 - example, 186
 - importance of, 186
 - relationship to marketecture, 186
 - Metering, 76
 - business models based on, 76–78
 - characteristics of, 82
 - methods of, 86–87
 - and tarchitecture, 86–87
 - user authentication, 86
 - Milestone payments, 110
 - Minor release, 252
 - Modular architecture, 227–228
 - Module release, 249

Moore, Geoffrey, 43

Multiplatform development, 32

N

Names

- attributes of, 175

- in configuration and log files and system messages, 176

- conventions for, 156

- in deployment model, 174

- internationalization of, 176

- of key components, 174, 176

- responsibility for, 174

- volatility of, 176–177

Network availability, 192

Niche markets, 41

O

OEM market, 74

One-way hash, 269

Open source software (OSS), 79

- licensing of, 108–109

Operational status logs, 236

- suggested content for, 239

Out of box experience (OOBE), 203–205

P

Parallel development of layered architecture, 153

Parameterization, 19

Partial release, 249

- identification of, 252–3

Partner preferences, 156

Password security, 274

Patch release, 249

- identification of, 253–254

Pay after use software, 73

- characteristics of, 82

Peak loads, 133

Per-thread logging, 241

Per-use software license, 73, 207

Percentage of revenue fees, 110

Performance

- defined, 193

- and marketecture, 195–197

- and tarchitecture, 198–201

- terminology of, 193–195

- user response, 197–198

Performance parameters, 229

Performance tuning logs, 236

- suggested content for, 238

Perpetual license, 71

- characteristics of, 82

Persistent data

- business logic and, 160

- ETL scripts, 162–163

- hook tables, 161–162

- spreadsheet pivot tables, 162

- user fields, 160–161

- views, 159–160

Persistent data layer, 150–151

Persistent storage, standards-based, 120

Physical view, of system, 20

Piracy, 265

- reducing, 84, 259

- techniques of, 272

Pivot tables, 162

Place (in marketing), 41

Platform preferences, 155–156

Portability

- applications and, 119–121

- business case for, 116–119

- case study of, 118

- configurations and, 122–126

- facilitation of, 121–126

- and installation, 211

- motivations for, 116

- importance of, 228

- perceived advantages of, 115–116

- platform issues, 121

Portable applications

- business logic, 120

- language for, 119–120

- obstacles to, 120

- storage issues, 120

- XML in, 120–121

- Positioning, 45, 47
 - effectiveness of, 46
 - Potential product, 44
 - Prelaunch, 33–34, 37
 - Prepaid fees, 109
 - Pricing model, 40–41, 75–76
 - Private key encryption, 270
 - Problem domain, 53
 - Process view, of system, 20
 - Product
 - augmented, 44, 45
 - branding of, 47
 - distribution of, 41
 - expected, 44, 45
 - generic, 44
 - importance of, 40
 - potential, 44
 - pricing of, 41
 - promotion of, 41
 - target, 45, 64–65
 - technical vs. market superiority of, 45
 - whole, 44–45
 - Product description, in business plan, 36
 - Product development
 - preparation for, 34–35
 - real-world, 33–35
 - Product development processes
 - augmentation of, 37–40
 - enumerated, 26–33
 - real-world, 37
 - Product differentiation, in business plan, 36
 - Product extensions and futures, in business plan, 36
 - Product positioning, in business plan, 36
 - Product proposal, 28, 37
 - Professional services, marketecting of, 164
 - Profilers, 199
 - Profitability, architecture and, 5
 - Program family, defined, 248
 - Project management
 - business plan, 35–36
 - concepts in, 40–48
 - defined, 25
 - importance of, 26
 - interaction with engineering, 27
 - and product development, 26–33
 - protocol changing, 38–39
 - real-world view of, 33–35, 37–40
 - Product management patterns, 285
 - applying, 286–287
 - feature/benefit map, 291–292
 - market events/market rhythms, 289–291
 - market map, 288–289
 - results of, 287–288
 - tarchitecture roadmap, 292–294
 - Promotion, in marketing, 41
 - Public key encryption, 270
- ## Q
- Quality assurance
 - aspects of, 30–31
 - case study of, 31–32
 - importance of, 30
 - involvement in training programs, 165
 - in release management, 247
 - and technology in-licensing, 99
- ## R
- Rational 4+1 model, 20–21
 - RC4 algorithm, 270
 - Recycle bin, importance of, 40
 - Registration, software, 258
 - extension through, 158–159
 - Release, defined, 249
 - Release checklist
 - for core product management, 282
 - for engineering/development, 281
 - knowledge transfer, 282–283
 - for professional services, 282
 - for quality assurance, 281–282
 - release activities, 283
 - for sales, 282–283
 - technical publications, 282
 - for technical support, 283
 - tracking information, 281
 - Release management
 - audiences for, 250

- defined, 247
- guidelines for, 260–261
- importance of, 247–248
- influences on tarchitecture, 259–261
- product in, 249
- release identification, 251–256
- serial numbers, 258
- SKU management, 256–258
- software activation, 258
- Reliability, defined, 193
- Remittance requirements, 83
- Rental software, 72
 - characteristics of, 82
- Replacement, indications for, 6
- Reporting requirements, 83
- Request
 - cancellation of, 190–191
 - undoing of, 191–192
- Response time, defined, 193
- Revenue forecast, in business plan, 36
- Revenue obtained, business model based on, 79–80
- Reversion, 220
- Revision, defined, 248
- Rework, in upgrading, 217–218
- Rijndael encryption, 270
- Ripple upgrades, 218
- Risk management, 265
- Role based access control (RBAC), 266
- Royalty market, 74

- S**
- S-shaped curve of adoption, 42–44
- Sales cycle, 136
- Scalability, defined, 193
- Security
 - algorithms for, 274–275
 - challenges to, 264–266
 - dangers of back doors, 275–276
 - digital identity management, 264, 266–269
 - in dispute resolution, 277–278
 - importance of, 263–264
 - and industry growth, 277
 - information security, 264, 273–274
 - of logs, 242
 - and marketecture, 276–278
 - need for, 265–266
 - regulatory issues, 277
 - risk management, 265
 - and session data, 156
 - software security, 264, 271–273
 - transaction security, 264, 269–271
 - and trust, 277
- Self-service operations, 200
- Serial numbers, 258
 - importance of, 258–259
 - management of, 55
 - as security measure, 271–272
- Service-based business models, 79
 - characteristics of, 82
- Services layer, 149
- SHA1 digest function, 269
- Shared resources, 192
- Signature, digital, 270, 271, 272
- SKUs, 256
 - allotment of, 258
 - assigning, 256–257
- Sleepycat Software, 80
- Slogans, brand identification by, 177
- Smith, Dave, 235
- Social structure, architecture and, 5
- Software activation, 258
 - importance of, 259
 - as security measure, 271–272
- Software security, 264
 - costs and benefits of, 273
 - hardware binding, 259, 272
 - protecting validation code, 272
 - techniques of, 271–273
- Solution development, 53
- Speed
 - network, 192
 - performance, 193–197, 198–201
 - user response, 197–198
- Spiking
 - defined, 151
 - illustrated, 152

- Spiking *continued*
 - importance of, 153
 - incremental development using, 151–152
 - Splash screens, brand identification by, 177
 - Spreadsheet pivot tables, 162
 - SSL (Secure Socket Layer), 270
 - Stability, of architecture, 4
 - Standards-based persistent storage, 120
 - Strategic fit, in business plan, 35
 - Studios, in business model, 69
 - Subcomponents, managing, 206
 - Subscription software, 72–73
 - characteristics of, 82
 - Subsystems
 - integrity in, 3
 - motivators for design of, 2–3
 - need for, 2
 - Successive freezing, 38
 - Suites, in business model, 69
 - Superiority, technical vs. market, 45
 - Support model, in business plan, 36
 - SWOT analysis, 36
 - Synchronization, of logs, 243
 - System configuration management logs, 236
 - suggested content for, 239
- T**
- Tarchitecture
 - and business model, 82–87
 - case study of, 52, 55
 - costs of, 84
 - cultural issues in, 61
 - data availability, 63–64
 - and decision making, 63
 - defined, 51
 - and marketecture, 54, 55, 57, 68
 - performance and, 198–201
 - projecting the future, 58
 - release management and, 259–261
 - teamwork and, 62–64
 - and training materials, 166
 - and user interface design, 187–193
 - Tarchitecture roadmap, 58, 292–294
 - Target product, 45
 - context diagrams and, 64–65
 - Team
 - building of, 21–22
 - cultural issues of, 61
 - working as, 62–64
 - Technological currency, 16
 - Technological debt, 14
 - causes of, 16–17
 - dealing with, 14–15
 - Technology in-licensing, 17
 - access to source code, 106
 - case study of, 103, 107
 - conflict resolution, 106–107
 - contracts for, 101
 - duration of, 102–103
 - economic issues, 110–112
 - exclusivity issues, 104
 - geographical scope of, 103
 - honoring agreements, 107–108
 - and installation, 206
 - management of, 108
 - marketing issues, 106
 - management of brands, 178
 - noncompete agreements, 105–106
 - OSS (Open Source Software) and, 108–109
 - payment and renewal terms, 105, 109–110
 - risks and rewards of, 98–101
 - reasons for, 97
 - restrictions on, 105
 - scope of, 102, 104
 - sublicensing, 104
 - termination of, 104
 - training and development issues, 111
 - Test-driven design, 31
 - Threading, 199
 - Throughput, defined, 193
 - Time-based access or usage, 71–74
 - characteristics of, 82
 - and tarchitecture, 84–85
 - Timeouts, 192
 - Total addressable market, defined, 41
 - Total available market, defined, 41
 - Trace statements, 239

Trademark symbol, 177
Trademarks, 173
 proper use of, 178
Training programs, 165–166
Transaction-based business models, 74–76
 characteristics of, 82
 and tarchitecture, 85–86
Transaction security, 264
 accountability, 270–271
 auditability, 269
 confidentiality, 270
 integrity, 269–270
Transaction support, 188–189
Transactional deployment, 131
Transactional volume licensing, 73–74
Transactions
 business models based on, 74–76, 82
 characteristics of, 86
 defining, 85
 identification of, 85
 relation to business model, 85
Trialware, 11

U

Uninstallation, 213–214
Universally unique identifiers (UUIDs), 85
Update release, 249
Upgrades
 APIs in, 219
 certification of, 219
 and configuration and customization information, 223
 customer readiness for, 221
 customer resistance to, 219
 data migration during, 218–219, 221, 223
 data retention during, 219
 facilitating, 221–224
 fears about, 217–220
 and license compliance, 17
 and license rights, 67, 81–82
 market maturity and, 224–225
 new features and, 220
 number and timing of, 221
 from old versions, 223, 224

 policies regarding, 141
 replacement vs. coexistence, 223–224
 reversion capability during, 220
 ripple, 218
 system accessibility during, 220
 warning customers, 62
Usability, 183
 financial importance of, 183–185
 market issues and, 185
 metaphors for, 185–186
Usage-based fees, 109–110
User community, 167
 nurturing, 167–168
User feedback
 importance of, 187–188
 performance and, 197–198
User fields, 160–161
User interface design, 183–184
 case study of, 190
 tarchitecture and, 187–193
 user models, 188
User interface layer, 147–149

V

Validation code, protecting, 272
Value setting, 230
 procedures for, 231–232
Variation
 defined, 248
 identification of, 255–256
Version
 defined, 248
 number, 174
Versioning, and release management, 260–261
Viewers, for logs, 243
Views, 159–160
Voice, brand identification by, 177
Volume capabilities, checking, 92
Volume licensing, 73–74, 207

W

Warning label, 241
Web-based software deployment, 131

Whole product, 44
 and marketecture, 52
 misuse of term, 45
Windows Event Logs, 242
Workflow support, 188
W3C Extended Log Format, 242

X

XML, in portable applications, 120–121