Beautiful Architecture

Leading Thinkers Reveal the Hidden Beauty in Software Design

O'REILLY®

Edited by Diomidis Spinellis & Georgios Gousios

Software Engineering

Beautiful Architecture

"The authors do a wonderful job in covering some of the fundamentals and best practices of software architecture, and they do so while also covering a wide spectrum of contemporary systems. I particularly enjoyed the range of architectures they touch upon, from Emacs to Facebook, from high ceremony systems to more ethereal ones.

In short, this is a very timely and useful contribution to the art and the science and the practice of software architecture."

—Grady Booch, Fellow, IBM

What are the ingredients of robust, elegant, flexible, and maintainable software architecture? *Beautiful Architecture* answers this question through a collection of remarkable essays from more than a dozen of today's leading software designers and architects. In each essay, contributors present a notable software architecture and analyze what makes it innovative and ideal for its purpose.

With this book, you'll discover:

- How Facebook's architecture is the basis for a data-centric application ecosystem
- The effect of Xen's innovative architecture on the future of operating systems
- How community processes within the KDE project help software architectures evolve from rough sketches to beautiful systems
- How creeping featurism has helped GNU Emacs gain unanticipated functionality
- The magic behind the Jikes RVM self-optimizable, self-hosting runtime

This book includes contributions from:

John Klein and David Weiss
Pete Goodliffe
Jim Waldo
Michael Nygard
Brian Sletten
Dave Fetterman
Derek Murray and Keir Fraser
Greg Lehey
Rhys Newman and Christopher Dennis
Ian Rogers and Dave Grove
Jim Blandy
Till Adam and Mirko Boehm
Bertrand Meyer
Panagiotis Louridas

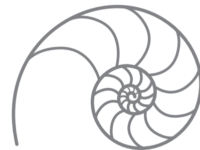
All author royalties will be donated to Doctors Without Borders.

US $44.99 CAN $44.99
ISBN: 978-0-596-51798-4

Safari Books Online
Free online edition for 45 days with purchase of this book. Details on last page.

O'REILLY®
www.oreilly.com

Beautiful Architecture



Edited by Diomidis Spinellis and Georgios Gousios

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Beautiful Architecture
Edited by Diomidis Spinellis and Georgios Gousios

Copyright © 2009 O'Reilly Media. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary Treseler
Production Editor: Sarah Schneider
Copyeditor: Genevieve d'Entremont
Proofreader: Nancy Reinhardt
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
January 2009: First Edition.

O'Reilly and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Beautiful Architecture* and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-51798-4

[M]

1231531335

All royalties from this book will be donated to Doctors Without Borders.

CONTENTS

FOREWORD ix
by Stephen J. Mellor

PREFACE xiii

Part One ON ARCHITECTURE

1 WHAT IS ARCHITECTURE? 3
by John Klein and David Weiss

Introduction 3
Creating a Software Architecture 10
Architectural Structures 14
Good Architectures 19
Beautiful Architectures 20
Acknowledgments 23
References 23

2 A TALE OF TWO SYSTEMS: A MODERN-DAY SOFTWARE FABLE 25
by Pete Goodliffe

The Messy Metropolis 26
Design Town 33
So What? 41
Your Turn 41
References 42

Part Two ENTERPRISE APPLICATION ARCHITECTURE

3 ARCHITECTING FOR SCALE 45
by Jim Waldo

Introduction 45
Context 47
The Architecture 51
Thoughts on the Architecture 57

4 MAKING MEMORIES 63
by Michael Nygard

Capabilities and Constraints 64
Workflow 65
Architecture Facets 66
User Response 87

Conclusion 88
References 88

5 RESOURCE-ORIENTED ARCHITECTURES: BEING "IN THE WEB" 89
by Brian Sletten

Introduction 89
Conventional Web Services 90
The Web 92
Resource-Oriented Architectures 98
Data-Driven Applications 102
Applied Resource-Oriented Architecture 103
Conclusion 109

6 DATA GROWS UP: THE ARCHITECTURE OF THE FACEBOOK PLATFORM 111
by Dave Fetterman

Introduction 111
Creating a Social Web Service 117
Creating a Social Data Query Service 124
Creating a Social Web Portal: FBML 133
Supporting Functionality for the System 146
Summation 151

Part Three SYSTEMS ARCHITECTURE

7 XEN AND THE BEAUTY OF VIRTUALIZATION 155
by Derek Murray and Keir Fraser

Introduction 155
Xenoservers 156
The Challenges of Virtualization 159
Paravirtualization 159
The Changing Shape of Xen 163
Changing Hardware, Changing Xen 169
Lessons Learned 172
Further Reading 173

8 GUARDIAN: A FAULT-TOLERANT OPERATING SYSTEM ENVIRONMENT 175
by Greg Lehey

Tandem/16: Some Day All Computers Will Be Built Like This 176
Hardware 176
Mechanical Layout 178
Processor Architecture 179
The Interprocessor Bus 184
Input/Output 184
Process Structure 185
Message System 186
File System 190
Folklore 195
The Downside 195

Posterity 197
Further Reading 198

9 JPC: AN X86 PC EMULATOR IN PURE JAVA 199
by Rhys Newman and Christopher Dennis

Introduction 200
Proof of Concept 202
The PC Architecture 205
Java Performance Tips 206
Four in Four: It Just Won't Go 207
The Perils of Protected Mode 210
Fighting A Losing Battle 214
Hijacking the JVM 217
Ultimate Flexibility 229
Ultimate Security 231
It Feels Better the Second Time Around 232

10 THE STRENGTH OF METACIRCULAR VIRTUAL MACHINES: JIKES RVM 235
by Ian Rogers and Dave Grove

Background 236
Myths Surrounding Runtime Environments 237
A Brief History of Jikes RVM 240
Bootstrapping a Self-Hosting Runtime 241
Runtime Components 246
Lessons Learned 259
References 259

Part Four END-USER APPLICATION ARCHITECTURES

11 GNU EMACS: CREEPING FEATURISM IS A STRENGTH 263
by Jim Blandy

Emacs in Use 264
Emacs's Architecture 266
Creeping Featurism 272
Two Other Architectures 275

12 WHEN THE BAZAAR SETS OUT TO BUILD CATHEDRALS 279
by Till Adam and Mirko Boehm

Introduction 279
History and Structure of the KDE Project 282
Akonadi 287
ThreadWeaver 303

Part Five LANGUAGES AND ARCHITECTURE

13 SOFTWARE ARCHITECTURE: OBJECT-ORIENTED VERSUS FUNCTIONAL 315
by Bertrand Meyer

Overview 315

The Functional Examples 318
Assessing the Modularity of Functional Solutions 321
An Object-Oriented View 330
Assessing and Improving OO Modularity 336
Agents: Wrapping Operations into Objects 341
Acknowledgments 345
References 346

14 REREADING THE CLASSICS 349
by Panagiotis Louridas

Everything Is an Object 353
Types Are Defined Implicitly 361
Problems 367
Brick and Mortar Architecture 372
References 380

AFTERWORD 383
by William J. Mitchell

CONTRIBUTORS 387

INDEX 393

Foreword

Stephen J. Mellor

The challenges of developing high-performance, high-reliability, and high-quality software systems are too much for ad hoc and informal engineering techniques that might have worked in the past on less demanding systems. The complexity of our systems has risen to the point where we can no longer cope without developing and maintaining a single overarching architecture that ties the system into a coherent whole and avoids piecemeal implementation, which causes testing and integration failures.

But building an architecture is a complex task. Examples are hard to come by, due to either proprietary concerns or the opposite, a need to "sell" a particular architectural style into a wide range of environments, some of which are inappropriate. And architectures are big, which makes them difficult to capture and describe without overwhelming the reader.

Yet beautiful architectures exhibit a few universal principles, some of which I outline here:

One fact in one place

Duplication leads to error, so it should be avoided. Each fact must be a single, nondecomposable unit, and each fact must be independent of all other facts. When change occurs, as it inevitably does, only one place need be modified. This principle is well known to database designers, and it has been formalized under the name of *normalization*. The principle also applies less formally to behavior, under the name *factoring*, such that common functionality is factored out into separate modules.

Beautiful architectures find ways to localize information and behavior. At runtime, this manifests as *layering*, the notion that a system may be factored into layers, each representing a *layer of abstraction* or *domain*.

Automatic propagation

One fact in one place sounds good, but for efficiency's sake, some data or behavior is often duplicated. To maintain consistency and correctness, propagation of these facts must be carried out automatically at construction time.

Beautiful architectures are supported by construction tools that effect *meta-programming*, propagating one fact in one place into many places where they may be used efficiently.

Architecture includes construction

An architecture must include not only the runtime system, but also how it is constructed. A focus solely on the runtime code is a recipe for deterioration of the architecture over time.

Beautiful architectures are *reflective*. Not only are they beautiful at runtime, but they are also beautiful at construction time, using the same data, functions, and techniques to build the system as those that are used at runtime.

Minimize mechanisms

The best way to implement a given function varies case by case, but a beautiful architecture will not strive for "the best." There are, for example, many ways of storing data and searching it, but if the system can meet its performance requirements using one mechanism, there is less code to write, verify, maintain, and occupy memory.

Beautiful architectures employ a minimal set of mechanisms that satisfy the requirements of the whole. Finding "the best" in each case leads to proliferation of error-prone mechanisms, whereas adding mechanisms parsimoniously leads to smaller, faster, and more robust systems.

Construct engines

If you wish to build brittle systems, follow Ivar Jacobson's advice and base your architecture on use cases and one function at a time (i.e., use "controller" objects). Extensible systems, on the other hand, rely on the construction of virtual machines—engines that are "programmed" by data provided by higher layers, and that implement multiple application functions at a time.

This principle appears in many guises. "Layering" of virtual machines goes back to Edsger Dijkstra. "Data-driven systems" provide engines that rely on coding invariants in the system, letting the data define the specific functionality in a particular case. These engines are highly reusable—and beautiful.

O(G), the order of growth

Back in the day, we thought about the "order" of algorithms, analyzing the performance of sorting, say, in terms of the time it takes to sort a set of a certain number of elements. Whole books have been written on the subject.

The same applies for architecture. Polling, for example, works well for a small number of elements, but is a response-time disaster as the number of items increases. Organizing everything around interrupts or events works well until they all go off at once. Beautiful architectures consider the direction of likely growth and account for it.

Resist entropy

Beautiful architectures establish a path of least resistance for maintenance that preserves the architecture over time and so slows the effects of the Law of System Entropy, which states that systems become more disorganized over time. Maintainers must internalize the architecture so that changes will be consistent with it and not increase system entropy.

One approach is the Agile concept of the *Metaphor*, which is a simple way to represent what the architecture is "like." Another is extensive documentation and threats of unemployment, though that seldom works for long. Usually, however, it generally means tools, especially for generating the system. A beautiful architecture must remain beautiful.

These principles are highly interrelated. One fact in one place can work only if you have automatic propagation, which in turn is effective when the architecture takes construction into account. Similarly, constructing engines and minimizing mechanisms support one fact in one place. Resisting entropy is a requirement for maintaining an architecture over time, and it relies on the architecture including construction and support for propagation. Moreover, a failure to consider the way in which a system will likely grow will cause the architecture to become unstable, and eventually fail under extreme but predictable circumstances. And combining minimal mechanisms with the notion of constructing engines means that beautiful architectures usually feature a limited set of patterns that enable construction of arbitrary system extensions, a kind of "expansion by pattern."

In short, beautiful architectures do more with less.

As you read this book, ably assembled and introduced by Diomidis Spinellis and Georgios Gousios, you might look for these principles and consider their implications, using the specific examples presented in each chapter. You might also look for violations of these principles and ask whether the architecture is thus ugly or whether some higher principle is involved.

During the development of this Foreword, your authors asked me if I might say a few words about how someone becomes a good architect. I laughed. If we only knew that.... But then I recalled from my own experience that there is a powerful, if nonanalytic, way of becoming a

beautiful architect. That way[*] is never to believe that the last system you built is the only way to build systems, and to seek out many examples of different ways of solving the same type of problem. The example beautiful architectures presented in this book are a step forward in helping you meet that goal.

[*] Or exercise more and eat less.

Preface

The idea for the book you're reading was conceived in 2007 as a successor to the award-winning, best-selling *Beautiful Code*: a collection of essays about innovative and sometimes surprising solutions to programming problems. In *Beautiful Architecture*, the scope and purpose is different, but similarly focused: to get leading software designers and architects to describe a software architecture of their choice, peeling back the layers of their creations to show how they developed software that is functional, reliable, usable, efficient, maintainable, portable, and, yes, elegant.

To put together this book, we contacted leading architects of well-known or less-well-known but highly innovative software projects. Many of them replied promptly and came back to us with thought-provoking ideas. Some of the contributors even caught us by surprise by proposing not to write about a specific system, but instead investigating the depth and the extent of architectural aspects in software engineering.

All chapter authors were glad to hear that the work they put in their chapters is also helping a good cause, as the royalties of this book are donated to *Medécins Sans Frontières* (Doctors Without Borders), an international humanitarian aid organization that provides emergency medical assistance to suffering people.

How This Book Is Organized

We have organized the contents of this book around five thematic areas: overviews, enterprise applications, systems, end-user applications, and programming languages. There is an obvious, but not deliberate, lack of chapters on desktop software architectures. Having approached more than 50 software architects, this result was another surprise for us. Are there really no shining examples of beautiful desktop software architectures? Or are talented architects shying away from an area often driven by a quest to continuously pile ever more features on an application? We are really looking forward to hearing from you on these issues.

Part I: On Architecture

Part I of this book examines the breadth and scope of software architecture and its implications for software development and evolution.

Chapter 1, *What Is Architecture?*, by John Klein and David Weiss, defines software architecture by examining the subject through the perspectives of quality concerns and architectural structures.

Chapter 2, *A Tale of Two Systems: A Modern-Day Software Fable*, by Pete Goodliffe, provides an allegory on how software architectures can affect system evolution and developer engagement with a project.

Part II: Enterprise Application Architecture

Enterprise systems, the IT backbone of many organizations, are large and often tailor-made conglomerates of software usually built from diverse components. They serve large, transactional workloads and must scale along with the enterprise they support, readily adapting to changing business realities. Scalability, correctness, stability, and extensibility are the most important concerns when architecting such systems. Part II of this book includes some exemplar cases of enterprise software architectures.

Chapter 3, *Architecting for Scale*, by Jim Waldo, demonstrates the architectural prowess required to build servers for massive multiplayer online games.

Chapter 4, *Making Memories*, by Michael Nygard, goes through the architecture of a multistage, multisite data processing system and presents the compromises that must be made to make it work.

Chapter 5, *Resource-Oriented Architectures: Being "In the Web"*, by Brian Sletten, discusses the power of resource mapping when constructing data-driven applications and provides an elegant example of a purely resource-oriented architecture.

Chapter 6, *Data Grows Up: The Architecture of the Facebook Platform*, by Dave Fetterman, advocates data-centric systems, explaining how a good architecture can create and support an application ecosystem.

Part III: Systems Architecture

Systems software is arguably the most demanding type of software to design, partly because efficient use of hardware is a black art mastered by a selected few, and partly because many consider systems software as infrastructure that is "simply there." Seldom are great systems architectures designed on a blank sheet; most systems that we use today are based on ideas first conceived in the 1960s. The chapters in Part III walk you through four innovative systems software architectures, discussing the complexities behind the architectural decisions that made them beautiful.

Chapter 7, *Xen and the Beauty of Virtualization*, by Derek Murray and Keir Fraser, gives an example of how a well-thought-out architecture can change the way operating systems evolve.

Chapter 8, *Guardian: A Fault-Tolerant Operating System Environment*, by Greg Lehey, presents a retrospective on the architectural choices and building blocks (both software and hardware) that made Tandem the platform of choice in high-availability environments for nearly two decades.

Chapter 9, *JPC: An x86 PC Emulator in Pure Java*, by Rhys Newman and Christopher Dennis, describes how carefully designed software and a good understanding of domain requirements can overcome the perceived deficiencies of a programming system.

Chapter 10, *The Strength of Metacircular Virtual Machines: Jikes RVM*, by Ian Rogers and Dave Grove, walks us through the architectural choices required for creating a self-optimizable, self-hosting runtime for a high-level language.

Part IV: End-User Application Architectures

End-user applications are those that we interact with in our everyday computing lives, and the software that our CPUs burn the most cycles to execute. This kind of software normally does not need to carefully manage resources or serve large transaction volumes. However, it does need to be usable, secure, customizable, and extensible. These properties can lead to popularity and widespread use and, in the case of free and open source software, to an army of volunteers willing to improve it. In Part IV, the authors dissect the architectures and the community processes required to evolve two very popular desktop software packages.

Chapter 11, *GNU Emacs: Creeping Featurism Is a Strength*, by Jim Blandy, explains how a set of very simple components and an extension language can turn the humble text editor into ~~an operating system~~* the Swiss army knife of a programmer's toolchest.

* As some die-hard users say, "Emacs is my operating system; Linux just provides the device drivers."

Chapter 12, *When the Bazaar Sets Out to Build Cathedrals*, by Till Adam and Mirko Boehm, demonstrates how community processes such as sprints and peer-reviews can help software architectures evolve from rough sketches into beautiful systems.

Part V: Languages and Architecture

As many people have pointed out in their works, the programming language we use affects the way we solve a problem. But can a programming language also affect a system's architecture and, if so, how? In the architecture of buildings, new materials and the adoption of CAD systems allowed the expression of more sophisticated and sometimes strikingly beautiful designs; does the same also apply to computer programs? Part V, which contains the last two chapters, investigates the relationship between the tools we use and the designs we produce.

Chapter 13, *Software Architecture: Object-Oriented Versus Functional*, by Bertrand Meyer, compares the affordances of object-oriented and functional architectural styles.

Chapter 14, *Rereading the Classics*, by Panagiotis Louridas, surveys the architectural choices behind the building blocks of modern and classical object-oriented software languages.

Finally, in the thought-provoking Afterword, William J. Mitchell, an MIT Professor of Architecture and Media Arts and Sciences, ties the concept of beauty between the building architectures we encounter in the real world and the software architectures residing on silicon.

Principles, Properties, and Structures

Late in this book's review process, one of the reviewers asked us to provide our personal opinion, in the form of commentary, on what a reader could learn from each chapter. The idea was intriguing, but we did not like the fact that we would have to second-guess the chapter authors. Asking the authors themselves to provide a meta-analysis of their writings would lead to a Babel tower of definitions, terms, and architectural constructs guaranteed to confuse readers. What was needed was a common vocabulary of architectural terms; thankfully, we realized we already had that in our hands.

In the Foreword, Stephen Mellor discusses seven principles upon which all beautiful architectures are based. In Chapter 1, John Klein and David Weiss present four architecture building blocks and six properties that beautiful architectures exhibit. A careful reader will notice that Mellor's principles and Klein's and Weiss's properties are not independent of each other. In fact, they mostly coincide; this happens because great minds think alike. All three, being very experienced architects, have seen many times in action the importance of the concepts they describe.

We merged Mellor's architectural principles with the definitions of Klein and Weiss into two lists: one containing principles and properties (Table P-1), and one containing structures (Table P-2). We then asked the chapter authors to mark the terms they thought applied to their chapters, and produced a corresponding legend for each chapter. In these tables, you can see the definition of each principle, property, or architectural construct that appears in the chapter legend. We hope the legends will guide your reading of this book by giving you a clean overview of the contents of each chapter, but we urge you to delve into a chapter's text rather than simply stay with the legend.

TABLE P-1. Architectural principles and properties

Principle or property	The ability of an architecture to...
Versatility	...offer "good enough" mechanisms to address a variety of problems with an economy of expression.
Conceptual integrity	...offer a single, optimal, nonredundant way for expressing the solution of a set of similar problems.
Independently changeable	...keep its elements isolated so as to minimize the number of changes required to accommodate changes.
Automatic propagation	...maintain consistency and correctness, by propagating changes in data or behavior across modules.
Buildability	...guide the software's consistent and correct construction.
Growth accommodation	...cater for likely growth.
Entropy resistance	...maintain order by accommodating, constraining, and isolating the effects of changes.

TABLE P-2. Architectural structures

Structure	A structure that...
Module	...hides design or implementation decisions behind a stable interface.
Dependency	...organizes components along the way where one uses functionality of another.
Process	...encapsulates and isolates the runtime state of a module.
Data access	...compartmentalizes data, setting access rights to it.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beautiful Architecture*, edited by Diomidis Spinellis and Georgios Gousios. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51798-4."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596517984

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

The publication of a book is a team effort, and an edited collection even more so. Many people deserve our thanks. First of all, we thank the book's contributors for submitting outstanding material in a timely manner, and then putting up with our requests for various changes and revisions. The book's reviewers, Robert A. Maksimchuk, Gary Pollice, David West, Greg Wilson, and Bobbi Young, gave us many excellent comments for improving each chapter and the book as a whole. At O'Reilly, our editor, Mary Treseler, helped us locate contributors, organized the review process, and oversaw the book's production with remarkable efficiency. Later, Sarah Schneider worked with us as the book's production editor, adroitly handling a pressing schedule and often conflicting requirements. The copyeditor, Genevieve d'Entremont, and the indexer, Fred Brown, deftly massaged material coming from authors around the world to form a book that flows as easily as if it was written by a single pen. The illustrator, Robert Romano, managed to convert the disparate variety of the graphics formats we submitted (including some hand-drawn sketches) into the professional diagrams you'll find in the book. The cover designer, Karen Montgomery, produced a beautiful and inspiring cover to match the book's contents, and the interior designer, David Futato, came up with a creative and functional scheme for integrating the chapter legends into the book's design. Finally, we wish to thank our families and friends for standing by us while we diverted to this book attention that should have belonged to them.

PART I

On Architecture

Chapter 1, *What Is Architecture?*
Chapter 2, *A Tale of Two Systems: A Modern-Day Software Fable*

CHAPTER ONE

What Is Architecture?

John Klein
David Weiss

Introduction

Builders, musicians, writers, computer designers, network designers, and software developers all use the term architecture, as do others (ever hear of a food architect?), yet each produces different results. A building is very different from a symphony, but both have architectures. Further, all architects talk about beauty in their work and its results. A building architect might say that a building should provide an environment suitable for working or living, and that it should be beautiful to behold; a musician that the music should be playable, with a discernible theme, and that it should be beautiful to the ear; a software architect that the system should be friendly and responsive to the user, maintainable, free of critical errors, easy to install, reliable, that it should communicate in standard ways with other systems, and that it, too, should be beautiful.

This book provides you with detailed examples of beautiful architectures drawn from the fields of computerized systems, a relatively young discipline. Because we are young, we have fewer examples to emulate than fields such as building, music, or writing, and therefore we need them even more. This book intends to help fill that need.

Before you proceed to the examples, we would like you to consider what an architecture is and what the attributes of a beautiful architecture might be. As you will see from the different definitions of architecture in this chapter, each discipline has its own definition, so we will first explore what is common among architectures in different disciplines and what problems one tries to solve with an architecture. Particularly, an architecture can help assure that the system satisfies the concerns of its stakeholders, and it can help deal with the complexity of conceiving, planning, building, and maintaining the system.

We then proceed to a definition of architecture and show how we can apply that definition to software architecture, since software is central to many of the later examples. Key to the definition is that an architecture consists of a set of structures designed to let the architects, builders, and other stakeholders see how their concerns are satisfied.

We end this chapter with a discussion of the attributes of beautiful architectures and cite a few examples. Central to beauty is conceptual integrity—that is, a set of abstractions and the rules for using them throughout the system as simply as possible.

In our discussion we will use "architecture" as a noun to denote a set of artifacts, including documentation such as blueprints and building specifications that describe the object to be built, wherein the object is viewed as a set of structures. The term is also used by some as a verb to describe the process of creating the artifacts, including the resulting work. As Jim Waldo and others have pointed out, however, there is no process that you can learn that guarantees you will produce a good system architecture, let alone a beautiful one (Waldo 2006), so we will focus more on artifacts than process.

Architecture: "The art or science of building; esp. the art or practice of designing and building edifices for human use, taking both aesthetic and practical factors into account."

—The Shorter Oxford English Dictionary, *Fifth Edition*, 2002

In all disciplines, architecture provides a means for solving a common problem: assuring that a building, or bridge, or composition, or book, or computer, or network, or system has certain properties and behaviors when it has been built. Put another way, the architecture is both a plan for the system so that the result can have the desired properties and a description of the built system. Wikipedia says: "According to the earliest surviving work on the subject, Vitruvius' 'On Architecture,' good building should have Beauty (Venustas), Firmness (Firmitas), and Utility (Utilitas); architecture can be said to be a balance and coordination among these three elements, with no one overpowering the others."

We speak of the "architecture" of a symphony, and call architecture, in its turn, "frozen music."

—*Deryck Cooke*, The Language of Music

A good system architecture exhibits conceptual integrity; that is, it comes equipped with a set of design rules that aid in reducing complexity and that can be used as guidance in detailed design and in system verification. Design rules may incorporate certain abstractions that are always used in the same way, such as virtual devices. The rules may be represented as a pattern, such as pipes and filters. In the best case there are verifiable rules, such as "any virtual device of the same type may replace any other virtual device of the same type in the event of device failure," or "all processes contending for the same resource must have the same scheduling priority."

A contemporary architect might say that the object or system under construction must have the following characteristics.

- It has the functionality required by the customer.
- It is safely buildable on the required schedule.
- It performs adequately.
- It is reliable.
- It is usable and safe to use.
- It is secure.
- It is affordable.
- It conforms to legal standards.
- It will outlast its predecessors and its competitors.

> **The architecture of a computer system we define as *the minimal set of properties that determine what programs will run and what results they will produce*.**
>
> —*Gerrit Blaauw & Frederick Brooks*, Computer Architecture

We've never seen a complex system that perfectly satisfies all of the preceding characteristics. Architecture is a game of trade-offs—a decision that improves one of these characteristics often diminishes another. The architect must determine what is sufficient to satisfy, by discovering the important concerns for a particular system and the conditions for satisfying them sufficiently.

Common among the notions of architecture is the idea of structures, each defined by components of various sorts and their relations: how they fit together, invoke each other, communicate, synchronize, and otherwise interact. Components could be support beams or internal rooms in a building, individual instruments or melodies in a symphony, book chapters or characters in a story, CPUs and memory chips in a computer, layers in a communications stack or processors connected to a network, cooperating sequential processes, objects, collections of compile-time macros, or build-time scripts. Each discipline has its own sets of components and its own relationships among them.

In wider use, the term "architecture" always means "unchanging deep structure."

—*Stewart Brand*, How Buildings Learn

In the face of increasing complexity of systems and their interactions, both internally and with each other, an architecture comprising a set of structures provides the primary means for dealing with complexity in order to ensure that the resulting system has the required properties. Structures provide ways to understand the system as sets of interacting components.

Each structure is intended to help the architect understand how to satisfy particular concerns, such as changeability or performance. The job of demonstrating that particular concerns are satisfied may fall to others, but the architect must be able to demonstrate that *all* concerns have been met.

Network architecture: the communication equipment, protocols, and transmission links that constitute a network, and the methods by which they are arranged.

—*http://www.wtcs.org/snmp4tpc/jton.htm*

The Role of Architect

When buildings are designed, constructed, or renovated, we designate key designers as "architects" and give them a broad range of responsibilities. An architect prepares initial sketches of the building, showing both external appearance and internal layout, and discusses these sketches with clients until all concerned have agreed that what is shown is what they want. The sketches are abstractions: they focus attention on the pertinent details of a particular aspect of the building, omitting other concerns.

After the clients and architects agree on these abstractions, the architects prepare, or supervise the preparation of, much more detailed drawings, as well as associated textual specifications. These drawings and specifications describe many "nitty-gritty" details of a building, such as plumbing, siding materials, window glazing, and electrical wiring.

On rare occasions, an architect simply hands the detailed plans to a builder who completes the project in accordance with the plans. For more important projects, the architect remains involved, regularly inspects the work, and may propose changes or accept suggestions for change from both the builder and customer. When the architect supervises the project, it is not considered complete until he certifies that it is in substantial compliance with the plans and specifications.

We employ an architect to assure that the design (1) meets the needs of the client, including the characteristics previously noted; (2) has conceptual integrity by using the same design rules throughout; and (3) meets legal and safety requirements. An important part of the architect's role is to ensure that the design concepts are consistently realized during the implementation.

Sometimes the architect also acts as a mediator between builder and client. There is often some disagreement about which decisions are in the realm of the architect and which are left to others, but it is always clear that the architect makes the major decisions, including all that can affect the usability, safety, and maintainability of the structure.

MUSIC COMPOSITION AND SOFTWARE ARCHITECTURE

Whereas building architecture is often used as an analogy for software architecture, music composition may be a better analogy. A building architect creates a static description (blueprints and other drawings) of a relatively static structure (the architecture must account for movement of people and services within the building as well as the load-bearing structure). In music composition and software design, the composer (software architect) creates a static description of a piece of music (architecture description and code) that is later performed (executed) many times. In both music and software the design can account for many components interacting to produce the desired result, and the result varies depending on the performers, the environment in which it is performed, and the interpretation imposed by the performers.

The Role of the Software Architect

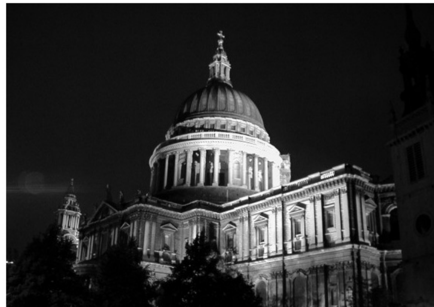
Software development projects need people who play the same role for software construction that traditional architects play when buildings are constructed or renovated. For software systems, however, it has never been clear exactly which decisions are the purview of the architect and which can be left to the implementers. The definition of what an architect does in a software project is more difficult than the analogous definition for building architects because of three factors: lack of tradition, the intangible nature of the product, and the complexity of the system. (See Grinter [1999] for a portrayal of how a software architect carries out her role within a large software development organization.)

In particular:

- Building architects can look back at thousands of years of history to see what architects have done in the past; they can visit and study buildings that have been standing for hundreds, and sometimes a thousand years or more, and that are still in use. In software we have only a few decades of history and our designs are often not public. Furthermore, building architects have and use standards for describing the drawings and specifications that the architects produce, allowing present architects to take advantage of the recorded history of architecture.
- Buildings are physical products; there is a clear distinction between the plans produced by the architects and the building produced by the workers.

ARCHITECTURAL REUSE

The Hagia Sophia (top), built in Istanbul in the sixth century, pioneered the use of structures called pendentives to support its enormous dome, and is an example of beauty in Byzantine architecture. Christopher Wren, 1,100 years later, used the same design for the dome of St. Paul's cathedral (bottom), a London landmark. Both still stand and are used today.



On major software projects, there are often many architects. Some architects are quite specialized in disciplines, such as databases and networks, and usually work as part of a team, but for now we will write as if there were only one.

What Constitutes a Software Architecture?

It is a mistake to think of "an architecture" as if it were a simple entity that could be described by a single document or drawing. Architects must make many design decisions. To be useful, these decisions must be documented so that they can be reviewed, discussed, modified, and

approved, and then serve to constrain subsequent decision making and construction. For software systems, these design decisions are behavioral and structural.

External behavioral descriptions show how the product will interface with its users, other systems, and external devices, and should take the form of requirements. Structural descriptions show how the product is divided into parts and the relations between those parts. Internal behavioral descriptions are needed to describe the interfaces between components. Structural descriptions often show several distinct views of the same part because it is impossible to put all the information in one drawing or document in a meaningful way. A component in one view may be a part of a component in another.

Software architectures are often presented as layered hierarchies that tend to commingle several different structures in one diagram. In the 1970s Parnas pointed out that the term "hierarchy" had become a buzzword, and then precisely defined the term and gave several different examples of structures used for different purposes in the design of different systems (Parnas 1974). Describing the structures of an architecture as a set of *views*, each of which addresses different concerns, is now accepted as a standard architecture practice (Clements et al. 2003; IEEE 2000). We will use the word "architecture" to refer to a set of annotated diagrams and functional descriptions that specify the structures used to design and construct a system. In the software development community there are many different forms used, and proposed, for such diagrams and descriptions. See Hoffman and Weiss (2000, chaps. 14 and 16) for some examples.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

"Externally visible" properties are those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

—Len Bass, Paul Clements, and Rick Kazman, Software Architecture in Practice, *Second Edition*

Architecture Versus Design

Architecture is a part of the design of the system; it highlights some details by abstracting away from others. Architecture is thus a subset of design. A developer focused on implementing a component of the system may not be very aware of how all the components fit together, but rather is primarily concerned with the design and development of a small number of component(s), including the architectural constraints that they must obey and the rules they can use. As such, the developer is working on a different aspect of the system design than the architect.

If architecture is concerned with the relationships among components and the externally visible properties of system components, then design will additionally be concerned with the internal structure of those components. For example, if one set of components consists of information-hiding modules, then the externally visible properties form the interfaces to those components, and the internal structure is concerned with the data structures and flow of control within a module (Hoffman and Weiss 2000, chaps. 7 and 16).

Creating a Software Architecture

So far, we have considered architecture in general and looked at how software architecture is both similar to and different from architecture in other domains. We now turn our attention to the "how" of software architecture. Where should the architect focus her attention when she is creating the architecture for a software system?

The first concern of a software architect is not the functionality of the system.

That's right—the first concern of a software architect is not the functionality of the system.

For example, if we offer to hire you to develop the architecture for a "web-based application," would you start by asking us about page layouts and navigation trees, or would you ask us questions such as:

- Who will host it? Are there technology restrictions in the hosting environment?
- Do you want to run on a Windows Server or on a LAMP stack?
- How many simultaneous users do you want to support?
- How secure does the application need to be? Is there data that we need to protect? Will the application be used on the public Internet or a private intranet?
- Can you prioritize these answers for me? For example, is number of users more important than response time?

Depending on our answers to these and a few other questions, you can begin sketching out an architecture for the system. And we still haven't talked about the functionality of the application.

Now, admittedly, we cheated a bit here because we asked for a "web-based application," which is a well-understood domain, so you already knew what decisions would have the most influence on your architecture. Similarly, if we had asked for a telecommunications system or an avionics system, an architect experienced in one of those domains would have some notion of required functionality in mind. But still, you were able to begin creating the architecture without worrying too much about the functionality. You did this by focusing on *quality concerns* that needed to be satisfied.

Quality concerns specify the way in which the functionality must be delivered in order to be acceptable to the system's stakeholders, the people with a vested interest in the outcome of

the system. Stakeholders have certain concerns that the architect must address. Later, we will discuss concerns that are typically raised when trying to assure that the system has the required qualities. As we said earlier, one role of the architect is to ensure that the design of the system will meet the needs of the client, and we use quality concerns to help us understand those needs.

This example highlights two key practices of successful architects: stakeholder involvement and a focus on *both* quality concerns and functionality. As the architect, you began by asking us what we wanted from the system, and in what priority. In a real project, you would have sought out other stakeholders. Typical stakeholders and their concerns include:

- Funders, who want to know if the project can be completed within resource and schedule constraints
- Architects, developers, and testers, who are first concerned with initial construction and later with maintenance and evolution
- Project managers, who need to organize teams and plan iterations
- Marketers, who may want to use quality concerns to differentiate the system from competitors
- Users, including end users, system administrators, and the people who do installation, deployment, provisioning, and configuration
- Technical support staff, who are concerned with the number and complexity of Help Desk calls

Every system has its own set of quality concerns. Some, such as performance, security, and scalability, may be well-specified, but other, often equally important concerns, such as changeability, maintainability, and usability, may not be defined with enough detail to be useful. Odd, isn't it, that stakeholders want to put functions in software and not hardware so that they can be easily and quickly modified, and then often give short shrift to changeability when stating their quality concerns? Architecture decisions will have an impact on what kinds of changes can be done easily and quickly and what changes will take time and be hard to do. So shouldn't an architect understand his stakeholders' expectations for qualities such as "changeability" as well as he understands the functional requirements?

Once the architect understands the stakeholders' quality concerns, what does she do next? Consider the trade-offs. For example, encrypting messages improves security but hurts performance. Using configuration files may increase changeability but could decrease usability unless we can verify that the configuration is valid. Should we use a standard representation for these files, such as XML, or invent our own? Creating the architecture for a system involves making many such difficult trade-offs.

The first task of the architect, then, is to work with stakeholders to understand and prioritize quality concerns and constraints. Why not start with functional requirements? Because there are usually many possible system decompositions. For example, starting with a data model

would lead to one architecture, whereas starting with a business process model might lead to a different architecture. In the extreme case, there is no decomposition, and the system is developed as a monolithic block of software. This might satisfy all functional requirements, but it probably will not satisfy quality concerns such as changeability, maintainability, or scalability. Architects often must do architecture-level refactoring of a system, for example to move from simplex to distributed deployment, or from single-threaded to multithreaded in order to meet scalability or performance requirements, or hardcoded parameters to external configuration files because parameters that were *never* going to change now need to be modified.

Although there are many architectures that can meet functional requirements, only a subset of these will also satisfy quality requirements. Let's go back to the web application example. Think of the many ways to serve up web pages—Apache with static pages, CGI, servlets, JSP, JSF, PHP, Ruby on Rails, or ASP.NET, to name just a few. Choosing one of these technologies is an architecture decision that will have significant impact on your ability to meet certain quality requirements. For example, an approach such as Ruby on Rails might provide the fast time-to-market benefit, but could be harder to maintain as both the Ruby language and the Rails framework continue to evolve rapidly. Or perhaps our application is a web-based telephone and we need to make the phone "ring." If you need to send true asynchronous events from the server to the web page to satisfy performance requirements, an architecture based on servlets might be more testable and modifiable.

In real-world projects, satisfying stakeholder concerns requires many more decisions than simply selecting a web framework. Do you really need an "architecture," and do you need an "architect" to make the decisions? Who should make them? Is it the coder, who may make many of them unintentionally and implicitly, or is it the architect, who makes them explicitly with a view in mind of the entire system, its stakeholders, and its evolution? Either way, you will have an architecture. Should it be explicitly developed and documented, or should it be implicit and require reading of the code to discover?

Often, of course, the choice is not so stark. As the size of the system, its complexity, and the number of people who work on it increase, however, those early decisions and the way that they are documented will have greater and greater impact.

We hope you understand by now that architecture decisions are important if your system is going to meet its quality requirements, and that you want to pay attention to the architecture and make these decisions intentionally rather than just "letting the architecture emerge."

What happens when the system is very large? One of the reasons that we apply architecture principles such as "divide and conquer" is to reduce complexity and enable work to proceed in parallel. This allows us to create larger and larger systems. Can the architecture itself be decomposed into parts, and those parts worked on by different people in parallel? In considering computer architecture, Gerrit Blaauw and Fred Brooks asserted:

> ...if, after all techniques to make the task manageable by a single mind have been applied, the architectural task is still so large and complex that it cannot be done in that way, the product

> conceived is too complex to be usable and should not be built. In other words, the mind of a single user must comprehend a computer architecture. If a planned architecture cannot be designed by a single mind, it cannot be comprehended by one. (1997)

Do you need to understand all aspects of an architecture in order to use it? An architecture separates concerns so, for the most part, the developer or tester using the architecture to build or maintain a system does not need to deal with the entire architecture at once, but can interact with only the necessary parts to perform a given function. This allows us to create systems larger than a single mind can comprehend. But, before we completely ignore the advice of the people who built the IBM System/360, one of the longest-lived computer architectures, let's look at what prompted them to make this statement.

Fred Brooks said that conceptual integrity is the most important attribute of an architecture: "It is better to have a system...reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas" (1995). It is this conceptual integrity that allows a developer who already knows about one part of a system to quickly understand another part. Conceptual integrity comes from consistency in things such as decomposition criteria, application of design patterns, and data formats. This allows a developer to apply experience gained working in one part of the system to developing and maintaining other parts of the system. The same rules apply throughout the system. As we move from system to "system-of-systems," the conceptual integrity must also be maintained in the architecture that integrates the systems, for example by selecting an architecture style such as *publish/subscribe message bus* and then applying this style uniformly to all system integrations in the system-of-systems.

The challenge for an architecture team is to maintain a single-mindedness and a single philosophy as they go about creating the architecture. Keep the team as small as possible, work in a highly collaborative environment with frequent communication, and have one or two "chiefs" act as benevolent dictators with the final say on all decisions. This organizational pattern is commonly seen in successful systems, whether corporate or open source, and results in the conceptual integrity that is one of the attributes of a beautiful architecture.

Good architects are often formed by having better architects mentor them (Waldo 2006). One reason may be that there are certain concerns that are common to nearly all projects. We have already alluded to some of them, but here is a more complete list, with each concern phrased as a question that the architect may need to consider during the course of a project. Of course, individual systems will have additional critical concerns.

Functionality
: What functionality does the product offer to its users?

Changeability
: What changes may be needed in the software in the future, and what changes are unlikely and need not be especially easy to make in the future?

Performance
: What will the performance of the product be?

Capacity
: How many users will use the system simultaneously? How much data will the system need to store for its users?

Ecosystem
: What interactions will the system have with other systems in the ecosystem in which it will be deployed?

Modularity
: How is the task of writing the software organized into work assignments (modules), particularly modules that can be developed independently and that suit each other's needs precisely and easily?

Buildability
: How can the software be built as a set of components that can be independently implemented and verified? What components should be reused from other products and which should be acquired from external suppliers?

Producibility
: If the product will exist in several variations, how can it be developed as a product line, taking advantage of the commonality among the versions, and what are the steps by which the products in the product line can be developed (Weiss and Lai 1999)? What investment should be made in creating a software product line? What is the expected return from creating the options to develop different members of the product line?

: In particular, is it possible to develop the smallest minimally useful product first and then develop additional members of the product line by adding (and subtracting) components without having to change the code that was written previously?

Security
: If the product requires authorization for its use or must restrict access to data, how can security of data be ensured? How can "denial of service" and other attacks be withstood?

Finally, a good architect realizes that the architecture affects the organization. Conway noted that the structure of a system reflects the structure of the organization that built it (1968). The architect may realize that Conway's Law can be used in reverse. In other words, a good architecture may influence an organization to change so as to be more efficient in building systems derived from the architecture.

Architectural Structures

How, then, does a good architect deal with these concerns? We have already mentioned the need to organize the system into structures, each defining specific relationships among certain types of components. The architect's chief focus is to organize the system so that each structure

helps answer the defining questions for one of the concerns. Key structural decisions divide the product into components and define the relationships among those components (Bass, Clements, and Kazman 2003; Booch, Rumbaugh, and Jacobson 1999; IEEE 2000; Garlan and Perry 1995). For any given product, there are many structures that need to be designed. Each must be designed separately so that it is viewed as a separate concern. In the next few sections we discuss some structures that you can use to address the concerns on our list. For example, the Information Hiding Structures show how the system is organized into work assignments. They can also be used as a roadmap for change, showing for proposed changes which modules accommodate those changes. For each structure we describe the components and the relations among them that define the structure. Given the concerns on our list, we consider the following structures to be of primary importance.

The Information Hiding Structures

COMPONENTS AND RELATIONS: The primary components are Information Hiding Modules, where each module is a work assignment for a group of developers, and each module embodies a design decision. We say that a design decision is the secret of a module if the decision can be changed without affecting any other module (Hoffman and Weiss 2000, chaps. 7 and 16). The most basic relation between the modules is "part of." Information Hiding Module A is part of Information Hiding Module B if A's secret is a part of B's secret. Note that it must be possible to change A's secret without changing any other part of B; otherwise, A is not a submodule according to our definition. For example, many architectures have virtual device modules, whose secret is how to communicate with certain physical devices. If virtual devices are organized into types, then each type might form a submodule of the virtual device module, where the secret of each virtual device type would be how to communicate with devices of that type.

Each module is a work assignment that includes a set of programs to be written. Depending on language, platform, and environment, a "program" could be a method, a procedure, a function, a subroutine, a script, a macro, or other sequence of instructions that can be made to execute on a computer. A second Information Hiding Module Structure is based on the relation "contained in" between programs and modules. A program P is contained in a module M if part of the work assignment M is to write P. Note that every program is contained in a module because every program must be part of some developer's work assignment.

Some of these programs are accessible on the module's interface, whereas others are internal. Modules may also be related through interfaces. A module's interface is a set of assumptions that programs outside of the module may make about the module and the set of assumptions that the module's programs make about programs and data structures of other modules. A is said to "depend on" B's interface if a change to B's interface might require a change in A.

The "part of" structure is a hierarchy. At the leaf nodes of the hierarchy are modules that contain no identified submodules. The "contained in" structure is also a hierarchy, since each

program is contained in only one module. The "depends on" relation does not necessarily define a hierarchy, as two modules may depend on each other either directly or through a longer loop in the "depends on" relation. Note that "depends on" should not be confused with "uses" as defined in a later section.

Information Hiding Structures are the foundation of the object-oriented design paradigm. If an Information Hiding Module is implemented as a class, the public methods of the class belong to the interface for the module.

CONCERNS SATISFIED: The Information Hiding Structures should be designed so that they satisfy changeability, modularity, and buildability.

The Uses Structures

COMPONENTS AND RELATION: As defined previously, Information Hiding Modules contain one or more programs (as defined in the previous section). Two programs are included in the same module if and only if they share a secret. The components of the Uses Structure are programs that may be independently invoked. Note that programs may be invoked by each other or by the hardware (for example, by an interrupt routine), and the invocation may come from a program in a different namespace, such as an operating system routine or a remote procedure. Furthermore, the time at which an invocation may occur could be any time from compile time through runtime.

We will consider forming a Uses Structure only among programs that operate at the same binding time. It is probably easiest first just to think about programs that operate at runtime. Later, we may also think about the uses relation among programs that operate at compile time or load time.

We say that program A uses program B if B must be present and satisfy its specification for A to satisfy its specification. In other words, B must be present and operate correctly for A to operate correctly. The Uses Relation is sometimes known as "requires the presence of a correct version of." For a further explanation and example, see Chapter 14 of Hoffman and Weiss (2000).

The Uses Structure determines what working subsets can be built and tested. A desirable property in the Uses Relation for a software system is that it defines a hierarchy, meaning that there are no loops in it. When there is a loop in the Uses Relation, all programs in the loop must be present and working in the system for any of them to work. Since it may not be possible to construct a completely loop-free Uses Relation, an architect may treat all of the programs in a Uses loop as a single program for the purpose of creating subsets. A subset must include either the whole program or none of it.

When there are no loops in the Uses Relation, a levels structure is imposed on the software. At the bottom level, level 0, are all programs that use no other programs. Level n consists of all programs that use programs in level n–1 or below. The levels are often depicted as a series

of layers, with each layer representing one or several levels in the Uses Relation. Grouping adjacent levels in Uses helps to simplify the representation and allows for cases where there are small loops in the relation. One guideline in performing such a grouping is that programs at one layer should execute approximately 10 times as quickly and 10 times as often as programs in the next layer above it (Courtois 1977).

A system that has a hierarchical Uses Structure can be built one or a few layers at a time. These layers are sometimes known as "levels of abstraction," but this is a misnomer. Because the components are individual programs, not whole modules, they do not necessarily abstract from (hide) anything.

Often a large software system has too many programs to make the description of the Uses Relation among programs easily understandable. In such cases, the Uses Relation may be formed on aggregations of programs, such as modules, classes, or packages. Such aggregated descriptions lose important information but help to present the "big picture." For example, one can sometimes form a Uses Relation on Information Hiding Modules, but unless all programs in a module are on the same level of the programmatic Uses hierarchy, important information is lost.

In some projects, the Uses Relation for a system is not fully determined until the system is implemented, because the developers determine what programs they will use as the implementation proceeds. The architects of the system may, however, create an "Allowed-to-Use" Relation at design time that constrains the developers' choices. Henceforth, we will not distinguish between "Uses" and "Allowed-to-Use."

A well-defined Uses Structure will create proper subsets of the system and can be used to drive iterative or incremental development cycles.

CONCERNS SATISFIED: Producibility and ecosystem.

The Process Structures

COMPONENTS AND RELATION: The Information Hiding Module Structures and the Uses Structures are static structures that exist at design and code time. We now turn to a runtime structure. The components that participate in the Process Structure are Processes. Processes are runtime sequences of events that are controlled by programs (Dijkstra 1968). Each program executes as part of one or many Processes. The sequence of events in one Process proceed independently of the sequence of events in another Process, except where the Processes synchronize with each other, such as when one Process waits for a signal or a message from the other. Processes are allocated resources, including memory and processor time, by support systems. A system may contain a fixed number of Processes, or it may create and destroy Processes while running. Note that *threads* implemented in operating systems such as Linux and Windows fall under this definition of Processes. Processes are the components of several distinct relations. Some examples follow.

Process gives work to

One Process may create work that must be completed by other Processes. This structure is essential in determining whether a system can get into a deadlock.

CONCERNS SATISFIED: Performance and capacity.

Process gets resources from

In systems with dynamic resource allocation, one Process may control the resources used by another, where the second must request and return those resources. Because a requesting Process may request resources from several controllers, each resource may have a distinct controlling Process.

CONCERNS SATISFIED: Performance and capacity.

Process shares resources with

Two Processes may share resources such as printers, memory, or ports. If two Processes share a resource, synchronization is necessary to prevent usage conflicts. There may be distinct relations for each resource.

CONCERNS SATISFIED: Performance and capacity.

Process contained in module

Every Process is controlled by a program and, as noted earlier, every program is contained in a module. Consequently, we can consider each Process to be contained in a module.

CONCERNS SATISFIED: Changeability.

Access Structures

The data in a system may be divided into segments with the property so that if a program has access to any data in a segment, it has access to all data in that segment. Note that to simplify the description, the decomposition should use maximally sized segments by adding the condition that if two segments are accessed by the same set of programs, those two segments should be combined. The data access structure has two kinds of components, programs and segments. This relation is entitled "has access to," and is a relation between programs and segments. A system is thought to be more secure if this structure minimizes the access rights of programs and is tightly enforced.

CONCERNS SATISFIED: Security.

Summary of Structures

Table 1-1 summarizes the preceding software structures, how they are defined, and the concerns that they satisfy.

TABLE 1-1. Structure summary

Structure	Components	Relations	Concerns
Information Hiding	Information Hiding Modules	Is a part of Is contained in	Changeability Modularity Buildability
Uses	Programs	Uses	Producibility Ecosystem
Process	Processes (tasks, threads)	Gives work to Gets resources from Shares resources with Contained in ...	Performance Changeability Capacity
Data Access	Programs and Segments	Has access to	Security Ecosystem

Good Architectures

Recall that architects play a game of trade-offs. For a given set of functional and quality requirements, there is no single correct architecture and no single "right answer." We know from experience that we should evaluate an architecture to determine whether it will meet its requirements before spending money to build, test, and deploy the system. Evaluation attempts to answer one or more of the concerns discussed in previous sections, or concerns specific to a particular system.

There are two common approaches to architecture evaluation (Clements, Kazman, and Klein 2002). The first class of evaluation methods determines properties of the architecture, often by modeling or simulation of one or more aspects of the system. For example, performance modeling is carried out to assess throughput and scalability, and fault tree models can be used to estimate reliability and availability. Other types of models include using complexity and coupling metrics to assess changeability and maintainability.

The second, and broadest, class of evaluation methods is based on questioning the architects to assess the architecture. There are many structured questioning methods. For example, the

Software Architecture Review Board (SARB) process developed at Bell Labs uses experts from within the organization and leverages their deep domain expertise in telecommunications and related applications (Maranzano et al. 2005).

Another variation of the questioning approach is the Architecture Trade-off Analysis Method (ATAM) (Clements, Kazman, and Klein 2002), which looks for risks that the architecture will not satisfy quality concerns. ATAM uses scenarios, each describing a particular stakeholder's quality concern for the system. The architects then explain how the architecture supports each of the scenarios.

Active reviews are another type of questioning approach that turns the process on its head, requiring the architects to provide the reviewers with the questions that the architects think are important to answer (Hoffman and Weiss 2000, chap. 17). The reviewers then use the existing architecture documents and descriptions to answer the questions. Finally, searching the Web for "software architecture review checklist" returns dozens of checklists, some very general and some specific to an application domain or technology framework.

Beautiful Architectures

All of the preceding methods help to evaluate whether an architecture is "good enough"—that is, whether it is likely to guide the developer and testers to produce a system that will satisfy the functional and quality concerns of the system's stakeholders. There are many good architectures in systems that we use every day.

But what about architectures that are more than good enough? What if there were a "Software Architecture Hall of Fame"? Which architectures would line the walls of that gallery? The idea is not as far-fetched as you might think—in the field of software product lines, just such a Hall of Fame exists.* The criteria for induction into the Software Product Line Hall of Fame include commercial success, influence on other product line architectures (others have "borrowed, copied, or stolen" from the architecture), and sufficient documentation that others can understand the architecture "without resorting to hearsay."

What criteria would we add to these for nominees for a more general "Architecture Hall of Fame," or perhaps a "Gallery of Beautiful Architectures"?

First, we should recognize that this is a gallery of software systems, not art, and our systems are built to be used. So, perhaps we should begin by looking at the Utility of the architecture: it should be used every day by many people.

But before an architecture can be used, it must be built, and so we should look at the Buildability of the architecture. We would look for architectures with a well-defined Uses Structure that would support incremental construction, so that at each iteration of construction we would have a useful, testable system. We would also look for architectures that have

* See *http://www.sei.cmu.edu/productlines/plp_hof.html*.

well-defined module interfaces and that are inherently testable, so that the construction progress is transparent and visible.

Next, we want architectures that demonstrate Persistence—that is, architectures that have stood the test of time. We work in an era when the technical environment is changing at an ever-increasing rate. A beautiful architecture should anticipate the need for change, and allow expected changes to be made easily and efficiently. We want to find architectures that have avoided the "aging horizon" (Klein 2005) beyond which maintenance becomes prohibitively expensive.

Finally, we would want to include architectures that have features that delight the developers and testers who use the architecture and build it and maintain it, as well as the users of the system(s) built from it. Why delight developers? It makes their job easier and is more likely to result in a high-quality system. Why delight testers? They are the ones who have to attempt to emulate what the users will do as part of the testing process. If they are delighted, it is likely that the users will be, too. Think of the chef who is unhappy with his culinary creations. His customers, who consume those creations, are likely to be unhappy, too.

Different systems and application domains offer opportunities for architectures to exhibit specific delightful features, but Conceptual Integrity is a feature that cuts across all domains and that always delights. A consistent architecture is easier and faster to learn, and once you know a little, you can begin to predict the rest. Without the need to remember and handle special cases, code is cleaner and test sets are smaller. A consistent architecture does not offer two (or more) ways to do the same thing, forcing the user to waste time choosing. As Ludwig Mies van der Rohe said of good design, "Less is more," and Albert Einstein might say that beautiful architectures are as simple as possible, but no simpler.

Given these criteria, we propose some initial candidates for our "Gallery of Beautiful Architectures."

The first entry is the architecture for the A-7E Onboard Flight Processor (OFP), developed at the Naval Research Laboratory (NRL) in the late 1970s, and described in Bass, Clements, and Kazman (2003). Although this particular system never went into production, it meets every other criterion for inclusion. This architecture has had tremendous influence on the practice of software architecture by demonstrating in a real-world system the separation of a design-time Information Hiding Module and Uses structures from the runtime Process Structures. It showed that information hiding could be used as a primary decomposition principle for a complex system. Since the U.S. government funded and developed the architecture, all project documentation is available in the public domain.† The architecture had a well-defined Uses structure that facilitated incremental construction of the system. Finally, the Information Hiding Module structure provided clear and consistent criteria for decomposing the system,

† See, for example, Chapters 6, 15, and 16 in Hoffman and Weiss (2000), or conduct a search for "A-7E" in the NRL Digital Archives (*http://torpedo.nrl.navy.mil/tu/ps*).