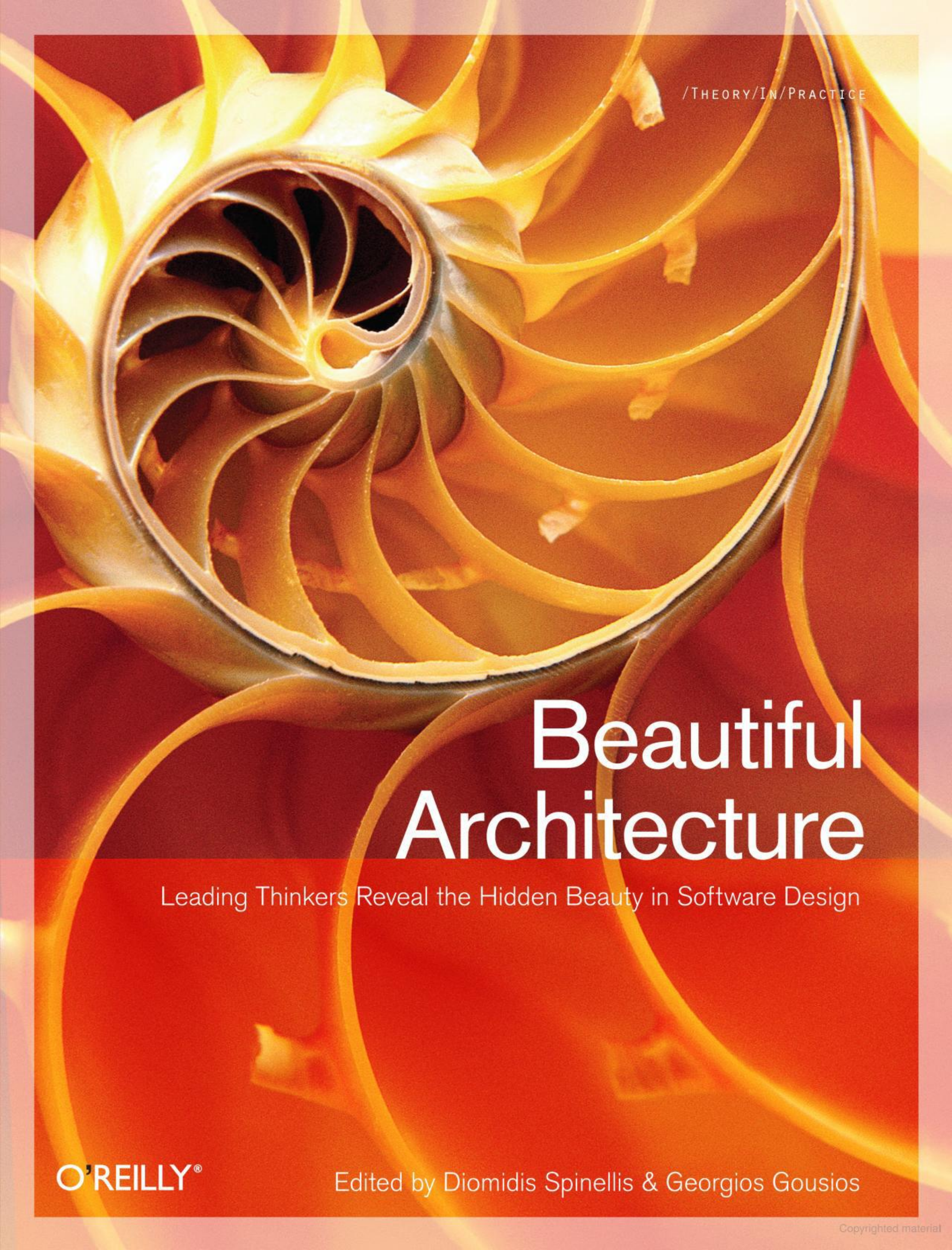


/Theory/In/Practice

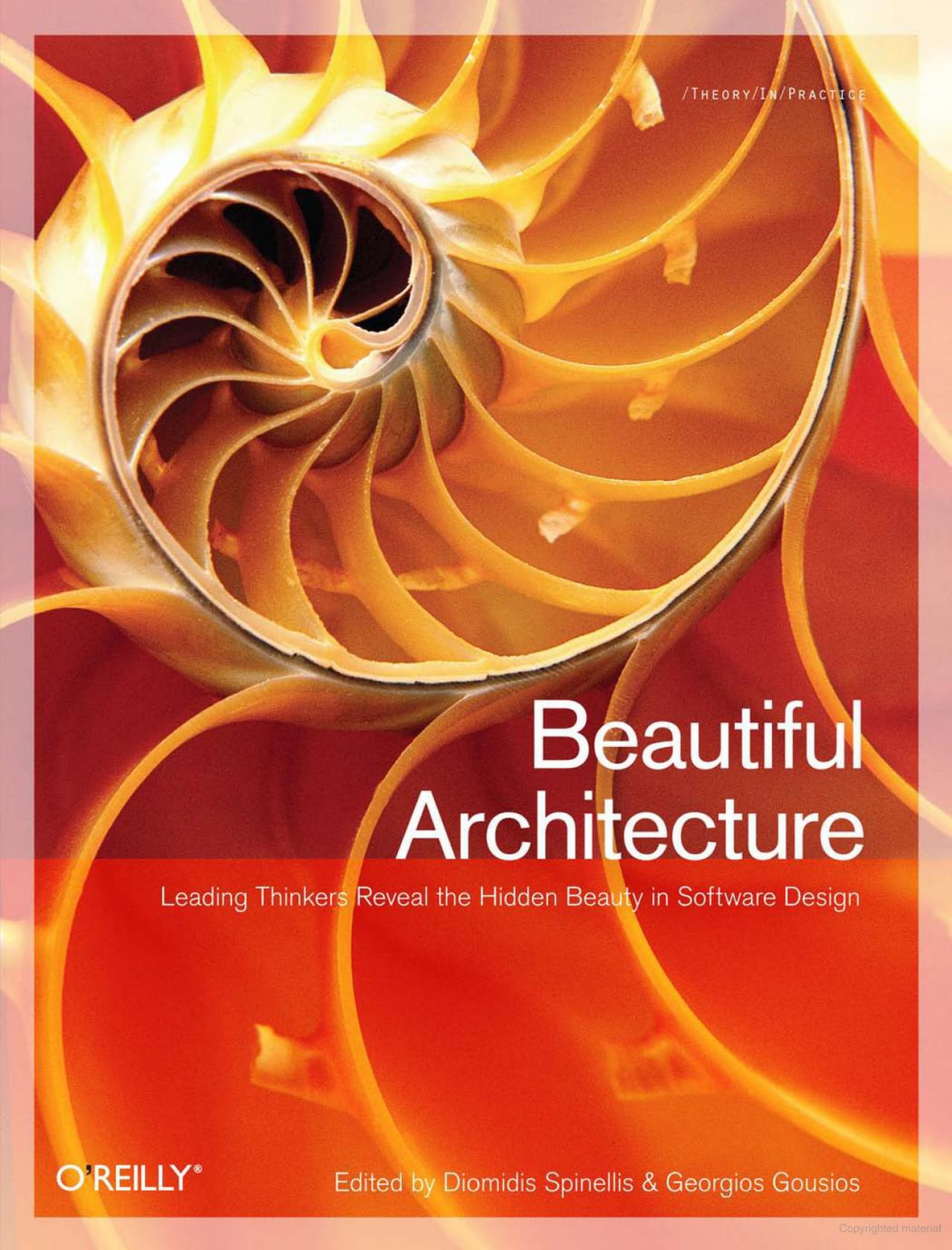
Beautiful Architecture

Leading Thinkers Reveal the Hidden Beauty in Software Design

O'REILLY®

Edited by Diomidis Spinellis & Georgios Gousios

Copyrighted material



/THEORY/IN/PRACTICE

Beautiful Architecture

Leading Thinkers Reveal the Hidden Beauty in Software Design

O'REILLY®

Edited by Diomidis Spinellis & Georgios Gousios

Copyrighted material

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

Copyrighted material

CONTENTS

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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."

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

`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*

Copyrighted material

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.

Copyrighted material

Copyrighted material

PART I

On Architecture

Copyrighted material

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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

Copyrighted material

> 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?

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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.*

Copyrighted material

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*).

Copyrighted material

resulting in strong Conceptual Integrity. As an exemplar of embedded system software architecture, the A-7E OFP certainly belongs in our gallery.

Another architecture that we would want to include in our gallery is the software architecture for the Lucent 5ESS telephone switch (Carney et al. 1985). The 5ESS has been a global commercial success, providing core telephone network switching for networks in countries around the world. It has set the standard for performance and availability, with each unit capable of handling over one million call connections per hour with less than 10 seconds of unplanned downtime per year (Alcatel-Lucent 1999). The architecture's unifying concepts, such as the "half call model" for managing telephone connections, have become standard patterns in the domains of telephony and network protocols (Hanmer 2001). In addition to keeping the number of call types that must be handled to $2n$, where n is the number of call protocols, the half call pattern links the operating system concept of process to the telephony concept of call type, thereby providing a simple design rule and introducing a beautiful Conceptual Integrity. A development team of up to 3,000 people has evolved and enhanced the system over the past 25 years. Based on success, persistence, and influence, the 5ESS architecture is a fine addition to our gallery.

Another system to consider for inclusion in our Gallery of Beautiful Architectures is the architecture of the World Wide Web (WWW), created by Tim Berners-Lee at CERN, and described in Bass, Clements, and Kazman (2003). The WWW has certainly been commercially successful, transforming the way that people use the Internet. The architecture has remained intact, even as new applications are created and new capabilities introduced. The overall simplicity of the architecture contributes to its Conceptual Integrity, but decisions such as using a single library for both clients and servers and creating a layered architecture to separate concerns have ensured that the integrity of the architecture remains intact. The persistence of the core WWW architecture and its ability to continue to support new extensions and features certainly qualify it for inclusion in our gallery.

Our last example is the Unix system, which exhibits conceptual integrity, is widely used, and has had great influence. The pipe and filters design is a lovely abstraction that permits rapid construction of new applications.

WHAT'S AN ARCHITECT?

A stranger is traveling down a road on a hot summer day. As he progresses, he comes upon a man working by the side of the road breaking rocks.

"What are you doing?" he asks the man.

The man looks up at him. "I'm breaking rocks. What does it look like I'm doing? Now get out of my way and let me get back to it."

The stranger continues down the road and soon comes upon a second man breaking rocks in the hot sun. The man is working hard and sweating freely.

Copyrighted material

"What are you doing?" asks the stranger.

The man looks up and smiles.

"I'm working for a living," he says. "But it's hard work. Maybe you have a better job for me?"

The stranger shakes his head and moves on. Pretty soon he comes on a third man breaking rocks. The sun is at its zenith now, the man is straining, and sweat is pouring off him.

"What are you doing?" asks the stranger. The man pauses, takes a drink of water, smiles, and raises his arms to the sky.

"I'm building a cathedral," he breathes.

The stranger looks at him for a moment and says, "We're starting a new company. How would you like to be our chief architect?"

We have gone to considerable length to describe architectures, the role of architects, and considerations that go into creating architectures, and we have offered several brief examples of beautiful architectures. We invite you now to read more detailed examples from accomplished architects in the following chapters as they describe the beautiful architectures that they have created and used.

Acknowledgments

David Parnas defined many of the structures we described in several papers, including his "Buzzword" paper (Parnas 1974). Jon Bentley was an inspiration in this work and he, Deborah Hill, and Mark Klein made many useful suggestions on earlier drafts.

References

Alcatel-Lucent. 1999. "Lucent's record-breaking reliability continues to lead the industry according to latest quality report." *Alcatel-Lucent Press Releases*. June 2. *http://www.alcatel-lucent.com/wps/portal/NewsReleases/DetailLucent?LMSG_CABINET=Docs_and_Resource_Ctr&LMSG_CONTENT_FILE=News_Releases_LU_1999/LU_News_Article_007318.xml* (accessed May 15, 2008).

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice*, Second Edition. Boston, MA: Addison-Wesley.

Blaauw, G., and F. Brooks. 1997. *Computer Architecture: Concepts and Evolution*. Boston, MA: Addison-Wesley.

Booch, G., J. Rumbaugh, and I. Jacobson. 1999. *The UML Modeling Language User Guide*. Boston, MA: Addison-Wesley.

Copyrighted material

Brooks, F. 1995. *The Mythical Man-Month*. Boston, MA: Addison-Wesley.

Carney, D. L., et al. 1985. "The 5ESS switching system: Architectural overview." *AT&T Technical Journal*, vol. 64, no. 6, p. 1339.

Clements, P., et al. 2003. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley.

Clements, P., R. Kazman, and M. Klein. 2002. *Evaluating Software Architectures*. Boston: Addison-Wesley.

Conway, M. 1968. "How do committees invent." *Datamation*, vol. 14, no. 4.

Courtois, P. J. 1977. *Decomposability: Queuing and Computer Systems*. New York, NY: Academic Press.

Dijkstra, E. W. 1968. "Co-operating sequential processes." *Programming Languages*. Ed. F. Genuys. New York, NY: Academic Press.

Garlan, D., and D. Perry. 1995. "Introduction to the special issue on software architecture." *IEEE Transactions on Software Engineering*, vol. 21, no. 4.

Grinter, R. E. 1999. "Systems architecture: Product designing and social engineering." *Proceedings of ACM Conference on Work Activities Coordination and Collaboration (WACC '99)*. 11–18. San Francisco, CA.

Hanmer, R. 2001. "Call processing." *Pattern Languages of Programming (PLoP)*. Monticello, IL. *http://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/rhanmer0/PLoP2001_rhanmer0_1.pdf*.

Hoffman, D., and D. Weiss. 2000. *Software Fundamentals: Collected Papers by David L. Parnas*. Boston, MA: Addison-Wesley.

IEEE. 2000. "Recommended practice for architectural description of software intensive systems." Std 1471. Los Alamitos, CA: IEEE.

Klein, John. 2005. "How does the architect's role change as the software ages?" *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Washington, DC: IEEE Computer Society.

Maranzano, J., et al. 2005. "Architecture reviews: Practice and experience." *IEEE Software*, March/April 2005.

Parnas, David L. 1974. "On a buzzword: Hierarchical structure." *Proceedings of IFIP Congress*. Amsterdam, North Holland. [Reprinted as Chapter 9 in Hoffman and Weiss (2000).]

Waldo, J. 2006. "On system design." *OOPLSA '06*. October 22–26. Portland, OR.

Weiss, D., and C. T. R. Lai. 1999. *Software Product Line Engineering*. Boston, MA: Addison-Wesley.

Copyrighted material

CHAPTER TWO

A Tale of Two Systems: A Modern-Day Software Fable

Pete Goodliffe

Architecture is the art of how to waste space.

—Philip Johnson

A software system is like a city—an intricate network of highways and hostelries, of back roads and buildings. There's a lot going on in a busy city; flows of control are continually being born, weaving their life through it, and dying. A wealth of data is amassed, stored, and destroyed. There are a range of buildings: some tall and beautiful, some squat and functional, others dilapidated and falling into disrepair. As data flows around them there are traffic jams and tailbacks, rush hours and road works. The quality of your software city is directly related to how much town planning went into it.

Some software systems are lucky, created through thoughtful design from experienced architects. They are structured with a sense of elegance and balance. They are well-mapped and easy to navigate. Others are not so lucky, and are essentially software settlements that grew up around the accidental gathering of some code. The transport infrastructure is inadequate, and the buildings are drab and uninspiring. Placed in the middle of it, you'd get completely lost trying to find a route out.

Where would your code rather live? What kind of software city would you rather construct?

Copyrighted material

In this chapter, I tell the story of two such software cities. It's a true story and, like all good stories, this one has a moral at the end. They say *experience is a great teacher*, but other people's experience is even better—if you can learn from these projects' mistakes and successes, you might save yourself (and your software) a lot of pain.

The two systems in this chapter are particularly interesting because they turned out very differently, despite being superficially very similar:

- They were of similar size (around 500,000 lines of code).
- They were both "embedded" consumer audio appliances.
- Each software ecosystem was mature and had gone through many product releases.
- Both solutions were Linux-based.
- The code was written in C++.
- They were both developed by "experienced" programmers (who, in some cases, *should* have known better).
- The programmers themselves were the architects.

In this story, names have been changed to protect the innocent (and the guilty).

The Messy Metropolis

Build up, build up, prepare the road! Remove the obstacles out of the way of my people.

—Isaiah 57:14

The first software system we'll look at is known as the Messy Metropolis. It's one I look back on fondly—not because it was good or because it was enjoyable to work with, but because it taught me a valuable lesson about software development when I first came across it.

My first contact with the Messy Metropolis was when I joined the company that created it. It initially looked like a promising job. I was to join a team working on a Linux-based, "modern" C++ codebase that had been in development for a number of years. Exciting stuff, if you have the same peculiar fetishes as me.

The work wasn't smooth sailing at first, but you never expect an easy ride when you start to work in a new team on a new codebase. However, it didn't get any better as the days (and weeks) rolled by. The code took a *fantastically* long time to learn, and there were no obvious routes into the system. That was a warning sign. At the microlevel, looking at individual lines, methods, and components, the code was messy and badly put together. There was no consistency, no style, and no unifying concepts drawing the separate parts together. That was another warning sign. Control flew around the system in unfathomable and unpredictable ways. That was yet another warning sign. There were so many bad "code smells" (Fowler 1999)

Copyrighted material

that the codebase was not just putrid, it was a pungent landfill site on a hot summer's day. A clear warning sign. The data was rarely kept near where it was used. Often extra baroque caching layers were introduced to try to persuade it to hang around in more convenient places. Again, a warning sign.

As I tried to build a mental picture of the Metropolis, no one was able to explain the structure; no one knew all of its layers, tendrils, and dark, secluded corners. In fact, no one actually knew how any of it really worked (it was actually by a combination of luck and heroic maintenance programmers). People knew the small areas they had worked on, but no one had an overall comprehension of the system. And, naturally, there was no documentation. That was a warning sign. What I needed was a map.

This was the sad story I had become a part of: the Metropolis was a town planning disaster. Before you can improve a mess, you need to understand that mess, so with much effort and perseverance we pulled together a map of the "architecture." We charted every highway, all the arterial roads, the uncharted back roads, and all of the dimly lit side passages, and placed them on one master diagram. For the first time we could see what the software looked like. Not a pretty sight. It was a tangle of blobs and lines. In an effort to make it more comprehensible, we color-coded the control paths to signify their type. Then we stood back.

It was stunning. It was psychedelic. It was as if a drunk spider had stumbled into a few pots of poster paint and then spun a chromatic web across a piece of paper. It looked something like Figure 2-1 (it's a simplified version, with details changed to protect the guilty). Then it became clear. We had all but drawn a map of the London Underground. It even had the circle line.

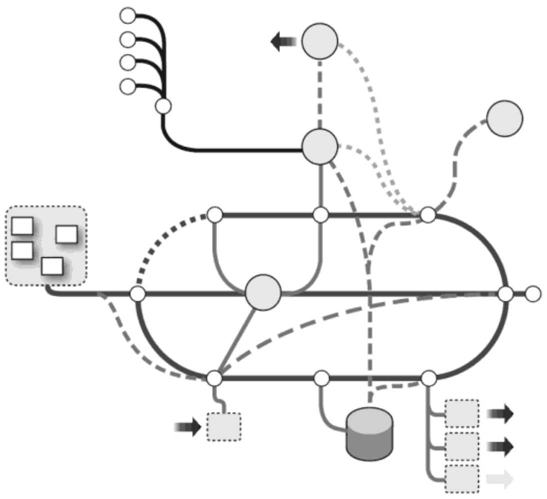


FIGURE 2-1. The Messy Metropolis "architecture"

Copyrighted material

This was the kind of system that would vex a traveling salesman. In fact, the architectural similarity to the London Underground was remarkable: there were many routes to get from one end of the system to the other, and it was rarely obvious how best to do so. Often a destination was geographically nearby but not accessible, and you wished you could bore a new tunnel between two points. Sometimes it would have actually have been better to get out and take a bus. Or walk.

That's not a "good" architecture by any metric. The Metropolis's problems went beyond the design, right up to the development process and company culture. These problems had actually caused a lot of the architectural rot. The code had grown "organically" over a period of years, which is a polite way to say that no one had performed any architectural design of note, and that various bits had been bolted on over time without much thought. No one had ever stopped to impose a sane structure on the code. It had grown by accretion, and was a classic example of a system that had received absolutely no architectural design. But a codebase never has *no* architecture. This just had a very poor one.

The Metropolis's state of affairs was understandable (but not condonable) when you looked at the history of the company that built it: it was a startup with heavy pressure to get many new releases out rapidly. Delays were not tolerable—they would spell financial ruin. The software engineers were driven to get code shipping as quickly as humanly possible (if not sooner). And so the code had been thrown together in a series of mad dashes.

NOTE

Poor company structure and unhealthy development processes will be reflected in a poor software architecture.

Down the Tubes

The Metropolis's lack of town planning had many consequences, which we'll see here. These ramifications were severe and went far beyond what you might naïvely expect of a bad design. The underground train had turned into a roller coaster, headed rapidly downward.

Incomprehensibility

As you can already see, the Metropolis's architecture and its lack of imposed structure had led to a software system that was remarkably tricky to comprehend, and practically impossible to modify. New recruits coming into the project (like myself) were stunned by the complexity and unable to come to grips with what was going on.

The bad design actually encouraged further bad design to be bolted onto it—in fact, it literally forced you to do so—as there was no way to extend the design in a sane way. The path of least resistance for the job in hand was always taken; there was no obvious way to fix the structural problems, and so new functionality was thrown in wherever it would cause less hassle.

Copyrighted material

NOTE

It's important to maintain the quality of a software design. Bad architectural design leads to further bad architectural design.

Lack of cohesion

The system's components were not at all cohesive. Where each one should have had a single, well-defined role, instead each component contained a grab bag of functionality that wasn't necessarily related. This made it hard to determine why a component existed at all, and hard to work out where a particular piece of functionality had been implemented in the system.

Naturally, this made bug fixing a nightmare, which seriously affected the quality and reliability of the software.

Both functionality and data were located in the wrong place in the system. Many things you'd consider "core services" were not implemented in the hub of the system, but were simulated by the outlying modules (at great pain and expense).

Further software archaeology showed why: there had been personality struggles in the original team, and so a few key programmers had begun to build their own little software empires. They'd grab the functionality they thought was cool and plonk it into their module, even if it didn't belong there. To deal with this, they would then make ever more baroque communication mechanisms to stitch the control back to the correct place.

NOTE

The health of the working relationships in your development team will feed directly into the software design. Unhealthy relationships and inflated egos lead to unhealthy software.

COHESION AND COUPLING

Key qualities of software design are *cohesion* and *coupling*. These are not newfangled "object-oriented" concepts; developers have been talking about them for many years, since the emergence of structured design in the early 1970s. We aim to design systems with components that have:

Strong cohesion

Cohesion is a measure of how related functionality is gathered together and how well the parts *inside* a module work as a whole. Cohesion is the glue holding a module together.

Weakly cohesive modules are a sign of bad decomposition. Each module must have a clearly defined role, and not be a grab bag of unrelated functionality.

Low coupling

Coupling is a measure of the interdependency *between* modules—the amount of wiring to and from them. In the simplest designs, modules have little coupling and so are less reliant on one

Copyrighted material

another. Obviously, modules can't be totally decoupled, or they wouldn't be working together at all!

Modules interconnect in many ways, some direct, some indirect. A module can call functions on other modules or be called by other modules. It may use web services or facilities published by another module. It may use another module's data types or share some data (perhaps variables or files).

Good software design limits the lines of communication to only those that are absolutely necessary. These communication lines are part of what determines the architecture.

Unnecessary coupling

The Metropolis had no clear layering. Dependencies between modules were not unidirectional, and coupling was often bidirectional. Component A would hackily reach into the innards of component B to get its work done for one task. Elsewhere, component B had hardcoded calls onto component A. There was no bottom layer or central hub to the system. It was one monolithic blob of software.

This meant that the individual parts of the system were so tightly coupled that you couldn't bring up a skeletal system without creating every single component. Any change in a single component rippled out, requiring changes in many dependent components. The code components did not make sense in isolation.

This made low-level testing impossible. Not only were code-level unit tests impossible to write, but component-level integration tests could not be constructed, as every component depended on almost every other component. Of course, testing had never been a particularly high priority in the company (we didn't have anywhere near enough time to do that), so this "wasn't a problem." Needless to say, the software was not very reliable.

NOTE

Good design takes into account connection mechanisms and the number (and nature) of inter-component connections. The individual parts of a system should be able to stand alone. Tight coupling leads to untestable code.

Code problems

The problems with bad top-level design had wormed their way down to the code level. Problems beget problems (see the discussion of broken windows in Hunt and Davis [1999]). Since there was no common design and no overall project "style," no one bothered with common coding standards, using common libraries, or employing common idioms. There were no naming conventions for components, classes, or files. There was not even a common build system; duct tape, shell scripts, and Perl glue nestled alongside makefiles and Visual Studio project files. Compiling this monster was considered a rite of passage!

Copyrighted material

One of the most subtle yet serious Metropolis problems was duplication. Without a clear design and a clear place for functionality to live, wheels had been reinvented across the entire codebase. Simple things like common algorithms and data structures were repeated across many modules, each implementation with its own set of obscure bugs and quirky behavioral traits. Larger-scale concerns such as external communication and data caching were also implemented multiple times.

More software archaeology showed why: the Metropolis started out as a series of separate prototypes that got tacked together when they should have been thrown away. The Metropolis was actually an accidental conurbation. When stitched together, the code components had never really fit together properly. Over time, the careless stitches began to tear, so the components pulled against one another and caused friction in the codebase, rather than working in harmony.

> NOTE
> **A lax and fuzzy architecture leads to individual code components that are badly written and don't fit well together. It also leads to duplication of code and effort.**

Problems outside the code

The problems within the Metropolis spilled out from the codebase to cause havoc elsewhere in the company. There were problems in the development team, but the architectural rot also affected the people supporting and using the product.

The development team

New recruits coming into the project (like myself) were stunned by the complexity and were unable to come to grips with what was going on. This partially explains why very few new recruits stayed at the company for any length of time—staff turnover was very high.

Those who remained had to work very hard, and stress levels on the project were high. Planning new features instilled a dread fear.

Slow development cycle

Since maintaining the Metropolis was a frightful task, even simple changes or "small" bug fixes took an unpredictable length of time. Managing the software development cycle was difficult, timescales were hard to plan, and the release cycle was cumbersome and slow. Customers were left waiting for important features, and management got increasingly frustrated at the development team's inability to meet business requirements.

Support engineers

The product support engineers had an awful time trying to support a flaky product while working out the intricate behavioral differences between relatively minor software releases.

Copyrighted material

Third-party support

An external control protocol had been developed, enabling other devices to control the Metropolis remotely. Since it was a thin veneer over the guts of the software, it reflected the Metropolis's architecture, which means that it was baroque, hard to understand, prone to fail randomly, and impossible to use. Third-party engineers' lives were also made miserable by the poor structure of the Metropolis.

Intra-company politics

The development problems led to friction between different "tribes" in the company. The development team had strained relations with the marketing and sales guys, and the manufacturing department was permanently stressed every time a release loomed on the horizon. The managers despaired.

NOTE

The consequence of a bad architecture is not constrained within the code. It spills outside to affect people, teams, processes, and timescales.

Clear requirements

Software archaeology highlighted an important reason that the Messy Metropolis turned out so messy: at the very beginning of the project *the team did not know what it was building*.

The parent startup company had an idea of which market it wanted to capture, but didn't know which kind of product to capture it with. So they hedged their bets and asked for a software platform that could do *many* things. *Oh, and we wanted it yesterday.* So the programmers rushed to create a hopelessly general infrastructure that could potentially do many things (badly), rather than craft an architecture that supported one thing well and could be extended to do more in the future.

NOTE

It's important to know what you're designing before you start designing it. If you don't know what it is and what it's supposed to do, *don't design it yet*. Only design what you know you need.

At the earliest stages of Metropolis planning there were far too many architects. With woolly requirements, they all took a disjoint piece of the puzzle and tried to work on it individually. They didn't keep the entire project in sight as they worked, so when they tried to put the puzzle pieces back together, they simply didn't fit. Without time to work on the architecture further, the parts of the software design were left overlapping slightly, and thus began the Metropolis town planning disaster.

Copyrighted material

Where Is It Now?

The Metropolis's design was almost completely irredeemable—believe me, over time we tried to fix it. The amount of effort required to rework, refactor, and correct the problems with the code structure had become prohibitive. A rewrite wasn't a cheap option, as support for the old, baroque control protocol was a requirement.

As you can see, the consequence of the Metropolis's "design" was a diabolical situation that was inexorably getting worse. It was so hard to add new features that people were just applying more kludges, Band-Aids, and calculated fudges. No one enjoyed working with the code, and the project was heading in a downward spiral. The lack of design had led to bad code, which led to bad team morale and increasingly lengthy development cycles. This eventually led to severe financial problems for the company.

Eventually, management acknowledged that the Messy Metropolis had become uneconomical, and it was thrown away. This is a brave step for any organization, especially one that is constantly running 10 paces ahead of itself while trying to tread water. With all of the C++ and Linux experience the team had gained form the previous version, the system was rewritten in C# on Windows. Go figure.

A Postcard from the Metropolis

So what have we learned? Bad architecture can have a profound effect and severe repercussions. The lack of foresight and architectural design in the Messy Metropolis led to:

- A low-quality product with infrequent releases
- An inflexible system that couldn't accommodate change or the addition of new functionality
- Pervasive code problems
- Staffing problems (stress, low morale, turnover, etc.)
- A lot of messy internal company politics
- Lack of success for the company
- Many painful headaches and late nights working on the code

Design Town

Form ever follows function.

—Louis Henry Sullivan

The Design Town software project was superficially very similar to the Messy Metropolis. It too was a consumer audio product written in C++, running on a Linux operating system.

Copyrighted material

However, it was built in a very different way, and so the internal structure worked out very differently.

I was involved with the Design Town project from the very start. A brand-new team of capable developers had been assembled to build it from scratch. The team was small (initially four programmers) and, like the Metropolis, the team structure was flat. Fortunately, there was none of the interpersonal rivalry apparent in the Metropolis project, or any vying for positions of power in the team. The members didn't know each other well beforehand and didn't know how well we'd work together, but we were all enthused about the project and relished the challenge.

So far, so good.

Linux and C++ were early decisions for the project, and that shaped the team that had been assembled. From the outset the project had clearly defined goals: a particular first product and a roadmap of future functionality that the codebase had to accommodate. This was to be a general-purpose codebase that would be applied in a number of product configurations.

The development process employed was eXtreme Programming (or XP) (Beck and Andres 2004), which many believe eschews design: *code from the hip, and don't think too far ahead*. In fact, some observers were shocked at our choice and predicted that it would all end in tears, just like the Metropolis. But this is a common misconception. XP does not discourage design; it discourages work that isn't necessary (this is the YAGNI, or *You Aren't Going To Need It*, principle). However, where upfront design is required, XP requires you to do that. It also encourages rapid prototypes (known as *spikes*) to flesh out and prove the validity of designs. Both of these were very useful and contributed greatly to the final software design.

First Steps into Design Town

Early in the design process, we established the main areas of functionality (these included the core audio path, content management, and user control/interface). We considered where they each fit in the system, and an initial architecture was fleshed out, including the core threading models that were necessary to achieve performance requirements.

The relative positions of the separate parts of the system was established in a conventional layer diagram, a simplified part of which is shown in Figure 2-2. Notice that this was *not* a big upfront design. It was an intentionally simple conceptual model of the Design Town: just some blobs on a diagram, a basic system design that could grow easily as pieces of functionality were added. Although basic, this initial architecture proved a solid basis for growth. Whereas the Metropolis had no overall picture and saw functionality grafted (or bodged) in wherever was "convenient," this system had a clear model of what belonged where.

Extra design time was spent on the heart of the system: the audio path. It was essentially an internal subarchitecture of the entire system. To define this, we considered the flow of data through a series of components and arrived at a filter-and-pipeline audio architecture, similar

Copyrighted material

to Figure 2-3. The products involved a number of these pipelines, depending on their physical configuration. Again, at first this pipeline was nothing more than a concept—more blobs on a diagram. We hadn't decided how it would all be stitched together.

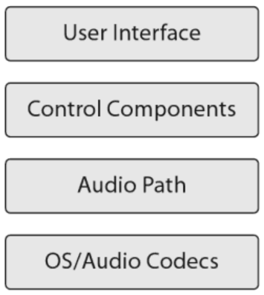


FIGURE 2-2. The Design Town initial architecture

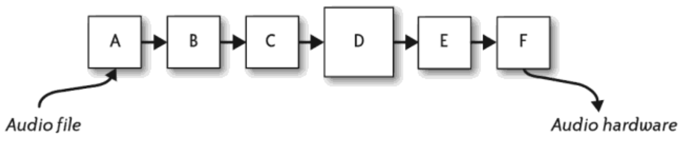


FIGURE 2-3. The Design Town audio pipeline

We also made an early choice of supporting libraries the project would employ (for example, the Boost C++ libraries available at *http://www.boost.org* and a set of database libraries). Decisions about some of the basic concerns were made at this point to ensure that the code would grow easily and cohesively, including:

- The top-level file structure
- How we would name things
- A "house" presentation style
- Common coding idioms
- The choice of unit test framework
- The supporting infrastructure (e.g., source control, a suitable build system, and continuous integration)

These "fine detail" factors were very important: they allied closely with the software architecture and, in turn, influenced many later design decisions.

Copyrighted material

The Story Unfolds

Once the initial design had been established by the team, the Design Town project proceeded following the XP process. Design and code construction was either done in pairs or carefully reviewed to ensure that work was correct.

The design and the code developed and matured over time, and as the story of Design Town unfolded, there were the following consequences.

Locating functionality

With a clear overview of the system structure in place from the very beginning, new units of functionality were consistently added to the correct functional areas of the codebase. There was never a question about where code belonged. It was also easy to find the implementation of existing functionality in order to extend it or to fix problems.

Now, sometimes putting new code in the "right" place was harder than simply bodging it into a more convenient, but less tasteful, place. So the existence of an architectural plan sometimes made the developers work harder. The payoff for this extra effort was a *much* easier life later on, when maintaining or extending the system—there was very little cruft to trip over.

NOTE

An architecture helps you to locate functionality: to add it, to modify it, or to fix it. It provides a template for you to slot work into and a map to navigate the system.

Consistency

The entire system was consistent. Every decision at every level was taken in the context of the whole design. The developers did this intentionally from the outset so all the code produced matched the design fully, and matched all the other code written.

Over the project's history, despite many changes ranging across the entire scope of the codebase—from individual lines of code to the system structure—everything followed the original design template.

NOTE

A clear architectural design leads to a consistent system. All decisions should be made in the context of the architectural design.

The good taste and elegance of the top-level design naturally fed down to the lower levels. Even at the lowest levels, the code was uniform and neat. A clearly defined software design ensured that there was no duplication, that familiar design patterns were used throughout, familiar interface idioms were adopted, and that there were no unusual object lifetimes or odd resource management issues. Lines of code were written in the context of the town plan.

Copyrighted material

NOTE
Clear architecture helps reduce duplication of functionality.

Growing the architecture

Some entirely new functional areas appeared in the "big picture" design—storage management and an external control facility, for example. In the Metropolis project, this was a crushing blow and incredibly hard to do. But in Design Town, things worked differently.

The system design, like the code, was considered malleable and refactorable. One of the development team's core principles was to stay nimble—that nothing should be set in stone—and so the architecture could be changed when necessary. This encouraged us to keep our designs simple and easy to change. Consequently, the code could grow rapidly and maintain a good internal structure. Accommodating new functional blocks was not a problem.

NOTE
Software architecture is not set in stone. Change it if you need to. To be changeable, the architecture must remain simple. Resist changes that compromise simplicity.

Deferring design decisions

One of the XP principles that really enhanced the quality of the Design Town architecture was YAGNI (don't do anything if *you aren't going to need it*). It encouraged us to design only the important stuff early on, and to defer all remaining decisions until later, when we had a clearer picture of the actual requirements and how best to fit them into the system. This is an immensely powerful design approach, and quite liberating.

- One of the worst things you can do is design something you don't yet understand. YAGNI forces you to wait until you know what the problem really is and how it should be accommodated by the design. It eliminates guesswork and ensures the design will be correct.
- It is dangerous to add everything you *might* need (including the kitchen sink) to a software design when you first create it. Most of your design work will be wasted effort, and produce extra baggage that you'll need to support over the entire changing life of the software. It costs more at first, and continues to cost over the life of the project.

NOTE
Defer design decisions until you *have* to make them. Don't make architectural decisions when you don't know the requirements yet. Don't guess.

Maintaining quality

From the outset, the Design Town project put a number of quality control processes in place:

- Pair programming

Copyrighted material

- Code/design reviews for anything not pair-programmed
- Unit tests for every piece of code

These processes ensured that the system never had an incorrect, badly fitting change applied. Anything that didn't mesh with the software design was rejected. This might sound draconian, but they were processes that the developers bought into.

This buy-in highlights an important attitude: the developers believed in the design, and considered it important enough to protect. They took ownership of, and personal responsibility for, the design.

> NOTE
> **Architectural quality must be maintained. This can happen only when the developers are given and take responsibility for it.**

Managing technical debt

Despite these quality control measures, Design Town development was fairly pragmatic. As deadlines approached, a number of corners were cut to allow projects to ship on time. Small code "sins" or design warts were allowed to enter the codebase, either to get functionality working quickly or to avoid high-risk changes near a release.

However, unlike the Messy Metropolis project, these fudges were marked as *technical debt* and scheduled for later revision. These warts stood out clearly, and the developers were not happy about them until they were dealt with. Again, we see the developers taking responsibility for the quality of the design.

Unit tests shape design

One of the core decisions about the codebase (which is also mandated by XP development) was that everything should be unit tested. Unit testing brings many advantages, one of which is the ability to change sections of the software without worrying about destroying everything else in the process. Some areas of the Design Town internal structure received quite radical rework, and the unit tests gave us confidence that the rest of the system had not been broken. For example, the thread model and interconnection interface of the audio pipeline was changed fundamentally. This was a serious design change relatively late in the development of that subsystem, but the rest of the code interfacing with the audio path continued executing perfectly. The unit tests enabled us to change the design.

This kind of "major" design change slowed down as Design Town matured. After an amount of design rework, things settled down, and subsequently there were only minor design changes. The system developed quickly, in an iterative manner, with each step improving the design, until it reached a relatively stable plateau.

Copyrighted material

NOTE

Having a good set of automated tests for your system allows you to make fundamental architectural changes with minimal risk. It gives you space in which to work.

Another major benefit of the unit tests was their remarkable shaping of the code design: they practically enforced good structure. Each small code component was crafted as a well-defined entity that could stand alone, as it had to be constructible in a unit test without requiring the rest of the system to be built up around it. Writing unit tests ensured that each module of code was internally cohesive and loosely coupled from the rest of the system. The unit tests forced careful thought about each unit's interface, and ensured that the unit's API was meaningful and internally consistent.

NOTE

Unit testing your code leads to better software designs, so design for testability.

Time for design

One of the contributing factors to Design Town's success was the allotted development timescale, which was neither too long nor too short (just like Goldilocks's porridge). A project needs a conducive environment in which to thrive.

Given too much time, programmers often want to create their magnum opus (the kind of thing that will always be *almost* ready, but never quite materializes). A little pressure is a wonderful thing, and a sense of urgency helps to get things done. However, given too little time, it simply isn't possible to achieve any worthwhile design, and you'll get only a half-baked solution rushed out—just like the Metropolis.

NOTE

Good project planning leads to superior designs. Allot sufficient time to create an architectural masterpiece—they don't appear instantly.

Working with the design

Although the codebase was large, it was coherent and easily understood. New programmers could pick it up and work with it relatively easily. There were no unnecessarily complex interconnections to understand, or weird legacy code to work around.

Since the code has generated relatively few problems and is still enjoyable to work with, there has been very, very low turnover of team members. This is due in part to the developers taking ownership of the design and continually wanting to improve it.

It was interesting to observe how the development team dynamics followed the architecture. Design Town project principles mandated that no one "owned" any area of the design, meaning that any developer could work anywhere in the system. Everyone was expected to write

Copyrighted material

high-quality code. Whereas the Metropolis was a sprawling mess created by many uncoordinated, fighting programmers, Design Town was clean and cohesive, closely cooperating software components created by closely cooperating colleagues. In many ways, Conway's Law[*] worked in reverse, and the team gelled together as the software did.

NOTE

A team's organization has an inevitable affect on the code it produces. Over time, the architecture also affects how well the team works together. When teams separate, the code interacts clumsily. When they work together, the architecture integrates well.

Where Is It Now?

After some time, the Design Town architecture looked like Figure 2-4. That is, it was remarkably similar to the original design, with a few notable changes—and a lot more experience to prove the design was right. A healthy development process, a smaller, more thoughtful development team, and an appropriate focus on ensuring consistency led to an incredibly simple, clear, and consistent design. This simplicity worked to the advantage of the Design Town, leading to malleable code and rapidly developed products.

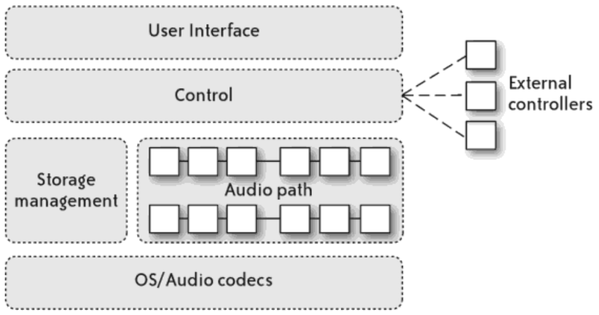


FIGURE 2-4. The Design Town final architecture

At the time of this writing, the Design Town project has been alive for three years. The codebase is still in production use and has spawned a number of successful products. It is still being developed, still growing, still being extended, and still being changed daily. Its design next month might be quite different from how it looks this month, but it probably won't.

Let me make this clear: the code is by no means perfect. It has areas of technical debt that need work, but they stick out against the backdrop of neatness and will be addressed in the future. Nothing is set in stone, and thanks to the adaptable architecture and flexible code structure,

[*] Conway's Law states that code structure follows team structure. Simply stated, it says, "If you have four groups working on a compiler, you'll get a four-pass compiler."

Copyrighted material

these things can be fixed. Almost everything is in the right place, because the architecture is sound.

So What?

When perfection comes, the imperfect disappears.

—1 Corinthians 13:10

This simple story about two software systems is certainly not an exhaustive treatise on software architecture, but I have shown how architecture profoundly affects a software project. An architecture influences almost everything that comes into contact with it, determining the health of the codebase and also the health of the surrounding areas. Just as a thriving city can bring prosperity and renown to its local area, a good software architecture will help its project to flourish and bring success to those depending on it.

Good architecture is the product of many factors, including (but not limited to):

- Actually doing intentional upfront design. (Many projects fail in this way before they even start.)
- The quality and experience of the designers. (It helps to have made a few mistakes beforehand to point you in the right direction next time! The Metropolis project certainly taught me a thing or two.)
- Keeping the design clearly in view as development progresses.
- The team being given and taking responsibility for the overall design of the software.
- Never being afraid of changing the design: nothing is set in stone.
- Having the right people on the team, including designers, programmers, and managers, and ensuring the development team is the right size. Ensure they have healthy working relationships, as these relationship will inevitably feed into the structure of the code.
- Making design decisions at the appropriate time, when you know all the information necessary to make them. Defer design decisions you cannot yet make.
- Good project management, with the right kind of deadlines.

Your Turn

Never lose a holy curiosity.

—Albert Einstein

You are reading this book right now because you *care* about software architecture, and you care about improving your own software. So here's an excellent opportunity. Consider these simple questions about your software experience to date:

Copyrighted material

1. What's the best system architecture you've ever seen?
   - How did you recognize it as good?
   - What were the consequences of this architecture, both inside and outside the codebase?
   - What have you learned from it?
2. What's the worst architecture system you've ever seen?
   - How did you recognize it as bad?
   - What were the consequences of this architecture, both inside and outside the codebase?
   - What have you learned from it?

References

Beck, Kent, with Cynthia Andres. 2004. *Extreme Programming Explained*, Second Edition. Boston, MA: Addison-Wesley Professional.

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional.

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley Professional.

Copyrighted material

PART II

Enterprise Application Architecture

Copyrighted material

Copyrighted material

	Principles and properties		Structures
✓	Versatility		Module
✓	Conceptual integrity	✓	Dependency
	Independently changeable		Process
	Automatic propagation	✓	Data access
	Buildability		
✓	Growth accommodation		
	Entropy resistance		

CHAPTER THREE

Architecting for Scale

Jim Waldo

Introduction

One of the more interesting problems in designing an architecture for a system is ensuring flexibility in the scale of that system. Scaling is becoming increasingly important, as more of our systems are run on networks or are available on the Web. For such systems, the idea of capacity planning is absurd if you want a margin of error that is under a couple of orders of magnitude. If you put up a site and it becomes popular, you might suddenly find that there are millions of users accessing your site. Just as easily (and just as much of a disaster), you can put up a site and find that no one is particularly interested, and all of the equipment in which you invested now lies idle, soaking up money in energy costs and administrative effort. In the networked world, a site can transition from one of these states to the other in a matter of minutes.

The scaling problem is faced by anyone who attaches a system to a network, but it is particularly interesting in the case of massively multiplayer online games (MMOs) and virtual worlds. These systems must be capable of scaling to large numbers of users. Unlike web servers, however, where the users are requesting fairly static information and are not interacting with each other, players in an MMO or residents in a virtual world are there to interact with both the world (changing the underlying information in the world) and each other. These interplays complicate the scaling of the infrastructures for such systems, as the user interactions with the

Copyrighted material

system are mostly independent (except when they aren't) and don't change much state in the world. Given any two participants in such a world, the likelihood that they are interacting at any given time is vanishingly small. But nearly every player will be interacting with someone nearly all the time. The result is a kind of system that is embarrassingly parallel but interdependent in a small number of interactions.

Scaling of MMOs and virtual worlds is further complicated by the culture that has grown up around these systems. Both MMOs and virtual worlds trace their descent from the production of video games. This is a design culture that grew up in the PC and console game tradition, a tradition in which the programmer could assume that the game ran on a standalone machine or game console. In such an environment, all of the resources of the machine are at the command of the game program, and problems with the program are confined to the single user playing the game (and, in fact, bugs or odd behavior could often be taken as part of the logic of the game itself).

These games, and the companies that write, produce, and enhance them, are part of the entertainment industry. Teams writing a game are led by a producer, and there are scripts and back stories. The goal of a game is to be immersive, persuasive, and most of all, fun. Reliability is nice, but hardly required. Extensibility is a property of the game, allowing new plot lines and themes to be released as upgrades to the game, rather than a property of the code that allows the code to be used in new and different ways.

The rise of online games and virtual worlds brings this culture into an environment where the requirements are much more like those that are faced by the enterprise developer. With multiple players interacting on a server over the network, the crash of a server brought about by the unexpected actions of a player will affect many other players. As these worlds develop economies (some of which interact with the economy of the real world), the stability and consistency of the online world becomes more than just a game. And as the number of players or inhabitants in these worlds reaches the millions, the ability to scale becomes a primary requirement of any architecture.

Project Darkstar (referred to in the rest of this chapter as simply Darkstar) is a response to these changing needs of the builders of games and virtual worlds. The project, undertaken by a research group inside of Sun Microsystems Laboratories, is an ongoing exploration in the architecture of scale. What makes the project particularly interesting is that it is targeted to the MMO and virtual-world builder, a group of programmers who have very different needs from those that we (as system designers) have been used to. The resulting architecture has much that seems familiar until you look at it closely, at which point you can see why it differs from what your experience told you it must be. The result is an architecture with its own sort of beauty, and an object lesson in how different requirements can change the way you have to think about building a system.

Copyrighted material

Context

Like the physical architecture of a building or a city, the architecture of a system has to be adapted to the context in which the artifact built using the architecture will reside. In physical architecture, this context includes the historical surroundings of the work, the climate in which it will exist, the ability of the local artisans and the available building materials, and the intended use of the building. For a software architecture, the context includes not only the applications that will use the architecture, but also the programmers who will build within that architecture and the constraints on the systems that will result.

In building the Darkstar architecture, the first thing we* realized is that any architecture for scaling would need to involve multiple machines. It is not clear that even the largest of mainframes could scale to meet the demands of some of today's online games (*World of Warcraft*, for example, is reported to have five million current subscribers, with hundreds of thousands of them active at any one time). Even if there were a single machine that could handle this load, it would be economically impossible to assume that a game would be so successful that it would require such a hardware investment at the beginning. This kind of application needs to be able to start small and then increase capacity as the user base increases, and then decrease capacity as interest in the game wanes. This maps well to a distributed system, where (reasonably small) machines can be added as demand increases and taken away when demand decreases. Thus we knew at the beginning that the overall architecture would need to be a distributed system.

We also knew that the system would need to exploit the current trends in chip architectures. MMOs and (to a lesser extent) virtual worlds have historically exploited Moore's law for scaling. As a processor doubles in speed, the world that can be created doubles in complexity, richness, and interactivity. No other area of computing has exploited the benefits of increased processor speed in quite the way the game world has. Personal computers designed for games are always pushing the limits of CPU speed, memory, and graphics capabilities. Game consoles push these limits even more aggressively, containing graphics systems far beyond those found in high-end workstations and building the entire machine around the specialized needs of the game player.

The recent change in chip evolution, from the constant increase in clock speeds to the construction of multicore processors, has changed the dynamic of what can be done in games. Rather than doing one thing faster, new chips are being designed to do multiple things at the same time. The introduction of concurrent execution at the chip level will give better total performance if the tasks being run by the chip can in fact be executed at the same time. Without

* In talking about the development of the Project Darkstar architecture, I will generally refer to what "we" did rather than speak about what "I" did. This is more than the use of the editorial "we." The design of the architecture was very much a collaborative project, started by Jeffrey Kesselman, Seth Proctor, and James Megquier, and put into its current form by Seth, James, Tim Blackman, Ann Wollrath, Jane Loizeaux, and me.

Copyrighted material

a change in clock speed, a chip with four cores ought to be able to do four times as much as a chip with a single core. In fact, the speed-up will not be quite linear, as there are other parts of the system that are not made concurrent in the same way. But increases in the overall performance of the system can be obtained by the use of concurrency, and building chips for such concurrent use is far simpler than building chips in which the clock speed is increased.

On the face of it, MMOs and virtual worlds ought to be reasonable candidates for multicore chips and distributed systems. Most of what goes on in an MMO or virtual world, like most of what goes on in the real world, is independent of the other things that are happening in that world. Players go on their own quests or decorate their own rooms. They battle monsters or design clothes. Even when they are engaged with another player or occupant of the world, they are interacting with only a very small percentage of the occupants of the world. This is the characterization of an embarrassingly parallel computational task, and that is just the sort of thing that multiple cores and multiple machines ought to be good at doing.

Although the tasks in these systems may be embarrassingly parallel, the programmers who work on such systems are not trained or experienced in the techniques of either distributed computing or concurrent programming. These are exceptionally subtle fields, difficult even for those who have been trained in them and who have considerable experience in using these techniques. To ask most game programmers to develop a highly concurrent, distributed game server would be asking them to go well outside of their area of expertise or experience.

The First Goal

This context gave us our first goal for the architecture. The requirements for scaling dictated that the system be distributed and concurrent, but we needed to present a much simpler programming model to the game developer. The simple statement of the goal is that the game developer should see the system as a single machine running a single thread, and all of the mechanisms that would allow deployment on multiple threads and multiple machines should be taken care of by the Project Darkstar infrastructure.

In the general case, hiding either distribution or concurrency from the application is not possible. But MMOs and virtual worlds are not the general case. The kind of hiding that we are trying to accomplish comes at the price of requiring a very specific and restricted programming model. Fortunately, it is just the sort of model that lends itself to the kind of programming already used in the server-side components of games and virtual worlds.

The general programming model that Project Darkstar requires is a reactive one, in which the server side of the game is written as a listener for events generated by the clients (that is, the machines being used by the game players, generally either a PC or a game console). When an event is detected, the game server should generate a task, which is a short-lived sequence of computations that includes manipulation of information in the virtual world and communication with the client that generated the original event and possibly with other clients. Tasks can also be generated by the game server itself, either as a response to some

Copyrighted material

internal change or on a periodic, timed basis. In this way, the game server can generate characters in the game or world that aren't controlled by some outside player.

This sort of programming model fits well with games and virtual worlds, but is also used in a number of enterprise-level architectures, such as J2EE and web services. The need to build an architecture different from those enterprise mechanisms was dictated by the very different environment in which MMOs and virtual worlds exist. This environment is nearly a mirror image of the classic enterprise environments, which means that if you have been trained in the enterprise environment, almost everything you know is going to be wrong in this new world.

The classic enterprise environment is envisioned as a thin client connected to a thick server (which is itself often connected to an even thicker database server). The server will hold most of the information needed by the clients, and will act as a filter to the backend database. Very little state is held at the client; in the best case, the client has very little memory, no disk of its own, and is a highly competent display device for the server, which is where most of the real work occurs.

The Game World

The MMO and virtual world environment starts with a very thick client—typically a top-of-the-line PC with the most powerful CPU available, lots of memory, and a graphics card that is itself computationally excellent, or a game console that is specially designed for graphics-intensive, highly interactive tasks. As much as possible, data is pushed out to these clients, especially data that is unchanging, such as geographic information, texture maps, and rule sets. The server is kept as simple as possible, generally holding a very abstract representation of the world and the entities within that world. Further, the server is designed to do as little computation as possible. Most of the computation goes on at the client. The real job of the server is to hold the shared truth of the state of the world, ensuring that any variation in the view of the world held at the various clients can be corrected as needed. The truth needs to be held by the server, since those who control the clients have a vested interest in maximizing their own performance, and thus might be tempted to change the shared truth (if they could) in their favor. Put more directly, players will cheat if they can, so the server must be the ultimate source of shared truth.

The data access patterns of MMOs and virtual worlds are also quite different from those that are seen in enterprise situations. The usual rule of thumb within the enterprise is that 90% of data accesses will be read-only, and most tasks read a large amount of data before altering a small amount. In the MMO and virtual world environment, most tasks access only a very small amount of the state on the server, but of the data that they access, about half of it will be altered.

Copyrighted material

Latency Is the Enemy

But the biggest difference in the two environments traces back to the differences in what the users are doing. In an enterprise environment, the goal is to conduct business, and some lags in processing are acceptable if the overall throughput is improved. In the MMO and virtual world environment, the goal is to have fun, and latency is the enemy of fun. So the infrastructure for an MMO or virtual world needs to be designed around the requirement of bounding latency whenever possible, even at the cost of throughput.

Online games and virtual worlds have clearly found ways to scale to large numbers of users. The current mechanisms fall into two groups. The first of these is geographic in nature. The game is designed as a group of different areas, with each area designed to be run on a single server. It might be an island or room in a virtual world or a town or valley in an online game. The design of the game tries to make each geographic area independent, and scale the geographic area in such a way that the server will not be overwhelmed by too many users occupying the area. In practice, such areas are often self-limiting, as when the server is being overwhelmed, the play becomes less responsive and less interesting. As a result, players leave for more interesting areas, which makes the formerly overwhelmed area less occupied and improves response time.

The problem with scaling by assigning geographic areas to different servers is that the decision of what areas scale to a server must be made when the game is being written. Although new areas might be able to be added to a game or world fairly easily, changing the area that is assigned to a server is something that requires changing the code. The decision of what areas are the unit of scale has to be made as part of development.

A second way of dealing with areas that are overcrowded in a game or world is known as *sharding*. A shard is a copy of the area, run on its own server and independent of other shards, that presents the same portion of the game as the original area. Thus, a shard might present a different copy of a particular room or village, allowing twice as many players to occupy that part of the world. The drawback of shards is that they do not allow players in different shards to interact with each other. As games and worlds become more social experiences than simple game play, this disadvantage can be major. The goal of players is not only to be in the virtual world, but to occupy it with their (real or virtual) friends. Sharding interferes with that goal.

Thus, another major goal of the Darkstar architecture is to allow on-the-fly scaling in a way that does not require the game logic to become involved in the scaling. The infrastructure should allow the game to dynamically react to the load rather than make such reaction part of the design of the game.

Copyrighted material

The Architecture

Darkstar is built as a set of separate services available in the address space of the server side of a game or virtual world. Each service is defined by a small programming interface. Although not the original intention, the basic services provided by Project Darkstar are much like those of a classic operating system, allowing the server side of the game or virtual world to access persistent storage, schedule and run tasks, and perform communication with the client side of the game or virtual world.

Structuring the system as an interconnected set of services is an obvious way to begin the process of divide and conquer that is basic to the design of any large computer system. Each service can be characterized by an interface that protects those using the service from changes in the underlying implementation, and allows those implementations to be undertaken independently. Changes in the implementation of one service ought not affect the implementation of another, even if that other service makes use of the implementation being changed (assuming the interface and the semantics of the interface don't change).

We had other reasons to adopt the service decomposition approach. From the very beginning, Project Darkstar was envisioned as an open source project, with the hope that we could leverage the work of the core team by allowing other members of the community to build additional services that could enrich the functionality of the core. Running an open source community is complicated under any circumstance, and we believed that having the greatest level of isolation between the services that make up the infrastructure would allow a higher level of isolation between different service implementation levels. Additionally, it was not clear that there was a single set of services that would be just right for all MMOs and virtual worlds. By structuring the infrastructure as a set of independent services, different sets of those services could be used in different circumstances dictated by the needs of the particular project using the infrastructure. The services included in any particular Darkstar stack can be set by a configuration file.

The Macro Structure

Figure 3-1 shows the basic structure of a game or virtual world based on the Project Darkstar infrastructure. There will be some number of servers that form the backend of the game or virtual world. Each of these servers runs a copy of the selected set of services (labeled the Darkstar stack) and a copy of the game logic. Clients will connect to one of these servers to interact with the abstract representation of the world held by the server.

Copyrighted material

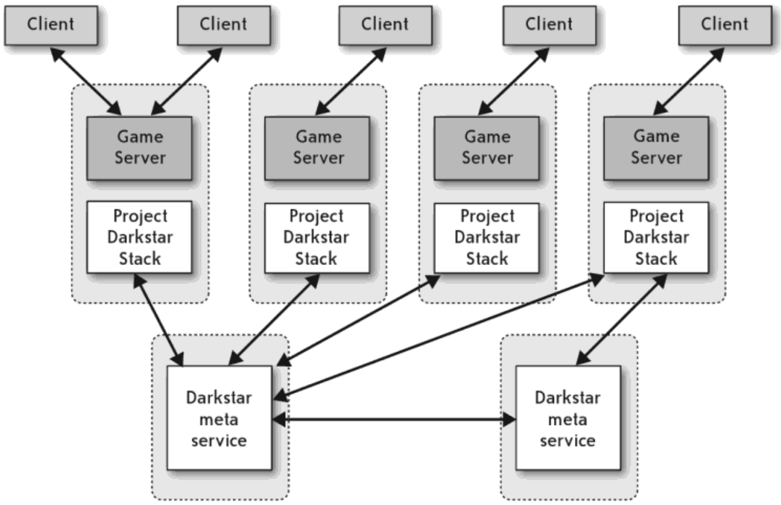


FIGURE 3-1. Project Darkstar high-level architecture

Unlike most replication schemes, the different copies of the game logic are not meant to process the same events. Instead, each copy can independently interact with the clients. Replication in this design is used primarily to allow scale rather than to ensure fault tolerance (although, as we will see later, fault tolerance is also achieved). Further, the game logic itself does not know or need to know that there are other copies of the server operating on other machines. The code written by the game programmer runs as if it were on a single machine, with coordination of the different copies done by the Project Darkstar infrastructure. Indeed, it is possible to run a Darkstar-based game on a single server if that is all the capacity the game needs.

Clients connect to the game logic using communication mechanisms that are part of the infrastructure. These mechanisms allow either direct client-to-server communication or a form of publish-subscribe channel, where any message sent on a channel is delivered to all of those subscribed to the channel.

The Darkstar stacks are coordinated by a set of meta-services—network-accessible services that are hidden from the game or virtual world programmer. These meta-services allow the various copies of the stack to coordinate the overall operation of the game. These meta-services will, for example, make sure that all of the separate copies continue to run and initiate failure recovery if some copy fails; keep track of the load on the copies and redistribute that load when needed; or allow new servers to be added at any time to increase the capacity of the whole. Since these services are completely hidden from the users of Project Darkstar, they can be changed or removed, or new ones can be added at any time without changing the code of the game or virtual world.

Copyrighted material

For the programmer building a game or virtual world in the Project Darkstar environment, the visible architecture is the set of services contained in the stack. The overall set of services is both changeable and configurable, but four basic services will always be present and form the core of the operating environment, as shown in Figure 3-2.

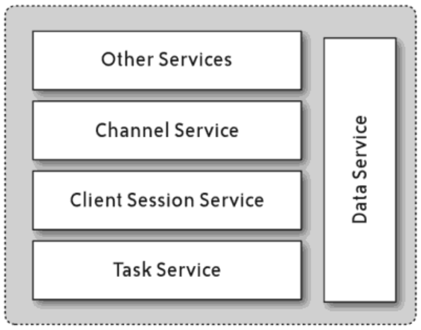


FIGURE 3-2. Darkstar stack

The Basic Services

The most basic of these stack-level services is the Data Service, which is used to store, retrieve, and manipulate all persistent data in the game or virtual world. The notion of persistence here is somewhat broader than might be found in other systems. In games or virtual worlds written in the Project Darkstar environment, any data that lasts longer than a single task is considered persistent and must be stored in the Data Service. Remember that we assume (and require) a programming model in which tasks are short-lived, so almost all of the data used to represent the server-side representation of the game or world will be persistent. The Data Service also knits together the separate copies of the game or world that are running on different servers, as all of these copies will share a single (conceptual) instance of the Data Service. All of the copies will have access to the same data, and all of the copies can read or change data stored in that service as needed.

Although the Data Store looks like a natural place for using a database, the requirements on the store are in fact very different from those that usually condition standard databases. There are very few static relations between the objects in the store, and there is no requirement within the game for any type of complex queries over the contents of the store. Instead, a simple naming scheme suffices, along with program-language-level references to the objects. The Data Store also has to be optimized for latency rather than throughput. The number of objects accessed by any particular task tends to be small (our preliminary measurements based on some prototype games and worlds suggest about a dozen objects per task), and about half of those objects that are accessed by any task are altered in the course of the task.

Copyrighted material

The second stack-level service is the Task Service, which is used to schedule and perform the tasks that are generated either in response to some event received from the clients or by the internal logic of the game or world server itself. Most tasks are one-time affairs, generated because of some action on the client, that read some data from the Data Service, manipulate that data, perhaps perform some communication, and then end. Tasks can also generate other tasks, or they can be generated as periodic tasks that will be run at particular times or intervals. All tasks must be short-lived; the maximal time for a task is a configured value, but the default is 100 milliseconds.

The game or world programmer sees a single task being generated either by an event or by the server logic itself, but under the covers the Darkstar infrastructure is scheduling as many simultaneous tasks as it can. In particular, tasks generated by the server logic will run in parallel with tasks generated in response to a client-initiated event, as will events generated in response to different clients.

Such concurrent execution leads to the possibility of data contention. To deal with such contention requires that the Task Service and the Data Service conspire. Under the covers and invisible to the server programmer, each task scheduled by the Task Service is wrapped in a transaction. This transaction ensures that either all of the operations in the task complete or none of them do. In addition, any attempts to alter values of objects held in the Data Service are mediated by that service. If more than one task attempts to alter the same data object, all but one of those tasks will be aborted and rescheduled to be performed later. The remaining task will run to completion. Once the running task has been completed, the other tasks can be run. Although it is possible for the server programmer to indicate that the data being accessed will be modified, this is not required. If a data object is simply read and then later modified, the modification will be detected by the Data Service before the task is committed. Indicating that modification is intended at the time of read is an optimization that allows early detection of conflicts, but the failure to indicate the intent to modify does not affect the correctness of a program.

Wrapping the tasks in a transaction means that the communication mechanisms must also be transactional, with messages sent only when the transaction wrapping the task that sends the messages commits. This is accomplished through the two remaining core services of the Darkstar stack.

Communication Services

The first of these is the Session Service, which mediates communication between a client and the game or world server. Upon login and authentication, a session is established between the client and the server. Servers listen for messages sent by the client on the session, parsing the contents of the message to determine what task to generate in response to the message. Clients listen on the channel to receive any responses from the server. These sessions mask the actual endpoints to both the client and the server, a factor that is important in the multimachine

Copyrighted material

scaling strategy of Darkstar. The session is also responsible for ensuring that the order of messages is maintained. A message from a given client will not be delivered if the tasks that resulted from previous message deliveries have not completed. Having the session service order tasks in this way significantly simplifies the Task Service, which can assume that all of the tasks that it has at any time are essentially concurrent. The ordering of messages from a particular client is the only message-ordering guarantee made within the Darkstar framework; external observers might see an ordering of messages from multiple clients that is very different from that seen within the game or virtual world.

The second communication service that is always available in the Darkstar stack is the Channel Service. Channels are a form of one-to-many communication. Conceptually, channels can be joined by any number of clients, and any message that is sent on the channel will be delivered to all of the clients that have been associated with the channel. This might seem to be a perfect place to utilize peer-to-peer technologies, allowing clients to directly communicate with other clients without adding any load to the server. However, these sorts of communications need to be monitored by some code that is trusted to ensure that neither inappropriate messages nor cheating can take place by utilizing different client implementations. Since the client is assumed to be under the control of the user or player, the code that is on that client cannot be trusted, because it is easy to swap out the original client code for some other, "customized" version of the client. So, in fact, all channel messages have to go through the server, after being (possibly) vetted by the server logic.

One of the complexities of both Sessions and Channels is that they must obey the transactional semantics of tasks. Thus the actual transmission of a message on either a Session link or a Channel cannot happen when the call is made to the appropriate `send()` method; it can happen only when the task in which that method occurs commits.

Supplying these communication mechanisms gives us some of the pieces that are needed for the second part of our scaling mechanism. Since all communication must go through the Darkstar Session or Channel abstractions, and since those abstractions do not reveal the actual endpoints of the communication to the client or the server, there is a layer of abstraction between the entities communicating and the actual locations that are the start and end to that communication. This means that we can move the endpoint of the server communication from one machine in the Darkstar system to another without changing the way the client views the communication. From the client's point of view, all communication happens on a particular session or channel. From the point of view of the game or virtual world logic, communication is also through a single session or channel. But the underlying infrastructure can move the session or channel from one machine to another as needed to balance load as that load changes over time.

Copyrighted material

Task Portability

The core of the ability to balance load is that, given the programming model we require and the basic stack services that must be used, tasks that are performed in response to a client-generated or game-internal event are portable from any of the machines running a copy of the game or world logic on a Darkstar stack to any other machine running such a copy. The tasks themselves are written in Java,† which means that they can be run on any of the other machines as long as those (physical) machines have the same Java Virtual Machine as part of the runtime stack. All data read and manipulated by the task must be obtained from the Data Service, which is shared by all of the instances of the game or virtual world and the Darkstar stack on all of the machines. Communication is mediated by the Session Service or by Channels, which abstract the actual endpoints of the communication and allow any particular session or channel to be moved from one server to another. Thus, any task can be run on any of the instances of the game server without changing the semantics of the task.

This makes the basic scaling mechanism of Darkstar seemingly simple. If there is a machine that is being overloaded, simply move some of the tasks from that machine to one that is less loaded. If all of the machines are being overloaded, add a new machine to the group running a copy of the game or virtual world server logic on top of a Darkstar stack, and the underlying load-balancing software will start distributing load to that new machine.

The monitoring of the load on the individual machines and the redistribution of the load when needed is the job of the meta-services. These are network-level services that are not visible to the game or virtual world programmer, but are seen by and can themselves observe the services in the Darkstar stack. These meta-services observe, for example, which machines are currently running (and if any of those machines fail), what users are associated with the tasks on a particular machine, and the current load on the different machines. Since the meta-services are not visible to the game or virtual world programmer, they can be changed at any time without having an impact on the correctness of the game logic. This allows us to experiment with different strategies and approaches to dynamically load balance the system, and allows us to enrich the set of meta-services as required by the infrastructure.

The same mechanism that we have used for scaling over multiple machines is used to obtain a high degree of fault-tolerance in the system. Given the machine-independent nature of the data that is used by a task and the communication mechanisms, it may be clear that it is possible to move a task from one machine to another. But if a machine fails, how can we recover the tasks that were on that machine? The answer is that the tasks themselves are persistent objects, stored in the Data Service for the overall system. Thus, if a machine fails, any of the tasks that were being performed by that machine will be treated as aborted transactions, and will be

† More precisely, all of the tasks consist of sequences of bytecodes that can be executed on the Java Virtual Machine. We don't care what the source-level language is; all we care about is that the compiled form of that source language can be run on any of the environments that make up the distributed set of machines running the game or virtual world.

Copyrighted material

rescheduled on different machines. Although the latency of such rescheduling may be greater than the rescheduling of an aborted transaction that stays on the same machine, the correctness of the system will be the same. At most, the user of the system (the game player or virtual world inhabitant) will notice a momentary lag in response time. Such a lag may be irritating, but it is far less extreme than the current impact of a server crash in game or virtual world environments, where the crash at least results in logging out the player, with the possibility of losing a considerable amount of game play state.

Thoughts on the Architecture

Perhaps the first question anyone asks of an architecture and its implementation is how well it performs. Although optimizing an architecture prematurely is the source of a multitude of sins, it is also possible to design an architecture that cannot be implemented in a way that performs well. Due to one of the basic choices in the Darkstar architecture, this worry is quite real. And because of the nature of the game industry, determining the performance of a server infrastructure is difficult to do.

The difficulty in determining the performance of a game or world server infrastructure is an outgrowth of the simple fact that there are no benchmarks or commonly accepted examples for a large-scale MMO or virtual world. The lack of benchmarks is not surprising, given that the server components of most games or virtual worlds are built from the ground up for a particular instance of the game or virtual world. There are only a few general infrastructures that are offered as reusable building blocks, and these are generally extracted from a particular game or world after the fact and offered to others who are building similar games. Whether it is the relative youth of the game industry or an accident of the historical emergence of the technology from the entertainment industry, no commonly accepted benchmarks are available to test a new infrastructure or to allow the comparison of different infrastructures.

There is also little or no information available concerning the expected computation, data manipulation, and communication loads for a game or virtual world server that would allow for the construction of benchmarks or performance tests. This is partly an outgrowth of the custom nature of the servers that have been produced. Each of these is built for a particular game or virtual world and thus is specialized for the particular workload characteristics of that game or world. Even more, it is an outgrowth of the intensely secretive nature of the game industry, in which any information about a game in development is jealously guarded, and information about the way in which a released game was implemented is both tightly guarded and, to many in the industry, considered uninteresting. Much more thought and discussion is given to the artwork, the storyline, or the player interaction patterns that make a new game interesting or fun than is given to the way in which the server for the game was designed or to the mechanisms used to scale the game to its current population of players (a statistic that is also closely guarded). So just getting information about the kinds of loads that current games or virtual worlds place on a server is difficult.

Copyrighted material

In our experience, even when we can get developers to talk about the loads placed on the server by their game or virtual world, they are often incorrect in their reports. This is not because they are attempting to maintain some commercial advantage by misreporting what their server actually does, but because they genuinely don't know themselves. There is very little instrumentation placed in game servers that would allow them to gather information on how the server is actually performing or what it is doing. The analysis of such servers is generally experiential at best. Programmers work on the server until it allows game play to be fun, which is achieved in an iterative manner rather than by doing careful measurements of the code itself. There is far more craft than science in these systems.

This is not to say that the servers backing such games and virtual worlds are shoddily constructed pieces of code or that they are badly built. Indeed, many of them are marvels of efficiency that demonstrate clever programming techniques and the advantages of one-time, special-purpose servers for highly demanding applications. However, the custom of building a new server for each game or world means that little knowledge of what is needed for those servers has developed, and there is no commonly accepted mechanism for comparing one infrastructure to another.

Parallelism and Latency

This lack of information about what is needed for acceptable performance in the server is of particular concern to the Darkstar team, as some of the core decisions that we have made fly in the face of the lore that has developed around how to get good performance from a game or virtual world server. Perhaps the most radical difference between the Darkstar architecture and common practice is the refusal in the Darkstar architecture to keep any significant information in the main memory of the server machine. The requirement that all data that lasts longer than a particular task be stored persistently in the Data Store is central to the functionality of the Darkstar infrastructure. It allows the infrastructure to detect concurrency problems, which in turn allows the system to hide those problems from the programmer while still allowing the server to exploit multicore architectures. It is also a key component to the overall scaling story, as it allows tasks to be moved from one machine to another to balance the load over a set of machines.

Storing the game state persistently at all times is heresy in the world of game and virtual world servers, where the worry over latency is paramount. The received wisdom when writing such servers is that only by keeping all of the information in main memory will the latency be kept small enough to allow the required response times. Snapshots of that state may be taken on occasion, but the need for interactive speeds means that such long-term operations must be done rarely and in the background. So it appears on the face of it that we have based our architecture on a premise that will keep that architecture from ever performing well enough to serve the needs of its intended audience.

Copyrighted material

Although it is certainly true that requiring data to be persistent is a major difference in the architecture, and that accessing data through the Data Store will introduce considerable latencies into the architecture, we believe that the approach we have taken will be more than competitive for a number of reasons. First, we believe that we can make the difference between accessing data in main memory and accessing it through the data store much smaller than is generally believed. Although conceptually every object that lasts longer than a single task needs to be read from and written to persistent storage, the implementation of such a store can utilize the years of research in database caching and coherence to minimize the data access latencies incurred by the approach.

This is especially true if we can localize the access to particular sets of objects on a particular server. If the only tasks that are making use of a particular set of objects are run on a single server, then the cache on that server can be used to give near main-memory access and write times for the objects (subject to whatever durability constraints need to be met). Tasks can be identified with particular players or users in the virtual world. And here we can utilize the requirement that data access and communications go through services provided by the infrastructure to gather information about the data access patterns and the communication patterns taking place in the game or world at a particular time. Given this information, we believe that we can make very accurate estimations of which players should be co-located with other players. Since we can move players to any server that we wish, we can maximize the co-location of players in an active fashion, based upon the runtime behavior that we observe. This should allow us to make use of standard caching techniques that are well-known in the database world to minimize the latencies of accessing and storing the persistent information.

This sounds very much like the geographic decomposition that is currently used in large-scale games and virtual worlds to allow scaling. There, the server developers decompose the world into areas that are assigned to servers, and the various areas act as localization devices for the players. Players in the same area are more likely to interact than those in other areas, and so co-location on a server is enhanced. The difference is that current geographic decompositions occur as part of the development of the game and are reified in the source code to the server. Our co-location is based on runtime information, and can be dynamically tuned to the actual patterns of play or interaction that are occurring at the time of placement. This is analogous to the difference between compile-time optimization and just-in-time optimization. The former seeks to optimize for all possible runs of a program, whereas the latter attempts to optimize for the current run.

We don't believe that we can make the difference between main-memory access and persistent access disappear, but we also don't think that this is necessary in order to end up with performance that is better than that of infrastructures that make use of main memory. Remember that by making all of the data persistent, we are enabling the use of multiple threads (and therefore the multiple cores) within the server. Although we don't believe that the concurrency will be perfect (that is, that for each additional core we will get complete use of that core), we do believe (and preliminary results encourage this belief) that there is a

Copyrighted material

significant amount of parallelism that can be exploited in games and virtual worlds. If the amount of concurrency that we can exploit is greater than the amount of latency that we might introduce, the overall performance of the game or virtual world will be better.

Betting on the Future

Our reliance on multithreading from multiple cores is essentially a bet on the way processors will evolve in the future. Currently servers are built with processors offering between 2 and 32 cores; we believe that the future of chip design will center around even more cores rather than on making any existing core run at a higher clock rate. When we began this project some years ago, this bet seemed far more speculative than it now appears. At that time, we often presented our designs as an exercise in “what if,” saying that we were experimenting with an architecture that would be viable if the performance of chips became more a function of the number of threads supported than the clock speed of a single thread. This is one of the advantages of doing such a project in a research lab, where it is acceptable to take a much higher risk in your approach to a design as a way of exploring an area that might turn out to be commercially viable. Current trends in chip design make the decision to build an architecture centered on multithreading look far more prescient than it appeared at the time the decision was made.[‡]

Even if we can get only 50% of perfect concurrency, we could hit a performance break-even point if we can reduce the penalty of using persistent storage to between 2 and 16 times that of main memory. We believe we can do better in both the dimension of concurrency and in the dimension of reducing the difference between accessing the persistent state and keeping everything in memory. But much will depend on the usage patterns of those building upon the infrastructure (which, as we noted earlier, are difficult to discover).

Nor should we think of minimizing latency as the only goal of the infrastructure. By keeping all the server game or world objects in the Data Store, we minimize the amount of data that would be lost in the event of a server failure. Indeed, in most cases a server failure will be noticed only as a short increase in latency as the tasks (which are themselves persistent objects) are moved from the server that failed to an alternate server; no data should be lost. Some caching schemes might result in the loss of a few seconds of play, but even this case is far better than the current schemes used by online games and virtual worlds, where occasional snapshots are the main form of persistence. In such infrastructures, hours of game play might be lost if a server crashes at just the wrong time. As long as latencies are acceptable, the greater reliability of the persistence mechanism used by Darkstar can be an advantage for both the developers of the system built on the infrastructure and the users of that system.

[‡] Showing, once again, that very little is as important as luck in the early stages of a design.

Copyrighted material

Simplifying the Programmer's Job

Indeed, if minimizing latency while allowing scale were the only goal of the server developer, that developer would be best served by writing his own distributed and multithreaded infrastructure customized for the particular game. But this would require that the server developer deal with the complexities of distributed and concurrent programming. Before getting too obsessed with the need for speed, we should remember that a second, but equally important, goal of Darkstar is to allow the production of multithreaded, distributed games while providing the programmer a model of writing on a single machine in a single thread.

To a considerable extent, we have succeeded in this goal. By wrapping all tasks in transactions and detecting data conflicts within the Data Service, programmers get the benefits of multiple threads without needing to introduce locking protocols, synchronization, or semaphores into their code. Programmers do not have to worry about how to move a player from one server to another, since Darkstar handles the load balancing transparently for them. The programming model, although stylized and restrictive, has been found by early members of the community to be natural for the kinds of games and virtual worlds that they are building.

Unfortunately, we have found that we can't hide everything from the programmer. This became apparent when the very first game to be written on top of Darkstar showed very little parallelism (and exceptionally poor performance). On examination of the source code, it did not take us long to find the explanation. The data structures in the game had been written in such a way that any change of state in the game involved a single object, which was used as a coordinator for everything. The use of this single object effectively serialized all of the actions within the game, making it impossible for the infrastructure to find or exploit any concurrency.

Once we saw this, we had a long discussion with the game developers about the need to design their objects with concurrent access in mind. An audit of the data objects in the game showed a number of similar cases where concurrency was (unintentionally) precluded by choices made in the data design. Once these objects were redesigned, the performance of the overall system increased by multiple orders of magnitude.

This taught us that it is not possible for the developers using Darkstar to be completely ignorant of the underlying concurrent and distributed nature of the system. However, their knowledge of these properties of the system need not include the usual problems of concurrency control, locking, and dealing with communication between the distributed parts of the system. Instead, they are confined to the design activity of ensuring that their data objects are defined in such a way that concurrency can be maximized. Such design usually takes the more general form of ensuring that the objects defined are self-contained and do not depend on the state of other objects for their own operations, which is not a bad design principle in any system.

Copyrighted material

There is still much about the Darkstar architecture that we have not tested or that we don't fully understand. Although we have produced a system that allows multiple machines to run a game or virtual world utilizing multiple threads in a way that is (mostly) transparent to the server programmer, we have not yet tested the ability of the architecture to add other services beyond the core. Given the transactional nature of Darkstar tasks, this may turn out to be more complex than we first imagined, and our hope is that the additional services will not need to be participants in the core service transactions. We have also just begun to experiment with various ways of gathering information about the load on the system and balancing that load. Fortunately, since the mechanisms that do this balancing are completely hidden from the programmers using the system, we can pull out old approaches and introduce new ones without affecting those using Darkstar.

As an architecture, Darkstar presents a number of novel approaches that make it interesting. It is one of the few attempts to build a game or virtual world infrastructure with the same reliability and dependability properties as enterprise software while also meeting the latency, communication, and scaling requirements of the game industry. By trying to gain efficiency by using more machines and more threads, we hope to offset the increases in latency we introduce by the use of a persistent storage mechanism. Finally, the very different world of games and virtual environments, in which the clients are thick and the servers are thin, presents a contrast to the usual environment in which highly concurrent, distributed systems are generally built. It is too early to tell whether the architecture is going to be successful, but we believe that it is already interesting.

Copyrighted material

	Principles and properties		Structures
	Versatility	✓	Module
✓	Conceptual integrity	✓	Dependency
✓	Independently changeable		Process
✓	Automatic propagation		Data access
✓	Buildability		
	Growth accommodation		
	Entropy resistance		

CHAPTER FOUR

Making Memories

Michael Nygard

SINCE THE EARLIEST TINTYPES AND DAGUERREOTYPES, we have always seen photographs as special, sometimes even magical. A photograph captures a fleeting moment in time, in a way that our fallible memories cannot. But the best portraits do more than just preserve a moment; they illuminate it. They catch a certain glance or expression, a characteristic pose that lets the subject's personality shine through.

If you've had children in a U.S. school, you probably already know the name Lifetouch. Lifetouch photographs most elementary school, middle school, and high school students in the United States every single year. What you may not know is that Lifetouch also runs high-quality portrait studios. Lifetouch Portrait Studios (LPS) operates in major retail stores across the country, along with the "Flash!" chain of studios in shopping malls. In these studios, LPS's photographers take portraits that last a lifetime.

Digital photography has transformed the entire photography industry, and LPS is no exception. Giant rolls of film and frame-mounted cameras are disappearing, replaced with professional-grade DSLRs and flash memory cards. Unfettered photographers can move around, try different angles, and get closer than ever to their subjects. In short, they have more freedom to take those great portraits. The photographer works with the camera to turn photons into electrons, but somehow, somewhere, some system has to turn those electrons into atoms of ink and paper.

Copyrighted material

In 2005, my colleagues and I from Advanced Technologies Integration (ATI) in Minneapolis worked together with developers from LPS to roll out a new system to do exactly that.

Capabilities and Constraints

Two dynamics drive a system's architecture: What must it do? What boundaries must it work within? These define the problem space.

We create, and simultaneously explore, the solution space by resolving these forces, navigating the positive pole of required behavior and the negative one of limitations. Sometimes we can create elegance, and even beauty, when the answers to individual constraints mesh together into a coherent whole. I'm happy to say that the Creation Center project did just that.

On this project, we faced several incontrovertible facts. Some are just the nature of the business; others could change, but not within our scope. Either way, we regarded these as immutable. These facts make up the left column in Figure 4-1.

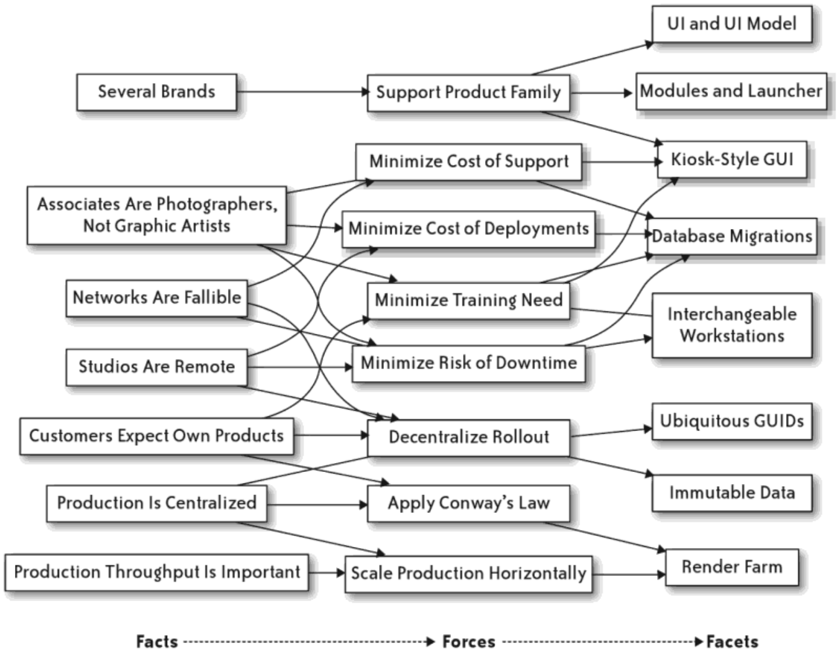


FIGURE 4-1. Facts, forces, and facets of Creation Center's architecture

Several brands

LPS supports multiple brands today and could add more in the future. At a minimum, Creation Center would have two visually distinct skins, and adding skins should not require extensive effort.

Copyrighted material

Associates are photographers, not graphic artists

Photographers are trained to use the camera, not Photoshop. When an inexperienced user sits down at Photoshop, the most likely result is a lousy image. It's a power tool for power users, and there should be no need for a photographer in a portrait studio to get up the Photoshop learning curve. Photoshop and its cousins would also slow down studio workflow. Instead, studio associates need to create beautiful images rapidly.

Studios are remote

Studios are geographically dispersed, with little to no local technical support. Hardware deliveries or replacements require shipping components back and forth.

Networks are fallible

Some studios have no network connections. Even for the ones that do, it's not acceptable to halt the studio if the connection goes down.

Customers expect their own products

Customers should receive their photos with their designs and text.

Production is centralized

High-quality photographic printers are becoming more common, but making products that can last for decades requires much more expensive equipment.

Production throughput is important

The same printers are also the constraint in the production process. Therefore, every other step in the process must be subordinated to the constraint.

These facts lead to several forces that we must balance. It's common to perceive the forces as fundamental, but they aren't. Instead, they emerge from the context in which the system exists. If the context changes, then the forces might be nullified or even negated.

We chose a handful of constructs to resolve these forces. The rightmost column of Figure 4-1 shows these facets of the architecture. Of course, these aren't the only Creation Center features worth discussing, but these facets of the architecture are of general interest. I also think they simultaneously illustrate a nice separation of concerns and mutually supporting structures.

Before digging into the specific features, we need to fill in one more piece of context: the system's workflow.

Workflow

The typical studio has two to four camera rooms, stocked with professional lighting, backdrops, and props. The photographers take pictures—each picture is called a "pose"—in the camera room. Outside of the camera room, photographers also handle customer service, scheduling, and customer pickups.

When the photographer finishes taking the pictures for a session, she sits down at any of several workstations to load the photographs from the camera's memory card.

Copyrighted material

After loading a session, the photographer deletes any obviously bad photographs: ones with closed eyes, sour expressions, babies looking away, and so on. After deleting the bad ones, the rest become "base images." She then creates a number of enhancements from those base images. Enhancements range from simple tonal applications, such as black and white or sepia, to elaborate compositions of multiple photos. For example, a photographer might take a group portrait of three children and embed it in a design with three "slots" for individual portraits of the children.

After creating these enhancements, the photographer helps the customer order various sizes and combinations of prints. These include everything from 8″ × 10″ portraits to "sheets" of smaller sizes: 5″ × 7″, 3″ × 5″, or wallet sizes. Then there are the large formats. Customers can order portraits in sizes up to 24″ × 30″, made for framing and hanging on the wall.

After completing the customer's order, the photographer moves on to the next session.

At the end of each day, the studio manager creates a DVD of the day's orders, which she sends to the printing facility.

In the printing facility, hundreds of DVDs arrive each day. (I'll talk about the contents of the DVDs later.) The DVDs contain orders and photographs that need to be printed and shipped back to the studio, so the customer can pick them up. Before they can be printed, however, the final print-resolution photographs must be rendered as images. These print-ready images are immense. A 24″ × 30″ portrait rendered for high-quality printing, has over 100 million pixels, each in 32-bit color. Every single pixel is composited according to the design the photographer created in the studio. Depending on the composition, the rendering pipeline can be anywhere from 6 to 10 steps long. A simple rendering takes two to five minutes, but complex compositions for large formats churn for ten minutes or more.

At the same time, the printers spit out several finished prints per minute. Keeping the printers busy is the duty of the Production Control System (PCS), a complex system that handles job scheduling and orchestrates the render farm, manages image storage, and feeds the print queues.

When the finished order reaches the studio, the manager lets the customer know that she can come in to pick it up.

This workflow partly came from LPS's business context and partly from our choices about how to partition the system. Now let's look at the different facets from Figure 4-1.

Architecture Facets

Reducing the structure of a multidimensional, dynamic system into a linear narrative form is always a challenge, whether we are communicating our vision of a system that doesn't exist or trying to explain the interacting parts of one that we've already built. Hypertext might make

Copyrighted material

it easier to approach the elephant from several perspectives, but paper doesn't yet support hyperlinks very well.

As we look at each of these facets, keep in mind that they are different ways of looking at the overall system. For instance, we used a modular architecture to support different deployment scenarios. At the same time, each module is built in a layered architecture. These are orthogonal but intersecting concerns. Each set of modules follows the same layering, and each layer is found across all the modules.

Indeed, we all felt deeply gratified that we were able to keep these concerns separated while still making them mutually supportive.

Modules and Launcher

All along, we were thinking "product family" rather than "application" because we had to support several different deployment scenarios with the same underlying code. In particular, we knew from the beginning that we would have the following configurations:

Studio Client
: A studio has between two and four of these workstations. The photographers use them for the entire workflow, from loading images through to creating the orders.

Studio Server
: The central server inside each studio runs MySQL for structured data such as customers and orders. The server also has much more robust storage than the workstations, using RAID for resiliency. The studio server also burns the day's orders to DVD.

Render Engine
: Once in production, we decided to build our own render engine. By using the same code for rendering to the screen in the studio and to the print-ready images in production, we could be absolutely certain that the customer would get what they expected.

At first, we thought these different deployment configurations would just be different collections of *.jar* files. We created a handful of top-level directories to hold the code for each deployment, plus one "Common" folder. Each top-level folder has its own *source*, *test*, and *bin* directories.

It didn't take long for us to become frustrated with this structure. For one thing, we had one giant */lib* directory that started to accumulate a mixture of build-time and runtime libraries. We also struggled with where to put noncode assets, such as images, color profiles, Hibernate configurations, test images, and so on. Several of us also felt a nagging itch over the fact that we had to manage *.jar* file dependencies by hand. In those early days, it was common to find entire packages in the wrong directory. At runtime, though, some class would fail to load because it depended on classes packaged into a different *.jar* file.

Copyrighted material

The breaking point came when we introduced Spring[*] about three iterations into the project. We were following an "agile architecture" approach: keep it minimal and commit to new architecture features only when the cost of avoiding them exceeds the cost of implementing them. That's what Lean Software Development calls "the last responsible moment." Early on, we had only a casual knowledge of Spring, so we chose not to depend on it, though we all expected to need it later.

When we added Spring, the *.jar* file dependency problems were multiplied by configuration file problems. Each deployment configuration needs its own *beans.xml* file, but well over half of the beans would be duplicated between files—a clear violation of the "don't repeat yourself" principle[†]—and a sure-fire source of defects. Nobody should have to manually synchronize bean definitions in thousand-line XML files. And, besides, isn't a multi-thousand-line XML file a code smell in its own right?

We needed a solution that would let us modularize Spring beans files, manage *.jar* file dependencies, keep libraries close to the code that uses them, and manage the classpath at build time and at runtime.

ApplicationContext

Learning Spring is like exploring a vast, unfamiliar territory. It's the NetHack of frameworks; they thought of *everything*. Wandering through the javadoc often yields great rewards, and in this case we hit pay dirt when I stumbled across the "application context" class.

The heart of any Spring application is a "bean factory." A bean factory allows objects to be looked up by name, creates them as needed, and injects configurations and references to other beans. In short, it manages Java objects and their configurations. The most commonly used bean factory implementation reads XML files.

An application context extends the bean factory with the crucial ability to make a chain of nested contexts, as in the "Chain of Responsibility" pattern from *Design Patterns* (Gamma et al. 1994).

The `ApplicationContext` object gave us exactly what we needed: a way to break up our beans into multiple files, loading each file into its own application context.

Then we needed a way to set up a chain of application contexts, preferably without using some giant shell script.

[*] *http://www.springframework.org/*

[†] See *The Pragmatic Programmer* by Andrew Hunt and David Thomas (Addison-Wesley Professional).

Copyrighted material

Module dependencies

Thinking of each top-level directory as a module, I thought it would be natural to have each module contain its own metadata. That way the module could just declare the classpath and configuration files it contributes, along with a declaration of which other modules it needs.

I gave each module its own manifest file. For example, here is the manifest file for the `StudioClient` module:

```
Required-Components: Common StudioCommon
Class-Path: bin/classes/ lib/StudioClient.jar
Spring-Config: config/beans.xml config/screens.xml config/forms.xml
        config/navigation.xml
Purpose: Selling station. Workflow. User Interface. Load images. Burn DVDs.
```

This format clearly derives from *.jar* file manifests. I found it useful to align the mental function "manifest file" with a familiar format.

Notice that this module uses four separate bean files. Separating the bean definitions by function was an added bonus. It reduced churn and contention on the main configuration files, and it provided a nice separation of concerns.

Our team strongly favored automatic documentation, so we built several reporting steps into the build process. With all the module dependencies explicitly written in the manifest files, it was trivial to add a reporting step to our automated build. Just a bit of text parsing and a quick feed to Graphviz generated the dependency diagram in Figure 4-2.

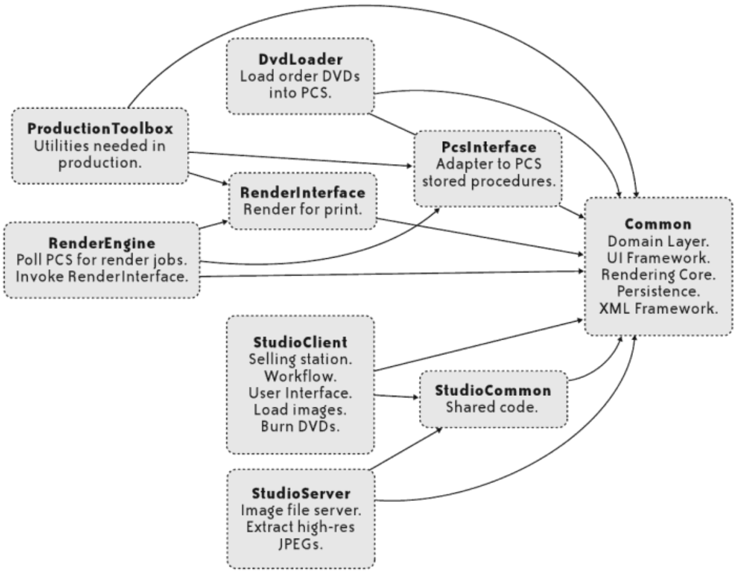

DvdLoader
Load order DVDs into PCS.
ProductionToolbox
Utilities needed in production.
PcsInterface
Adapter to PCS stored procedures.
RenderInterface
Render for print.
RenderEngine
Poll PCS for render jobs. Invoke RenderInterface.
Common
Domain Layer. UI Framework. Rendering Core. Persistence. XML Framework.
StudioClient
Selling station. Workflow. User Interface. Load images. Burn DVDs.
StudioCommon
Shared code.
StudioServer
Image file server. Extract high-res JPEGs.

FIGURE 4-2. Modules and dependencies

Copyrighted material

With these manifest files, we just needed a way to parse them and do something useful. I wrote a launcher program, imaginatively called "Launcher," to do just that.

Launcher

I've seen many desktop Java applications that come with huge shell or batch scripts to locate the JRE, set up environment variables, build the classpath, and so on. Ugh.

Given a module name, Launcher parses the manifest files, building the transitive closure of that module's dependencies. Launcher is careful not to add a module twice, and it resolves the set of partial orderings into a complete ordering. Figure 4-3 shows the fully resolved dependencies for `StudioClient`. `StudioClient` declares both `StudioCommon` and `Common` as dependencies, but Launcher gives it only one copy of each.

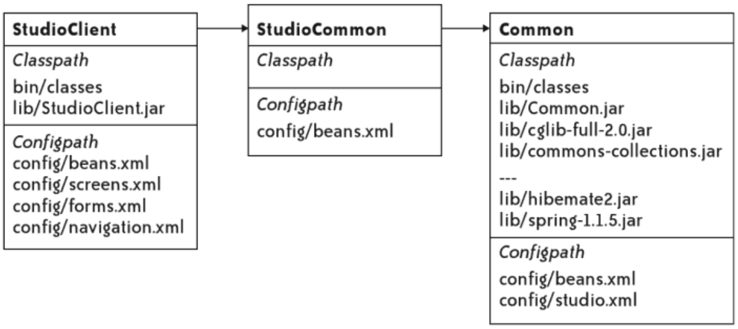


FIGURE 4-3. Resolved dependencies for StudioClient

To avoid classpath "pollution" from the host environment—ANT on a build box, or the JRE classpath on a workstation—Launcher builds its own class loader from the combined classpaths. All application classes get loaded inside that class loader, so Launcher uses that class loader to instantiate an initializer. Launcher passes the configuration path into the initializer, which creates all the application context objects. Once the application contexts are constructed, we're up and running.

Throughout the project, we refactored the module structure several times. The manifest files and Launcher held up with only minor changes throughout. We eventually arrived at six very different deployment configurations, all supported by the same structure.

The modules all share a similar structure, but they don't have to be identical. That was one of the side benefits of this approach. Each module can squirrel away stuff that other modules don't care about.

Copyrighted material

WHAT ABOUT OSGI?

When we started this project in late 2004, the OSGi framework was just beginning to gain broader visibility—thanks largely to Eclipse's adoption of it. We looked at it briefly, but were put off by the lack of widely available knowledge, expertise, and guidance.

OSGi's purpose, though, is a perfect fit for the problems we faced. Supporting multiple deployment configurations with a common codebase, managing the dependencies among modules, activating them in the correct sequence...clearly solving the same problem.

I suppose the fact that we didn't use OSGi was partly a quirk of timing and partly our own reluctance to take on what we perceived as more technical risk. I usually come down on the side of "acquire and integrate" rather than "roll your own," but there seems to be a tipping point: lightly supported open source projects with weak communities are more of a risk than well-understood, widely adopted ones. Likewise, I tend to avoid quasi-open frameworks that are actually vendor consortia. The community they serve is usually the community of vendors, not the community of users.

It wasn't clear to us which camp OSGi would fall into. If we were doing the project today, I think we probably would use OSGi instead of rolling our own.

Kiosk-Style GUI

Studio associates are hired for their ability to work well with the camera and the families, especially children, not for their computer skills. At home, they might be Photoshop gurus, but in the studio, nobody expects them to become power users. In fact, during the busy season, a studio might bring on a number of seasonal associates. Consequently, fast ramp-up is critical.

One of the architects also served as our UI designer. He always had a clear vision of the interface, even if we didn't always agree on how much was feasible to implement. He wanted the user interface to be friendly and visible. There would be no menus. Users would interact with images through direct manipulation. Large, candy-coated buttons made all options visible. In short, the workstation should look like a kiosk.

That left the decision about what technology to use for the display itself.

One of our team made a survey of the Java rich UI technologies available, mainstream and fringe. We hoped to find a good declarative UI framework, something to help us avoid an endless slog through Swing tweaks. The results shocked us all.

In 2005, even after a decade of Java, two basic choices dominated the mainstream: XML hell or GUI builder spaghetti. The XML variants map more or less directly from Swing components to XML entities and attributes. This made no sense to us. GUI changes require a code release, whether the changes are implemented in straight Java code or in XML files. Why keep two

Copyrighted material

languages in your head—Java plus the XML schema—instead of just Java? Besides, XML makes a clumsy programming language.

GUI builders had burned all of us before. Nobody wanted to end up with business logic woven into action listeners embedded in JPanels.

Reluctantly, we settled on a pure Swing GUI, but with some ground rules. Over a series of lunches at our local Applebee's, we hashed out a novel way of using Swing without getting mired in it.

UI and UI Model

The typical layered architecture goes "Presentation," "Domain," and "Persistence." In practice, the balance of code ends up in the presentation layer, the domain layer turns into anemic data containers, and the persistence layer devolves to calls into a framework.

At the same time, though, some important information gets duplicated up and down the layers. For instance, the maximum length of a last name will show up as a column width in the database, possibly a validation rule in the domain, and as a property setting on a `JTextField` in the UI.

At the same time, the presentation embeds logic such as "if *this* checkbox is selected, then enable *these* four other text fields." It sounds like a statement about the UI, but it really captures a bit of business logic: when the customer is a member of the Portrait Club, the application needs to capture their club number and expiration date.

So within the typical three-layer architecture, one type of information is spread out across layers, whereas another type of important information is stuck inside GUI control logic.

Ultimately, the answer is to invert the GUI's normal relationship to the domain layer. We put the domain in charge by separating the visual appearance of a screen from the logical manipulation of its values and properties.

Forms

In this model, a form object presents one or more domain objects' attributes as typed properties. The form manages the domain objects' lifecycles as well as calling down to the facades for transactions and persistence. Each form represents a complete screen full of interacting objects, though there are some limited cases where we use subforms.

The trick, though, is that a form is *completely* nonvisual. It doesn't deal with UI widgetry, only with objects, properties, and interactions among those properties. The UI can bind a Boolean property to any kind of UI representation and control gesture: checkbox, toggle button, text entry, or toggle switch. The form doesn't care. All it knows is that it has a property that can take a true/false value.

Copyrighted material

Forms never directly call screens. In fact, most of them don't even know the concrete class of their screens. All communication between forms and screens happens via properties and bindings.

Properties

Unlike typical form-based applications, the properties that a `Form` exposes are not just Java primitives or basic types like `java.lang.Integer`. Instead, a `Property` contains a value together with metadata about the value. A `Property` can answer whether it is single-valued or multivalued, whether it allows null values, and whether it is enabled. It also allows listeners to register for changes.

The combination of `Forms` and their `Property` objects gave us a clean model of the user interface without yet dealing with the actual GUI widgetry. We called this layer the "UI Model" layer, as shown in Figure 4-4.

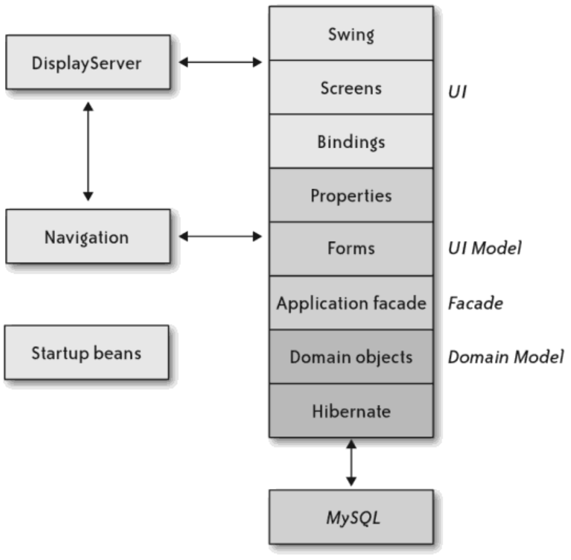


FIGURE 4-4. Layered architecture

Each subclass of `Property` works for a different type of value. Concrete subclasses have their own methods for accessing the value. For instance, `StringProperty` has `getStringValue()` and `setStringValue(String)`. Property values are always object types, not Java primitives, because primitives do not allow null values.

It might seem that property classes could proliferate endlessly. They certainly would if we created a property class for each domain object class. Most of the time, instead of exposing the domain object directly, the `Form` would expose multiple properties representing different

Copyrighted material

aspects of the domain object. For example, the customer form exposes `StringProperty` objects for the customer's first name, last name, street address, city, and zip code. It exposes a `DateProperty` for the customer's club membership expiration date.

Some domain objects would be awkward to expose this way. Connecting a slider that controls dilation of the image or embedded image in a design to the underlying geometry would have required more than half a dozen properties. Having the `Form` juggle this many properties just to drag a slider seemed like a pretty clear code smell. On the other hand, adding another type of property seemed like the path to wild type proliferation.

Instead, we compromised and introduced an object property to hold arbitrary Java objects. The animated discussion before that class appeared included the phrases "slippery slope" and "dumping ground." Fortunately, we kept that impulse in check—one of the perils of a type-checked language, I suppose.

We handled actions by creating a "command property," which encapsulates command objects but also indicates enablement. Therefore, we can bind command property objects to GUI buttons, using changes in the property's enablement to enable or disable the button.

The UI Model allowed us to keep Swing contained within the UI layer itself. It also provided huge benefits in unit testing. Our unit tests could drive the UI Model through its properties and make assertions about the property changes resulting from those actions.

So, forms are not visual themselves, but they expose named, strongly typed properties. Somewhere, those properties must get connected to visible controls. That's the job of the bindings layer.

Bindings

Whereas properties are specific to the types of their values, bindings are specific to individual Swing components. Screens create their own components, and then register bindings to connect those components to the properties of the underlying `Form` objects. An individual screen does not know the concrete type of form it works with, any more than a form knows the concrete type of the screen that attaches to it.

Most of our bindings would update their properties on every GUI change. Text fields would update on each keystroke, for instance. We used that for on-the-fly validation to provide constant, subtle feedback, rather than letting the user enter a bunch of bad data and then yelling at them with a dialog box.

Bindings also handle conversion from the property's object type to a sensible visual representation for their widgets. So, the text field binding knows how to convert integers, Booleans, and dates into text (and back again). Not every binding can handle every value type, though. There's no sensible conversion from an image property to a text field, for example. We made sure that any mismatch would be caught at application startup time.

Copyrighted material

An interesting wrinkle developed after we had built the first iteration of this property-binding framework. The first screen we tried it out on was the customer registration form. Customer registration is fairly straightforward, just a bunch of text fields, one checkbox, and a few buttons. The second screen, the album screen, is much more visual and interactive. It uses numerous GUI widgets: two proof sheets, a large image editor, a slider, and several command buttons. Even here, the form makes all the real decisions about selections, visibility, and enablement entirely through its properties. So the album form knows that the proof sheets' selections affect the central image editor, but the screen is oblivious. Keeping the screens "dumb" helped us eliminate GUI synchronization bugs and enabled much stronger unit testing.

IS ONE ENOUGH?

On some screens, proof sheets allow multiple selections; on others, only single selection. Worse yet, some actions are allowed only when exactly one thumbnail is selected. What component would decide which selection model to apply or when to enable other commands based on the selection? That's clearly logic about the UI, so it belongs in the UI Model layer. That is, it belongs in a form. The UI Model should never import a Swing class, so how can forms express their intentions about selection models without getting tangled up in Swing code?

We decided that there was no reason to restrict a GUI component to just one binding. In other words, we could make bindings that were specific to an aspect of the component, and those bindings could attach to different form properties.

For instance, we often had separate bindings to represent the content of a widget versus its selection state. The selection bindings would configure the widget for single- or multiselect, depending on the cardinality of its bound property.

Although it takes a long time to explain the property-binding architecture, I still regard it as one of the most elegant parts of Creation Center. By its nature, Creation Center is a highly visual application with rich user interaction. It's all about creating and manipulating photographs, so this is no gray, forms-based business application! Yet, from a small set of straightforward objects, each defined by a single behavior, we composed a very dynamic interface.

The client application eventually supported drag-and-drop, subselections inside an image, on-the-fly resizing, master-detail lists, tables, and double-click activation. And we never had to break out of the property-binding architecture.

Copyrighted material

Application facade

There's a classic pitfall in building a strong domain model. The presentation layer—or in this case, the UI Model—often gets too intimate with the domain model. If the presentation traverses relationships in the domain, then it becomes difficult to change the domain model. Like any agile team, we needed to stay flexible, and there was no way we would make design choices that would lead to less flexibility over time.

Martin Fowler's "Application Facade" pattern fit the bill (see the "References" section at the end of this chapter). An application facade presents only a portion of the domain model to the presentation layer. Instead of walking through graphs of domain objects, the presentation asks the application facade to assist with traversal, life cycle, activation, and so on.

Each form defined a corresponding facade interface. In fact, following the dictum that consumers—rather than their providers—should define interfaces we put the facade interface in the form's package. The form asks the facade to look up domain objects, relate them, and persist them. In fact, the facades managed all database transactions, so the forms were never aware of transaction boundaries.

The interfaces at this boundary, between forms and facades, also became an ideal place to isolate objects for unit testing. To test a particular form, the unit test creates a mock object that implements the facade's interface. The test trains the mock object to feed the form with some set of expected results, including error conditions that would be very difficult to reproduce with the real facade. I think we all regarded mock objects as a two-sided compromise: although they made unit tests possible, something still felt wrong about tying the tests so closely to the forms' implementations. For example, mock objects have to be trained with the exact sequence of method calls to expect, and the exact parameters. (Newer mock object frameworks are more flexible.) As a result, changes in the internal structure of the forms would cause tests to fail, even though no externally visible behavior changed. To a certain extent, this is just the price you pay for using mock objects.

All the Creation Center applications, both in the studio and in the printing facility, used the same stack of layers. Removing the GUI from the driver's seat kept the team from spending endless cycles in Swing tweaking. This inversion of control also provided a uniform structure that every application, and every pair, could follow. Even though we created more than the usual "three-layer cake," our stack was quite effective at separating concerns: Swing was limited to the UI, domain interaction in the forms, and persistence in the facades.

Interchangeable Workstations

When a photographer finishes a session, she grabs any open workstation. Depending on how busy the studio is, she'll usually finish with the customer at that time. It's common, though, for customers to come back later, maybe even on a different day. It would be ridiculous to permanently attach a customer to a single workstation—not just unworkable for scheduling, but also risky. Workstations break!

Copyrighted material

So any workstation in the studio must be interchangeable, but "interchangeable" presents some problems. The images for a single session can consume close to a gigabyte.

We briefly contemplated building the workstations as a peer-to-peer network with distributed replication. Ultimately, we opted for a more traditional client-server model, as shown in Figure 4-5.

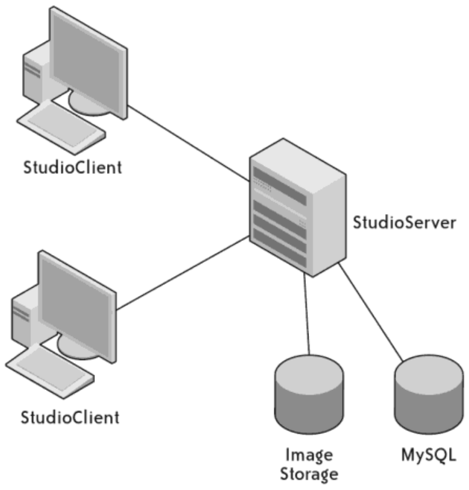


FIGURE 4-5. Studio deployment

The server is equipped with larger disks than the clients, and they are RAIDed for resilience. The server runs a MySQL database to hold structured data about customers, sessions, and orders. Most of the space, however, is devoted to storing the customers' photographs.

Because the studios are remote and the associates are not technically adept, we knew it would be important to make the "plumbing" invisible. Associates should never have to look at filesystems, investigate failures, or restart jobs. They should certainly never log into the database server! At worst, if a network cable should be bumped loose, once it is plugged back in, everything should work as normal and also should automatically recover from that temporary problem.

With that end in mind, we approached the system and application architecture.

Image repositories

To make the workstations interchangeable, the most essential feature would be automatic transfer of images, both from the workstation where the photographer loaded them to the server and from the server to another workstation.

The studio client and studio server both use a central component called an image repository. It deals with all aspects of storing, loading, and recording images, including their metadata. On

Copyrighted material

the client side, we built a local, caching, write-behind proxy. When a caller asks for an image, this client image repository either returns it directly from local cache or downloads the file into local cache, and then returns it. Either way, callers remain blissfully ignorant.

Likewise, when adding images on the client, the client image repository uploads it to the server. We use a pool of threads to run background transfers so the user doesn't have to wait on uploads.

Both the client and server repositories are heavily multithreaded. We created a system of locking called "reservations." Reservations are a soft form of collaborative locking. When a client wants to add an image to the repository, it must first request and hold a "write reservation." This way, we can be sure that no other thread is reading the image file when we issue the reservation. Readers have to acquire a "read reservation," naturally.

Although we did not implement distributed transactions or two-phase commit, in practice there is only a small window between when the client image repository grants a write reservation and when the server side grants a corresponding write reservation. When that second reservation is granted, we can be confident that we will avoid file corruption.

In practice, even lock contention is rare. It requires two photographers at two different workstations to access exactly the same customer's session. Still, there are several workstations in every studio, and each workstation has many threads, so it pays to be careful.

NIO image transfer

Obviously, that leaves the problem of getting the images from the client to the server. One option we considered and rejected early was CIFS—Windows shared drives. Our main concern here was fault-tolerance, but transfer speed also worried us. These machines needed to move a lot of data back and forth, while photographers and customers were sitting around waiting.

In our matrix of off-the-shelf options, nothing had the right mix of speed, parallelism, fault-tolerance, and information hiding. Reluctantly, we decided to build our own file transfer protocol, which led us into one of the most complex areas of Creation Center. Image transfer became a severe trial, but we emerged, at last, with one of the most robust features of the whole system.

I had some prior experience with Java NIO, so I knew we could use it to build a blazing-fast image transfer mechanism. Building the NIO data transfer itself wasn't particularly difficult. We used the common leader-follower pattern to provide concurrency while still keeping NIO selector operations on a single thread.

Although the protocol wasn't difficult to implement, there were a number of nuances to deal with:

- Either end can close a socket, particularly if the client crashes. Sample code never deals with this properly.

Copyrighted material

- While handling an IO event, the `SelectionKey` will still signal that it's ready. This can result in multiple threads calling into the same handler if you don't clear that operation from the key's interest set.
- The leader must perform all changes to a `SelectionKey`'s interest set or else you get race conditions with the `Selector`, so we had to build a queue of pending `SelectionKey` changes that the leader thread would execute before calling select.

Handling these tricky details led to quite a bit more coupling between the various objects than I initially expected. If we had been building a framework, this whole area would have needed much more attention to loose coupling. For an application, however, we felt it was acceptable to regard the collection of collaborating objects in the server as a cohesive unit.

One particularly interesting effect showed up only when we ran a packet sniffer to see if we were really getting the maximum possible throughput. We weren't. At first, when the reactor read from a socket that had data available, it would read one buffer full and then return. We figured that it wouldn't take very long to get back around the loop if more than 8,192 bytes were available. It turns out that the studio network is fast enough to fill the server's TCP window before the next thread could get back into the handler, so virtually every transfer would stall for about half of the total transfer time. We added a loop inside the reactor, so it would keep reading until the buffer was drained. That cut the transfer time by nearly half, and reduced the amount of overhead in threading and dispatching. I found this particularly interesting because it works only for fast networks with low latency and only if the total number of clients is small. With higher network latency or more clients, looping that way would risk starving some clients. Again, it was a trade-off that made sense in our context.

UNIT TESTING AND CODE REVIEW

This NIO file server was the one time that I found it helpful to do a large group review, even on an agile project with complete pairing.

My pair and I worked on the threading, locking, and NIO mechanisms over most of an iteration. We unit tested what we could, but between the threading and low-level socket IO, we found it difficult to gain confidence in the code. So we did the next best thing: we got more eyes on it. I'd call that a special case, though. We were compensating for our inability to write sufficient unit tests.

In general, having two sets of eyes on the code all the time provides all the benefits of a code review. Combine that with automatic formatting and style checking, and there's just not enough remaining advantage of a code review to offset its cost. And if you can get the benefits without the cost, then why bother with the code review?

We kept a projector in our lab, connected to two machines through an A/B switch. Whenever we had a technique to illustrate or a design pattern to share, we'd take a few minutes after lunch to fire up the projector and walk through some code. This was particularly handy during the early stages, when

Copyrighted material

the architecture and design were more fluid, and we were learning how to deal with Spring and Hibernate. It helped homogenize Eclipse practices and tricks, too.

The projector was also handy for iteration demos. We could have all the stakeholders in the room, without crowding around a single screen.

(Not to mention how helpful it was for projecting funny YouTube clips up on the wall.)

I knew it wouldn't be hard at all to build something fast but fragile. The real challenge would be making it robust, especially when the whole network would exist in a studio hundreds of miles away. One with no ability to log in remotely to debug problems or clean up after failures. One with small children, distracted parents, and servers sitting at toddlers' eye level. Talk about a hostile environment! Moving bits across the wire would not be enough; we needed atomic file transfer with guaranteed delivery.

The first layer of defense was the protocol itself. For a "put" operation—uploading a file from client to server—the first packet of the request includes the file's MD5 checksum. Once the client sends the last packet, it waits for a response from the server. The server responds with one of several codes: `OK`, `TIMEOUT`, `FAILED_CHECKSUM`, or `UNKNOWN_ERROR`. On anything but an `OK`, the client resends the entire file in what we call a "fast retry." The client gets three fast retries before the transfer fails.

Problems with file transfer will come in two varieties. One type is the "fast transient," a quick problem that will clear itself up, such as network errors. The other type requires human intervention. That means problems will either be cleared up in a few milliseconds, or they will take minutes to hours to correct. There's no point in retrying a fast file transfer over and over again. If it didn't work after the first few attempts, it's not likely to work for quite a while.

Therefore, if the client exhausts all the fast retries, it puts the file transfer job in a queue. A background job wakes up every 20 minutes looking for pending file transfer jobs. It tries each job again, and if it fails again, it goes right back into the queue. Using Spring's scheduling support made this "slow retry" almost trivial to implement.

This mix of fast and slow retries lets us decouple maintenance and support on the server from the clients. There's no need to "cold boot" an entire studio for upgrades or replacements.

Fast and robust

The local and remote image repository and their associated file transfer mechanics became a seriously tough slog. Once it was done, though, the whole thing could upload images to the server faster than they could be read from the memory card. Downloading them on another machine was fast enough that users never perceived any activity at all. The client would download all the thumbnails for an album during the transition from one screen to the next. Downloading the screen-sized images for full-size display could be done during a mouse click. This speed let us avoid the user frustration of "loading" dialogs.

Copyrighted material

Database Migrations

Imagine operating 600 remote database servers across four time zones. They might as well be on a desert island, and digitally speaking, they are. If a database administrator needed to apply changes by hand, he would have to travel to hundreds of locations.

In such circumstances, one option would be to get the database design exactly right before the first release, and then never change it again. There may still be a few people who think that's possible, but certainly none of them were on my team. We expected and even counted on change at every level, including the database.

Another option would be to send release notes out to the field. The studio managers always called the service desk for a verbal walkthrough when they executed the installs. Perhaps we could include SQL scripts in documents on the release CDs for them to type in or copy-and-paste. The prospect of dictating any command that starts with, "Now type `mysqladmin -u root -p`..." gives me cold sweats.

Instead, we decided to automate database updates. Ruby on Rails calls these "database migrations," but in 2005 it wasn't a common technique.

Updates as objects

The studio server defines a bean called a database updater. It keeps a list of database update objects, each representing an atomic change to the database. Each database update knows its own version and how to apply itself to the database.

At startup time, the database updater checks a table for the current version of the database. If it doesn't find the table, it assumes that no updates exist or have been applied. Accordingly, the very first update bootstraps the version table and populates it with one row. That single row contains a version number and a lock field. To avoid concurrent updates, the database updater first updates this row to set the lock field. If it cannot, then it assumes some other machine on the network is already applying updates.

We used this migration ability to apply some simple changes and some sophisticated ones. One of the simple ones just added indexes to a couple of columns that were affecting performance. One of the updates that made us really nervous changed all the table types from MyISAM to InnoDB. (MyISAM, the default MySQL table type, does not support transactions or referential integrity. InnoDB does. If we had known that before our first release, we could have just used InnoDB in the first place.) Given that we had deployed databases with production data, we had to use a sequence of "alter table" statements. It worked beautifully.

After a few releases had gone out to the field, we had about 10 updates. None of them failed.

Regular exercise

Every time we run a build, we reset the local development database to version zero and roll forward. That means we exercise the update mechanism dozens of times every day.

Copyrighted material

We also unit test every database update. Each test case makes some assertions about the state of the database prior to the update. It applies the update and then makes some assertions about the resulting state.

Still, these tests all work with "well-behaved" data. Weird things happen out in the field, though, and real data is always messier than any test data set. Our updates create tables, add indices, populate rows, and create new columns. Some of these changes can break badly if the data isn't what we expect. We worried about the risky time during the updates and looked for ways to make the process more resilient.

Safety features

Suppose something goes wrong with one of the updates. A studio could be shut down until Operations found a way to restore the database, and if the update really goes wrong, it might leave the database corrupted or in some intermediate state. Then the studio wouldn't even be able to roll back to the previous version of the application. To avoid that disaster scenario, the database updater makes a backup copy of the database before it starts applying the updates. If it can't make the backup copy, then it halts the update process.

If errors occur during the updates, the updater automatically attempts to reload from that backup copy. If even that step fails, well, at least there's a copy onsite so a support technician can talk the studio manager through a manual restore.

In fact, in the absolute worst case, the printing facility always has a copy of the database that's no more than one day old. We used some of the extra space on the daily DVD to send a complete copy of the database every day. There's something to be said for a small database and a lot of storage space.

Field results

The time we invested in automated database updates paid off in several ways. First, we improved performance and reliability through some early updates. Feedback from the user community was immediate and positive after that release. Second, the operations group greatly appreciated the easy deployment of new releases. Previous systems had required the studios to ship removable hard drives back and forth, with all the attendant logistics problems. Finally, having the update mechanism allowed us to focus on "just sufficient" database design. We did not peer into the crystal ball or overengineer the database schema. Instead, we just designed enough of the schema to support the current iteration.

Immutable Data and Ubiquitous GUIDs

In working with customers, the studio associate creates some compositions that use multiple photographs, inset into a design. These designs come from a design group at company headquarters. Some designs are perennial, others are seasonal. Christmas cards in a wide

Copyrighted material

variety of designs are a big seller, at least in the weeks before Christmas. Not surprisingly, demand drops precipitously after that.

A particular design includes some imagery for the background and a description of how many openings there are for base images, and the geometry of those openings. The associate can be very creative in filling those openings with photographs and with other compositions.

We found some interesting challenges dealing with these designs and the base images that go in them. For instance, what happens when a customer places an order, but then a new version of the design gets rolled out to the studio? At a smaller scale, what do you do if the associate nested one design within another—such as a sepia-tinted photograph inside a border—and then changes or deletes the original design?

At first, this looked like a nightmare of reference counting and hidden linkages. Every scheme we considered created a web of object references that could lead to gaps, missing images, or surprising changes. As a team, we all believed in "The Rule of Least Surprise," so hidden linkages causing changes to ripple from one product to another just wasn't going to work.

When our lead visionary came up with a simple, clear answer, it didn't take more than 30 seconds to sell the rest of us on it. The solution incorporated two rules:

1. Don't change anything after creating it. Designs and compositions would be immutable.
2. Copy, don't reference, the original.

Taken together, this means that selecting a design actually copies that design into the working space. If the associate adds the resulting composition to the album, it's actually a complete and self-contained copy of the design that gets added. Likewise, nesting one enhanced image into another makes a copy of the original and grafts it into the new composition. From the moment that graft happens, the original composition and the new one are completely independent of each other.

These copies are not just a trick of object references in memory. The actual XML description of the composition contains a complete copy of the design or the embedded compositions. This description lives in the studio's database, and it's the same description that gets sent on the DVD. When the studio manager burns the day's orders to DVD, the StudioServer packs in everything needed to create the final render: source images, backgrounds, alpha masks, and the instructions about how to combine them into the final image.

Having the complete description of the whole composition—including the design itself—on DVD became a huge advantage for production.

Previous systems kept the designs in a library, and orders just referenced them by ID. That meant the designers had to coordinate design IDs between the studios and the centralized printing facility. Therefore, designs had to be "registered" in production before they could be rolled out to the field. Should the IDs get out of sync, as sometimes happened, the wrong design would be produced and customers would not get the products they expected. Likewise,

Copyrighted material

whenever the designers updated a design, there would be a few days' worth of DVDs in the pipeline made with the old version of the design. Sometimes it would come out OK, and sometimes it wouldn't.

Under the new system, designs never have to be registered. Whatever comes through in the XML is what gets produced, which frees the designers to make much more frequent changes and roll them out however they want. New revisions of designs don't affect orders in the pipeline, because each order is self-contained. Once the new revision gets out to the studios, then it starts showing up in the order stream.

The only parts that weren't copied were the image files themselves. They're too large to copy, and so instead we assign every image—whether part of a design or taken in the studio—its own GUID. As a rule, once something gets a GUID, it is officially immutable. When it's getting ready to burn orders to DVD, the StudioServer walks through the orders collecting GUIDs (using the controversial Visitor pattern). It adds every image it finds to the DVD, including both the customers' photographs and the design backgrounds.

Render Farm

The StudioClient helps associates create enhanced portraits from the basic images. Those enhanced portraits can be as simple as a sepia or black and white effect to make the portrait look more dramatic, or they can be as complex as a multilayered structure with alpha-composited backgrounds, text, and soft focus. Whatever the effect, the workstations in the studio do not produce the final rendered image. The printing facility has a variety of printers, supporting different sizes and resolutions. They're free to change printers or move jobs between printers at any time. The studios just don't know enough to produce the print-ready images.

When those daily DVDs arrive, they get loaded into the production control system (PCS). PCS makes all the decisions about when to render the images for an order, when to print them, and what printers to send them to. A separate team, in a separate location and in a separate time zone, develops PCS. Previous projects had run into tremendous friction when trying to integrate too closely with PCS. All parties worked with good intentions, but the communication difficulty slowed both teams down. We needed to avoid that friction, and so we decided to apply Conway's Law (defined in the next section) proactively, by explicitly creating an interface in the software where we knew the team boundary would be.

Conway's Law, applied

Conway's Law is often invoked after the fact, to explain what might otherwise appear to be arbitrary divisions within a product. It speaks to a fundamental truth about development teams: anywhere there is a team boundary, you will find a software boundary. This emerges from the need to communicate about interfaces.

We felt it was important enough to keep the DVD format and layout under complete control of Creation Center that we added a program to our own scope: the DvdLoader. DvdLoader

Copyrighted material

runs in the production facility, reading DVDs and calling various stored procedures within PCS to add orders, compositions, and images. PCS treats the composition instructions as an opaque string, and we were careful to avoid any decisions that would have PCS "opening up" the XML in that string. That sometimes means we duplicate information, such as dependencies on the base images themselves, but that is an acceptable trade-off for maintaining a clear boundary.

Similarly, we defined an interface that let the RenderEngine pull render jobs from PCS while keeping the XML description of the rendering itself under Creation Center's control.

We worked out written specifications of those interfaces, and then used FIT running on our development server to "nail down" the precise meaning. In effect, we used FIT as an executable specification of the interfaces. That turned out to be vital because even the people who negotiated the interface still found discrepancies between what they thought they agreed to and what they actually built. FIT let us eliminate those discrepancies during development rather than during integration testing, or worse, in production.

INCREMENTAL ARCHITECTURE

One of the recurring questions in the agile community is, "How much architecture should you create up front?" Some of the leading agile thinkers will tell you, "None. Refactor mercilessly and the architecture will emerge." I've never been in that camp.

Refactoring improves the design of code without changing its functionality. But, to refactor your way to better design, you must first be able to recognize good and bad design. We have a good catalog of "code smells" to guide us there, but I don't know of any equivalent for "architecture smells." Second, it must be possible to change things continuously even across interface boundaries. This has always led me to believe that a system's fundamental architecture must be in place at the start of development.

Now, after the Creation Center project, I'm much less confident in that answer. We added major pieces of the architecture relatively late in the project. Here are some examples:

- Hibernate: Added after two or three iterations. We didn't need the database before this.
- Spring: Added nearly one-third of the way to release 1.0. It quickly became central to our architecture. I don't remember how we got along without it, but we did.
- FIT: Added halfway to release 1.0.
- DVD-burning software: Purchased and added near the end of initial development.
- Support for windowed UIs: Added in the final two iterations before launch.

Copyrighted material

In each case, we took the approach of exploring options thoroughly before making decisions. We would make a decision at the "last responsible moment," that point where the cost of *not* deciding outweighed the cost of implementing the feature. Although there were a few things that we might have done differently if Spring had been there from the start, we were not harmed by adding it later. In those early iterations, we focused on uncovering what the application wanted to be rather than how Spring wants us to build applications.

DVD loading

The `DvdLoader` program, which runs in the printing facility, is really a batch processor that reads orders from DVDs and loads them into PCS. As with everything else, we focused on robustness. `DvdLoader` reads an entire order, verifying that the DVD includes all the constituent elements, before it adds the order to PCS. That way it doesn't leave partial or corrupted orders in the database.

Because images can appear on many DVDs, the loader checks to see whether there's already an image loaded with that GUID. If not, the loader adds it. Orders can therefore be resent from the studio whenever necessary, even if PCS has already purged the order and its underlying images. This also means that the background images used in a design get loaded the first time an order for that design arrives.

The DVDs are therefore self-contained and idempotent.

Render pipeline

For the render engine itself, we drew on the classic pipes and filters architecture. "Pipeline" is a natural metaphor for rendering images, and separating the complex sequence of actions into discrete steps also made unit testing simple.

On pulling a job from PCS, the render engine creates a `RenderRequest`. It passes the `RenderRequest` into the rendering pipeline, where each stage operates on the request itself. One of the final stages in the pipeline saves the rendered image to the path specified by PCS. By the time the request exits the pipeline, it holds only a result object with a success indicator and an optional collection of problems.

Each step in the pipeline has its own opportunity to report problems by adding an error message to the result. If any step reports errors, the pipeline aborts and the engine reports the problem back to PCS.

Fail fast

Every system has failure modes; the only question is whether you design them in or just let them happen. We took care to design in "safe" failures, particularly in the production process. There was no way we wanted our software to be responsible for stopping the production line.

Copyrighted material

There's another aspect, too. When the customer picks up his order, it should be the right one! That is, the product we deliver really needs to match the product the customer ordered. It seems like a trivial statement, but it is very important to render the production scale images in the same way that the on-screen image was rendered. We worked hard to ensure that exactly the same rendering code would be used in production as in the studio. We also made sure that the rendering engine would use the same fonts and backgrounds in production.

In our render engine, we adopted a philosophy of "Fail Fast, Fail Loudly." As soon as the render engine pulls a job from PCS, it checks through all the instructions, validating that all the resources the job requires are actually available. If the job includes text, the render engine loads the font right away. If the job includes some background images or an alpha mask, the render engine loads the underlying images right away. If anything is missing, it immediately notifies PCS of the error and aborts that job. Out of the 16 steps in the rendering pipeline, the first 5 all deal with validation.

After several months in production, we finally found one error that the render engine didn't detect early: it didn't reserve disk space for the rendered image up front. One day when PCS filled its storage volumes, render jobs started to fail late instead of failing early. In all the preceding time, there were no remakes due to bad renders.

Scale out

Each render engine operates independently. PCS doesn't keep a roster of the render engines that exist; each engine just pulls jobs from PCS. In fact, engines can be added or removed as needed. Because each engine looks for a new job as soon as it finishes the previous one, we automatically get load balancing, scaled to the horsepower of the individual engines. Faster render engines just consume jobs at a higher rate. Heterogeneous render engines are no problem.

The only bottleneck would be PCS itself. Because the render engines call stored procedures to pull jobs and update status, each render engine generates two transactions every three to five minutes. PCS runs on a decent-sized cluster of Microsoft SQL Server hosts, so it is in no danger of limiting throughput anytime soon.

User Response

Our first release was installed at two local studios, both within easy "drive-and-debug" distance. The associates' feedback was immediate and very positive. One studio manager estimated that the new system was so much faster and easier to use that she would be able to handle 50% more customers during the holiday season. One customer was reported to ask where she could buy a copy of the software. We commonly heard reports of customers taking the mouse directly and making their own enhancements. You can imagine that customers are much more likely to order products they've created themselves.

Copyrighted material

We had a few kinks in the production process, but those were corrected very quickly. Thanks to the resilience we built into the loader and render farm, the printing facility has been able to scale up to handle the volume from many more studios than originally expected, while also enjoying higher production quality.

Conclusion

I could spend much more time and space with fond descriptions of every class, interaction, or design decision, with the devotion of a new parent describing his infant's every burp and wobble. Instead, this chapter condenses a year's worth of effort, exploration, blood, and sweat. It illustrates how the structure and dynamics of the Creation Center architecture emerged from fundamental forces about the business and its context. By keeping concerns well separated and guiding the incremental design and development, Creation Center balanced those forces in a pleasing way.

References

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern for Distributed Computing*, vol. 4. Hoboken, NJ: Wiley.

Fowler, Martin. 1996. *Analysis Patterns: Reusable Object Models*. Boston, MA: Addison-Wesley.

Fowler, Martin. "Application facades." *http://martinfowler.com/apsupp/appfacades.pdf.*

Gamma, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley.

Lea, Doug. 2000. *Concurrent Programming in Java*, Second Edition. Boston, MA: Addison-Wesley.

Martin, Robert C. 2002. *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice-Hall.

Copyrighted material