Eating the IT Elephant

Moving from Greenfield Development to Brownfield

Richard Hopkins and Kevin Jenkins

Forewords by IBM Fellows Grady Booch and Chris Winter

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

© Copyright 2008 by International Business Machines Corporation. All rights reserved.

Note to U.S. Government Users: Documentation related to restricted right. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

IBM Press Program Managers: Tara Woodman, Ellice Uffer

Cover design: IBM Corporation

Associate Publisher: Greg Wiegand
Marketing Manager: Kourtnaye Sturgeon
Publicist: Heather Fox
Acquisitions Editor: Katherine Bull
Development Editors: Kevin Ferguson, Ginny Bess
Managing Editor: Gina Kanouse
Designer: Alan Clements
Senior Project Editor: Lori Lyons
Copy Editor: Krista Hansing
Indexer: Lisa Stumpf
Compositor: Nonie Ratcliff
Proofreader: Anne Goebel
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc

Publishing as IBM Press

IBM Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com.

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both: IBM, the IBM logo, IBM Press, AD/Cycle, DB2, developerWorks, Rational, System 360, Tivoli and WebSphere. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

This Book Is Safari Enabled
The Safari, Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days. Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.
To gain 45-day Safari Enabled access to this book:

- Go to http://www.awprofessional.com/safarienabled
- Complete the brief registration form
- Enter the coupon code L3LW-38PM-8WA6-9FMJ-ZQUE

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Library of Congress Cataloging-in-Publication Data

Hopkins, Richard.
Eating the IT elephant : moving from greenfield development to brownfield / Richard Hopkins and Kevin Jenkins.
p. cm.
Includes index.
ISBN 0-13-713012-0 (pbk. : alk. paper) 1. Information technology. 2. Business enterprises—Planning. I. Jenkins, Kevin. II. Title.
T58.5.H69 2008
004.068—dc22
2008004231

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-013713012

ISBN-10: 0137130120

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

First printing May 2008

Foreword by Grady Booch

A simple back-of-the-envelope calculation suggests that, worldwide, we produce about 33 billion lines of new or modified code every year. Cumulatively, this means that since the 1940s and '50s (when higher order programming languages began to gain some traction), we've produced somewhere around one trillion source lines of code.

On the one hand, this volume of output suggests that ours is an incredibly vibrant and innovative industry. On the other hand, it's a humbling thought, for through those trillion lines of code, all handcrafted by individual human labor, we've changed the world.

Truth be told, some nontrivial percentage of the 33 billion lines yearly is dead on arrival or so transitory that it's thrown away quickly. Much of that code, however, has a longer half-life, and even some of that code lives after 10, 20, or even 30 or more years. For many developers, the code they write today becomes tomorrow's legacy that their children or their children's children may stare at some day, trying to use it, adapt it, evolve it, asking the question, "What the heck was this developer thinking?"

Greenfield development, quite honestly, is great fun, simply because you get to start with a clean slate and, thus, are not burdened by anything from the past. For the most part, we teach Greenfield development in our schools; furthermore, start-up companies look so much more nimble than their older counterparts because they don't have the millstone of legacy around their necks. Woe be unto the student who enters the real world (it's not like being at the university, unless you move from the college womb to the start-up womb immediately), and woe be unto the start-up company that begins to mature into sustainable development and soon realizes that you can't just start over.

Richard and Kevin introduce us to a reality that's often neglected in our industry: the problem of evolving legacy systems, a domain they call Brownfield development. The typical economically interesting system these days is continuously evolving (you can't shut it off) and ever-growing. The authors identify the root of the problem as that of complexity, and offer an approach that focuses on the fundamentals of abstraction and efficient communication to nibble at this problem of transformation bit by bit. Their model of Views, Inventory, Transforms, and Artifacts offers an approach to reasoning about and executing on the transformation of Brownfield systems. They propose a Brownfield lifecycle involving surveying, engineering, acceptance, and deployment that offers a means of governing this transformation.

As the old saying goes, the way you eat the elephant is one bite at a time. Richard and Kevin bring us to the table with knife and fork and other tools, and show us a way to devour this elephant in the room.

Grady Booch
IBM Fellow
January 2008

Foreword by Chris Winter

I joined the computer industry as a computer programmer, straight from school, in 1969. During a career that has spanned nearly 40 years, I have worked primarily in the area of applications development and systems integration. I wrote my first application in 1969; it was a Computer Aided Design (CAD) graphics application for hardware engineers to design Printed Circuit Boards. This application gave the board designer a tool with the necessary physical rules of the electronic components and how they could be used. In the early 1970s, I developed CAD and other applications to assist building architects in designing large public buildings, such as schools and hospitals. These systems assisted the architects and civil engineers in the design process of the building; by capturing the design, it was possible to produce all the necessary drawings together with the bills of materials for the building.

In the intervening 40 years, I have performed a variety of different roles, including programmer, analyst, designer, architect, project manager, and troubleshooter. The systems I developed were in a broad spectrum of industries, including manufacturing, banking, insurance, retail, utilities, and both local and federal government. Today, I am an IBM Fellow[1] in the IBM Global Business Services division and an active member of the IBM Academy of Technology.[2] My primary responsibility is to technically shape and ensure the technical health of large and complex systems integration and strategic outsourcing programs and bids. I am a Chartered IT Professional (CITP), a Chartered Engineer (CEng), a Fellow of the British Computer Society (FBCS),[3] and a Fellow of the Institution of Engineering and Technology (FIET).[4]

Looking back now on what we tried to achieve with the design and build of electronic circuits and buildings in the early 1970s, I am disappointed and somewhat disillusioned by the IT industry's lack of success in its own adoption of engineering-based methods supported by computer-based tools to architect, design, build, integrate, and test IT systems. In today's world, it would be inconceivable to develop a complex system such as the Airbus 380 without the engineering disciplines and without the engineering tools provided by the IT industry. The IT industry is significantly less mature at adopting engineering techniques to develop its complex systems. It can no longer rely on relatively immature practices often supported by office productivity tools such as word processors, presentation tools, and spreadsheets. The IT industry needs a broader adoption of true engineering-based techniques supported by tools designed for engineers.

It has been my personal experience in recent years that the overall cost and complexity of building bespoke (custom) applications or customizing Commercial Off The Shelf (COTS) packages has increased—as has the risk. On further investigation, it is apparent that it is not the build cost that has increased, but the increase in the size and complexity of the integration of such projects into the systems landscape. From my own recent experience, the ratio of effort of new build to integration is 3:1. For every dollar spent on new functionality, the total cost is four dollars to cutover this function into production. This cost excludes end-user training. In an environment where both size and complexity of the systems landscape are continually increasing, there is a resulting increase in the costs of maintenance. In addition, organizations are burdened with a need to meet increasing levels of legislation and regulation. All of this results in reduced budgets for new development together with decreasing windows of opportunity to deploy new function in the global 24 x 7 service culture. IT innovation is being stifled. The methods and tools that are in use today, albeit limited, are in the main, primarily targeted at Greenfield system's landscapes. The reality is that most organizations in the twenty-first century have an existing, complex systems landscape. When I refer to the systems landscape, I mean both the business and its enabling IT systems. These IT systems, in turn, are comprised of applications and their data deployed on often complex network and computer infrastructure. The documentation of such systems is typically poor and its ongoing maintenance is highly dependent on a small number of knowledgeable "system experts."[5] The IT industry needs a more structured approach to understanding these system landscapes.

This is the reality of the world in which the authors of this book, Richard Hopkins and Kevin Jenkins, and I, architect, design, and implement new

systems for our clients in existing complex systems landscapes. It is time that the IT industry face up to the reality of the situation and the need for new development methods and tools that address these issues and take our industry into the twenty-first century.

An important first step in resolving this is to provide a name that describes both the problem and its solution. In the search for a name, the authors have turned to the building industry where new buildings are increasingly being developed on Brownfield[6] sites. This is analogous to the majority of today's new systems that are being developed on Brownfield systems landscapes; it is my experience that more than 90 percent of new development is deployed into a Brownfield environment. The challenges are not restricted to just the transformation of legacy systems, but with the integration into the Brownfield systems landscape itself.

This book describes a new approach to the development of future systems. It is a structured approach that recognizes these challenges, it is based on engineering principles, and it is supported by appropriate tooling. It is specifically designed to solve the challenges of Brownfield development.

Chris Winter
CEng CITP FBCS FIET, IBM Fellow
Member of the IBM Academy of Technology

Foreword Endnotes

1 "IBM Appoints Six New Fellows Who Explore the Boundaries of Technology." http://www-03.ibm.com/press/us/en/pressrelease/21554.wss, May 2007.
2 IBM Academy. http://www-03.ibm.com/ibm/academy/index.html.
3 British Computer Society. http://www.bcs.org/.
4 The Institution of Engineering and Technology. http://www.theiet.org/.
5 Lindeque, P. "Why do large IT programmes fail?" http://www.ingenia.org.uk/ingenia/articles.aspx?Index=390, September 2006.
6 Brownfield is described by the National Association of Realtors® as "The redevelopment of existing urban, suburban, and rural properties already served by infrastructure including 'brownfields' sites, that are or may be contaminated, stimulates growth and improves a community's economic vitality. Development in existing neighborhoods is an approach to growth that can be cost-effective while providing residents with a closer proximity to jobs, public services, and amenities."

Preface

Within every business, there is a desire for rapid change to meet customer demands. Such changes usually involve changing supporting IT systems. When large business changes are required, the accompanying IT changes tend to be significant, too. However, all too often, these big projects hit problems, run over budget, are delayed, or simply get cancelled. Even in 2006, 65% of IT projects failed on one of these counts.[1] Large projects have an even poorer success rate. Such odds are very worrying when the stakes are very high. This book identifies the fundamental issues at the heart of the IT industry's current approaches and provides a new way forward. All people involved in large-scale business and IT change should read this book.

The Day the Elephant Was Born

The IT industry has many key dates, but the introduction in 1964 of IBM's new-generation mainframe, called the System/360, marked the start of a new era. Until that point, buying a new business computer meant rewriting your existing software. The System/360 changed all that with the introduction of a family of compatible computers and associated devices: A program that ran on one would run on any. The industry followed suit with equivalent products, and the nature of IT changed in one fell swoop.

IT investments could now be easily preserved. The programs that ran on the System/360 still run on IBM's mainframe platforms today.

This was an imperceptible change at first, but it was a hugely significant milestone. At this point, IT complexity started accumulating within the enterprise. Systems grew with the business. Thousands of person-years of

time, effort, and money flowed into these IT systems. They got complex. They became elephants.

In the meantime, IT fashions came and went. Over the years, the original structured programs have been augmented by object-oriented programming, wrapped by component-based development, and advertised by Service Oriented Architecture (SOA). Each of these movements has had its own strategy for dealing with the complexity, but none ever really took it to heart.

Today's IT systems are so complex that they simply defy everyday comprehension, spilling out of our minds as we try to get our heads around them. Responsibility for maintaining them is split among a variety of skilled groups and myriad products and programs that coexist to support the functions of the enterprise. To deal with this Hydra, we draw high-level architecture diagrams that comfort us by making things look simple. These diagrams are an illusion, a trick, a facade. They are, at best, approximations for easy consumption and high-level communication. At worst, they instill false optimism about our ability to make changes to that complexity.

Such "fluffy cloud" diagrams cannot hide genuine complexity forever. To achieve your business goals and change those systems, you must understand, communicate, and harness the real complexity. No one can understand the whole beast, so vast amounts of well-coordinated teamwork and unambiguous communication are required to complete such tasks. This combination of high levels of complexity and the need for clear communication of that complexity among hundreds of individuals destroys big projects.

Do I Need to Move from Greenfield to Brownfield?

IT systems are generally not implemented on Greenfields any more. The accumulated complexity since 1964 means that the environment for most big IT projects is one of immense challenge, entangled in an almost uncountable number of environmental constraints.

This is the underlying reason for the demise of most large-scale IT projects. Only 30% of large IT projects succeed.

Big projects are usually executed on "contaminated" sites, where you need to be careful of where and how you build; a change in one place can ripple through to other systems in unexpected ways. Such sites are more brown than green, and the IT industry needs to adopt a Brownfield-oriented approach to address them successfully.

This book introduces such a Brownfield approach and explains why current methods are still essentially Greenfield. It is specifically written for

people who want to change their business and know that they can do it only by building on what has gone before. If *any* of the following is true, this book is for you:

- You are a CIO, CTO, IT director, project executive, project director, chief architect, or lead analyst who is contemplating a significant change in your IT landscape.
- You cannot afford to replace your whole IT landscape.
- Your systems talk to a fair number of systems outside your direct control.
- You would like to reengineer your existing IT environment so that it will remain flexible for the future.
- You are deeply unhappy with the current failure rates of large IT projects.
- You are contemplating sending a significant part of your IT development and testing work off-shore.

Eating the IT Elephant was written by two full-time Executive IT Architects from IBM who can and have ticked every single one of those boxes on a number of occasions. We have been accountable for the technical direction and day-to-day implementation of some of the largest systems integration and reengineering projects that IBM has undertaken. We believe strongly that existing Greenfield development approaches are an increasingly poor means of addressing today's business problems through IT solutioning. To be blunt, we have a number of years of hard-won experience, and we have grown tired of the recurring problems of IT delivery. In recent years, we have deliberately sought a different approach; the following pages detail the fruits of our labors and that of our colleagues. Heretics we might be, but pragmatists we are also, and, hand on heart, we can say that the insight we share here has significantly accelerated and simplified a number of recent IBM engagements.

We don't think the high failure rate of major IT projects is doing our industry any favors and would like to popularize the approach that has served us well. If we can help mitigate the impact of the unavoidably complex IT environment and knock down some big project communication barriers, we believe that success rate will improve.

A Reader's Digest

This book is not a technical manual nor a cookbook; it does not contain a single line of code, and we have tried to minimize the use of technical diagrams and jargon. This is a book about changing the way we approach large and complex business and IT reengineering projects.

To make the book as accessible to as many people as possible, we have split it into two parts.

Part I is for all readers. Initially, it defines what is wrong with large-scale IT projects and determines the root cause of failure (see Chapters 1, "Eating Elephants Is Difficult," and 2, "The Confusion of Tongues"). The heart of the book (Chapters 3, "Big-Mouthed Superhero Required," and 4, "The Trunk Road to the Brain") concentrate on defining an alternative solution—an Elephant Eater—and the Brownfield approach that goes with it. In Chapter 5, "The Mythical Metaman," we look at the new species of businesses that emerge as a result.

Part II explains the technical and practical aspects of Brownfield for someone who might want to implement such an approach. It starts by analyzing existing Elephant Eating techniques (see Chapter 6, "Abstraction Works Only in a Perfect World") and explains why Brownfield is different (see Chapter 7, "Evolution of the Elephant Eater"). In Chapters 8, "Brownfield Development," and 9, "Inside the Elephant Eater," we look inside the Elephant Eater and at some of the new technologies that have been used to implement it. The book concludes by explaining how the Brownfield approach can be implemented on a project and the benefits it can bring (see Chapter 10, "Elephant Eater at Work").

For those who take the key messages on board, a wealth of technical information has already been published that will enable any organization to adopt the core technologies that we have used (or equivalent ones) to implement Brownfield in their own way (see the "Endnotes" sections of Chapters 8 and 9). We hope that enabling business and IT change via a new project approach, not technology, is at the heart of this book.

Part I: Introducing Brownfield

Chapter 1, "Eating Elephants Is Difficult," introduces the metaphor that performing a large IT project can be compared to eating an elephant. It looks at why big projects fail and provides best practices on how to overcome some of the common reasons for failure.

Chapter 2, “The Confusion of Tongues,” explains why this accumulated IT complexity is the root cause of failure, focusing on the human communication problems it creates. It goes on to specifically examine the “great divide” between business and IT that compounds the problem.

Chapter 3, “Big-Mouthed Superhero Required,” introduces the core concepts of Brownfield. It looks at how Brownfield can be implemented to create an efficient Elephant Eater.

Chapter 4, “The Trunk Road to the Brain”: We despair at IT professionals' inability to communicate as effectively and efficiently as those in other similar professions (such as real architects). Chapter 4 describes how the Brownfield approach combined with the VITA architecture opens up new forms of communication, remote collaboration, and visualization of complex IT problems.

Chapter 5, “The Mythical Metaman”: The first part of the book concludes with an examination of the likely impact of Brownfield. It forecasts a new breed of businesses that are infinitely more customer focused and agile than today's and explains how such businesses might come into being.

Part II: The Elephant Eater

Chapter 6, “Abstraction Works Only in a Perfect World”: This more technical half of the book opens by defining the characteristics of an Elephant Eater. It considers existing “Elephant Eating” approaches and notes that they tend to compound project difficulties via their extensive use of decomposition and abstraction.

Chapter 7, “Evolution of the Elephant Eater,” looks at Brownfield's technical and project roots, and explains its key differences from previous ideas. It ends with some likely scenarios and real-life project examples for which Brownfield has been or could be especially beneficial.

Chapter 8, “Brownfield Development,” introduces how the Brownfield development approach can be deployed on a project. It shows how to strike a new balance between Agile- and Waterfall-based development techniques and provides some of the best elements of each. It also describes the core phases of Survey, Engineer, Accept, and Deploy, and states the benefits of the approach.

Chapter 9, “Inside the Elephant Eater”: If Chapter 8 described what happens on a Brownfield project, Chapter 9 explains how it happens. This chapter looks inside the workings of an Elephant Eater and explains how it eats the elephant. The chapter also serves as an easy-to-read introduction to the new semantic technologies that underpin Web 2.0 and the semantic web.

Chapter 10, "Elephant Eater at Work": The book concludes with a look at the practical applications of the Elephant Eater and how it can help solve some of today's most difficult IT problems. This chapter includes a summary of the key benefits of the Brownfield approach.

Walking the Brownfields

We hope that you will enjoy reading this book as much as we enjoyed writing it. If you'd like to see more, go to the website www.elephanteaters.org. Additionally, if you would like to see more of the dynamic nature of Brownfield, there are two exhibitions in Second Life. One Second Life site is dedicated to the book {Cypa 30,180,302}. The other Second Life site is dedicated to the use of Brownfield within IBM at {IBM 1 140, 150, 60}. We look forward to meeting you there.

Endnotes

[1] The initial CHAOS report from Standish Group in 1994 reported a 16% success rate for IT projects. This success rate has generally increased over the intervening years. In 2006, Standish Group reported 35% of IT projects being on time and within budget, and meeting user requirements. The only blip in that record appeared in 2004, when failure rates increased. Standish Group explained that in 2004 there were more big projects—they fail more often because they are often forced to abandon iterative development techniques.

In 2007, a rival report to CHAOS by Sauer, Gemino, and Horner Reich looked at 412 projects. It found that more than 65% of IT projects succeeded, but it found no successful projects greater than 200 person-years. This book looks specifically at those large projects.

Hayes, F. *Chaos Is Back.* www.computerworld.com/managementtopics/management/project/story/0,10801,97283,00.html.

Krigsman, M. *Rearranging the Deck Chairs: IT Project Failures.* http://blogs.zdnet.com/projectfailures/?p=513.

Rubinstein, D. *Standish Group Report.* www.sdtimes.com/article/story-20070301-01.html.

Sauer, C., A. Gemino, and B. Horner Reigh. "The Impact and Size and Volatility on IT Project Performance." *Communications of the ACM* 50 no. 11 (November 2007): 79–84.

6

Abstraction Works Only in a Perfect World

"There is no abstract art. You must always start with something. Afterward you can remove all traces of reality."
—Pablo Picasso

Chapter Contents

- Considerations for an Elephant Eater 110
- Systems Integration and Engineering Techniques 112
- Abstraction Is the Heart of Architecture 118
- Do We Need a Grand Unified Tool? 128
- The Connoisseur's Guide to Eating Elephants 129
- Endnotes 131

In the first part of the book, we saw how IT systems have grown increasingly larger and more complex over time. This growing complexity is challenging the capability of businesses to innovate as more of the IT budget is channeled into regulatory compliance, replatforming, and maintenance of the status quo. As this book has shown, changing these systems is not primarily a technical difficulty, but one of coordinating and disambiguating human communication. In overcoming such difficulties, we have introduced the concept of an Elephant Eater and the Brownfield development approach.

This second part of the book explains the technical and practical aspects of Brownfield for someone who might want to implement such an approach. This chapter examines the necessary technical context, requirements, and characteristics of the Elephant Eater. The chapter then goes on to analyze existing IT elephant-eating approaches and highlights the problems these approaches present with their extensive use of decomposition and abstraction.

Considerations for an Elephant Eater

The following sections outline considerations for the Elephant Eater. The problems with large scale developments are many, and the first half of the book illustrated some of the problems that such developments pose. The high failure rate for such projects is the reason why the creation of an Elephant Eater was necessary. Like any problem, the starting point for a solution is the understanding of the requirements, so if an Elephant Eater is going to be created, it needs to cater to the considerations in this section.

Lack of Transparency

On very large-scale developments, the problem being solved usually is unclear. At a high level, the design and development task might seem to be understood—for example, "build a family home," "design a hospital," or "implement a customer relationship management system." However, such terms are insufficient to describe what is actually required.

For any complex problem, some degree of analysis and investigation is essential to properly frame the detailed requirements of the solution and understand its context. In conventional building architectures, the site survey is a fundamental part of the requirements-gathering process.

A thorough analysis of a complex site takes a great deal of time and effort. Even using traditional Greenfield methods, the analysis effort is often as

large as the build effort. Despite this effort, however, IT architects and business analysts rarely do as thorough a job of surveying a site as building architects do. As discussed in previous chapters, a thorough analysis that encompasses functional and nonfunctional requirements and multiple constraints requires vast documentation. As such, the real requirements in any situation are always less than transparent.

Unfortunately, in IT, relatively little time is spent on the equivalent of a site survey.

Multiple Conflicting Goals

Another problem is conflicting requirements. In any complex situation, a single optimal solution is rarely a given for such a problem. The problem itself might even be poorly described.

In the example of the house building discussion in Chapter 1, "Eating Elephants Is Difficult," the mother-in-law and the landowner could have very different perspectives on what is desirable. Will their combined requirements be entirely coherent and compatible? Whose job will it be to resolve these conflicts?

We have seen the same problem on multiple $100 million programs. Any big program owned by more than one powerful stakeholder is likely to fail because of confusing and conflicted directions. As we saw in Chapter 1, life is much easier when one powerful person is consistently in charge. Of course, assigning a single stakeholder is not easy, but failing to identify this stakeholder at the start of the project only ignores the problem.

Spotting requirements that are clearly expressed but in conflict is reasonably easy, and it is usually possible to resolve these through careful negotiation. No one would seriously demand two mutually incompatible set of requirements, right?

Let's return to the analogy of home building as an example. When designing a house, increasing the size of the windows will increase the feeling of light and space within the building and improve the view. But bigger windows will contribute to energy loss. Improved insulation in the walls or ceilings might compensate for this, but this could result in increased building costs or a reduced living area. Alternatively, the architect could request special triple glazing. That would make the windows more thermally efficient but could make the glass less translucent. As more concerns arise, the interplays between them become more complex. As a result, the final solution

becomes a trade-off between different aspects or characteristics of the solution. Possibly, the requirements are actually mutually incompatible—but this can be known only in the context of a solution.

These conflicting requirements also come up repeatedly when designing large computer systems. We hear comments similar to these: "We need the system to be hugely scaleable to cope with any unexpected demand. It must be available 24 hours a day, 7 days a week—even during upgrades or maintenance—but must be cheaper to build, run, and maintain than the last system." Obviously, such requirements are always in conflict.

Dynamic Aspects

The difficulty in coping with these requirements is compounded by the fact that they don't stand still. As you begin to interfere and interact with the problem, you change it. For example, talking to a user about what the system currently does could change that user's perception about what it needs to do. Interacting with the system during acceptance testing might overturn those initial perceptions again. Introducing the supposed solution into the environment could cause additional difficulties.

In IT systems, these side effects often result from a lack of understanding that installing a new IT system changes its surroundings. Subsequent changes also might need to be made to existing business procedures and best practices that are not directly part of the solution. These changes might alter people's jobs, their interaction with customers, or the skills they require.

In addition to these impacts, the time involved in such projects usually means that the business environment in which the solution is to be placed has evolved. Some of the original requirements might need to change or might no longer be applicable

Therefore, one of the key requirements for any Elephant Eater is tight and dynamic linkage between the business and IT.

Systems Integration and Engineering Techniques

But the problem we're talking about isn't new, is it? People have been trying to deliver complex systems for more than 40 years. There must already be some pretty reasonable Elephant Eaters out there.

Now that we have a good understanding of the problem, it's a good idea to take a closer look at some of the solutions that are already out there and see

why, given the meager 30 percent success rate noted in the Preface, we need a new Elephant Eater.

Generally, these big problems need to be approached via formal techniques. These techniques work from two different directions. They either work their way down from the top, gaining increasing levels of detail, or they start from the bottom, examining needs in detail and working their way upward, building toward a complete solution.

Walk the Easy Path or…

If you're infinitely lucky, the bottom-up approach might work. Considering a very simple example, you could select a package that seems close to what you need. You could then walk through the business processes you want to execute. As you go, you can write down all the changes you need to make to the package, and, *presto!* After you've made the changes, you've got a solution! You've designed the whole system from the ground up because the package dictates your choices for how you do pretty much everything else.

If you don't allow the package to dictate your choices, chances are, you will find yourself in a very sticky mess: Each major change you make will require extra development, testing, and long-term maintenance costs. If you've chosen the bottom-up approach, you must stick to it religiously and accept the changes it will impose on the process and the business.

Ultimately, a package with a good fit, whether imposed or a lucky choice, is the very best in bottom-up solutions. Start halfway up the hill—the package already approximates what you want. Then modify the solution iteratively with the end user and find a happy endpoint near the top of the hill.

However, chances are, for a really complex project, using the bottom-up approach with a single package will not work. You must break down the problem into smaller pieces and then integrate them to create a single solution. You can divide up the problem in two fundamental ways.

…Break the Boulders and Make Them Smooth

You can decompose the problem into smaller, more easily managed Views through two methods: splitting and abstraction. Splitting simply divides complex big chunks into smaller, more manageable pieces. Abstraction removes detail from each larger chunk to form more manageable and understandable pieces. These two techniques, splitting and abstraction, allow almost any gargantuan problem to be subdivided into smaller, better contained problems. Think of it as slicing the problem into little squares.

Abstraction gives you horizontal cuts, while View splitting gives you vertical ones. Everything becomes a manageable "chunk." This is the basis for most systems integration and engineering methods. Many of these methods are proprietary, but some, such as The Open Group Architecture Framework (TOGAF) from the Open Foundation, are freely available. Each approach tries to create a continuum of knowledge, from high-level representations to more detailed. These paths vary but can be characterized as moving in some way from logical to physical, general to specific, or taxonomy to specification.

When good methods or tools are used, there is traceability from the high level to the low level. This helps a reader understand why something has been designed the way it has.

Such movement is unsurprisingly characterized as a progression, starting from the high-level principles and overall vision of what needs to be achieved, and moving down through the perspectives of business, process, roles, and models of information. Figure 6.1 highlights the basic stages of the TOGAF method.

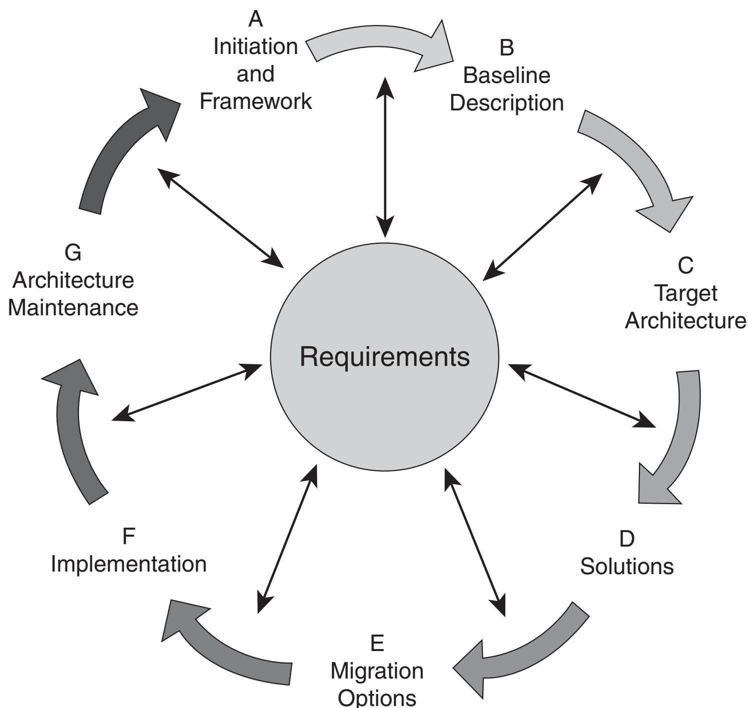

Figure 6.1 Even with its cyclic diagram, TOGAF is part of the progressive school of architecture.

Some approaches go even further. They segment each level of abstraction into a number of separate perspectives. Of these "frameworks," the enterprise architecture framework produced by John Zachman of IBM in the 1980s is probably the most famous. Called the Zachman Framework, it considers the additional dimension of Data, Function, Network, People, Time, and Motivation. Figure 6.2 illustrates how the Zachman Framework segments the architecture into these perspectives.

	What? Data	How? Function	Where? Network	Who? People	When? Time	Why? Motivation	
Planner							Scope
Owner							Enterprise Models
Designer							System Models
Builder							Technology Models
Sub-contractor							Detailed Represent-ations
Enterprise							Actual Systems

Figure 6.2 The Zachman Framework of Enterprise Architecture segments the architecture into a variety of perspectives.

These approaches enable you to decompose the full width and breadth of the problem (including the existing constraints) into separate Views so that a suitably skilled guru can independently govern and maintain them.

At the very top of this top-down approach is a simple sheet of paper that purports to show or describe the scope of the whole problem for that particular perspective. A single sheet of paper might even purport to summarize the 10,000-foot view for *all* the perspectives.

Below that top sheet are many more sheets that describe each element on the sheet above. This technique is so well recognized that it's applied to almost everything in complex problems, whether we're talking about the shape of the system, the business processes that it executes, or the description of the plan that will build it.

In this hierarchy of paper, the top tier is labeled Level 0; the next tier down, Level 1; and so on. At each layer, the number of sheets of paper increases, but each of these sheets is a manageable View. The problem has been successfully decomposed. In the example in Figure 6.3, our single-page business context that describes the boundaries of the problem we're solving is gradually decomposed into 60,000 pages of code, deployment information, and operational instructions that describe the whole solution. At each step of the way, the intermediate representations all correspond to a View.

After the problem has been decomposed into single sheets, or Views, rules must be written and applied to specify how they work together.

Surely that solves our problem. The elephant has been eaten. Complexity is reduced, so each area becomes manageable. Each person is dealing with only a bit of the problem.

This is, of course, precisely what the world's largest systems integrators do. They define their Views in terms of work products or deliverables. They come from different perspectives and at different levels of abstraction. The systems integrators have design and development methods that describe who should do what to which View (or work product) and in what order.

So if the problem is essentially solved, why does it go wrong so often?

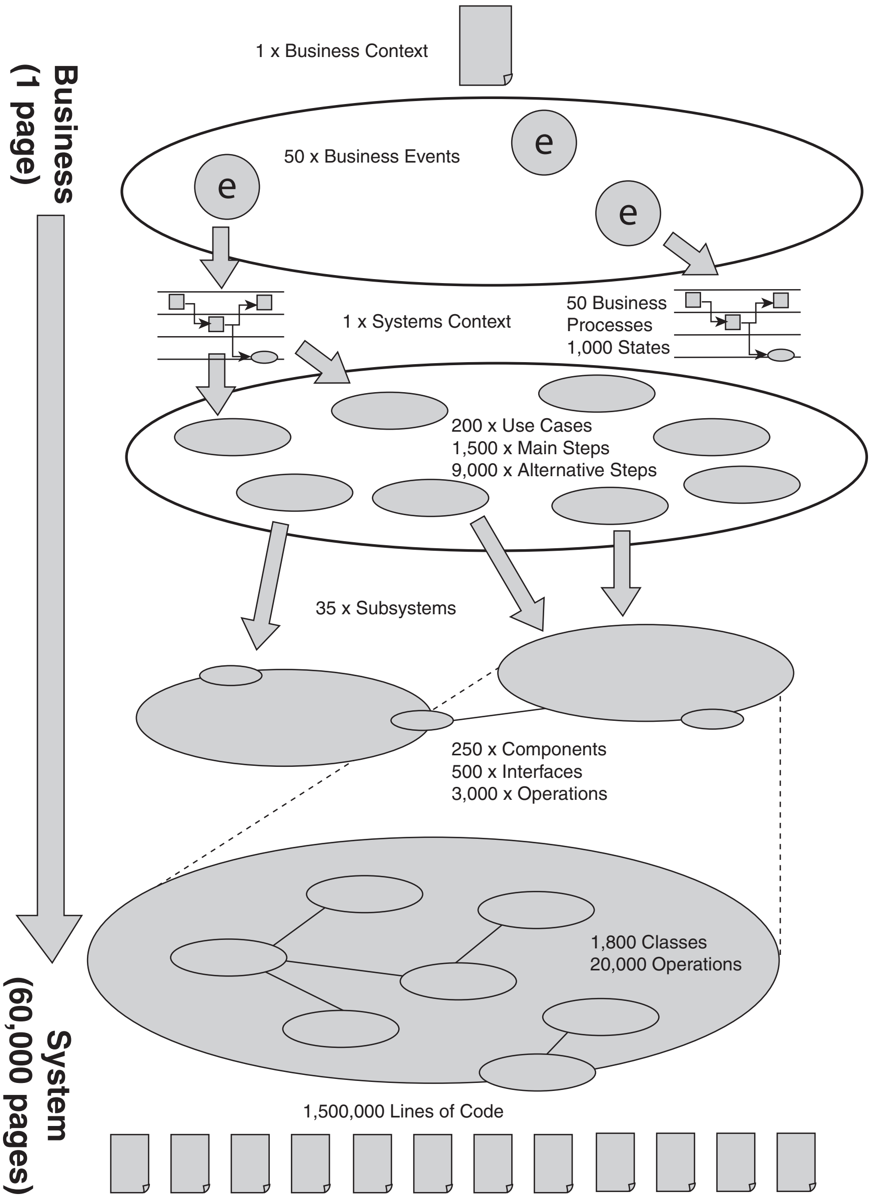

Figure 6.3 Decomposition of a complex problem space

Abstraction Is the Heart of Architecture

In all these cases, we move from the general to the specific, with the next layer of detail expanding upon the previous level of abstraction. This movement from general to specific gives architecture its power to simplify, communicate, and make ghastly complexity more aesthetically pleasing.

Abstraction is the heart of architecture. This powerful and persuasive concept has been at the center of most of the advances in complex systems architecting for the last 15 years. It underpins the history of software engineering—objects, components, and even IT services have their roots in abstraction. Because abstraction is one of our most powerful tools, we should consider its capabilities and limitations.

As systems have become more complex, additional layers of abstraction have been inserted into the software to keep everything understandable and maintainable. Year by year, programmers have gotten further away from the bits, registers, and native machine code, through the introduction of languages, layered software architectures, object-oriented languages, visual programming, modeling, packages, and even models of models (metamodeling).

Today, programs can be routinely written, tested, and deployed without manually writing a single line of code or even understanding the basics of how a computer works. A cornucopia of techniques and technologies can insulate today's programmers from the specifics and complexities of their surrounding environments. Writing a program is so simple that we can even get a computer to do it. We get used to the idea of being insulated from the complexity of the real world.

Mirror, Mirror on the Wall, Which Is the Fairest Software of All?

Software engineering approaches the complexity and unpredictability of the real world by abstracting the detail to something more convenient and incrementally improving the abstraction over time.

Working out the levels of abstraction that solve the problem (and will continue to solve the problem) is the key concern of the software architect. IBM's chief scientist Grady Booch and other leaders of the software industry are convinced that the best software should be capable of dealing with great complexity but also should be inherently simple and aesthetically pleasing.[1]

Thus, over time, we should expect that increasing levels of abstraction will enable our software to deal with more aspects of the real world. This is most obviously noticeable in games and virtual worlds, where the sophistication of the representation of the virtual reality has increased as individual elements of the problem are abstracted. Figure 6.4 shows how games architectures have matured over the last 20 years.

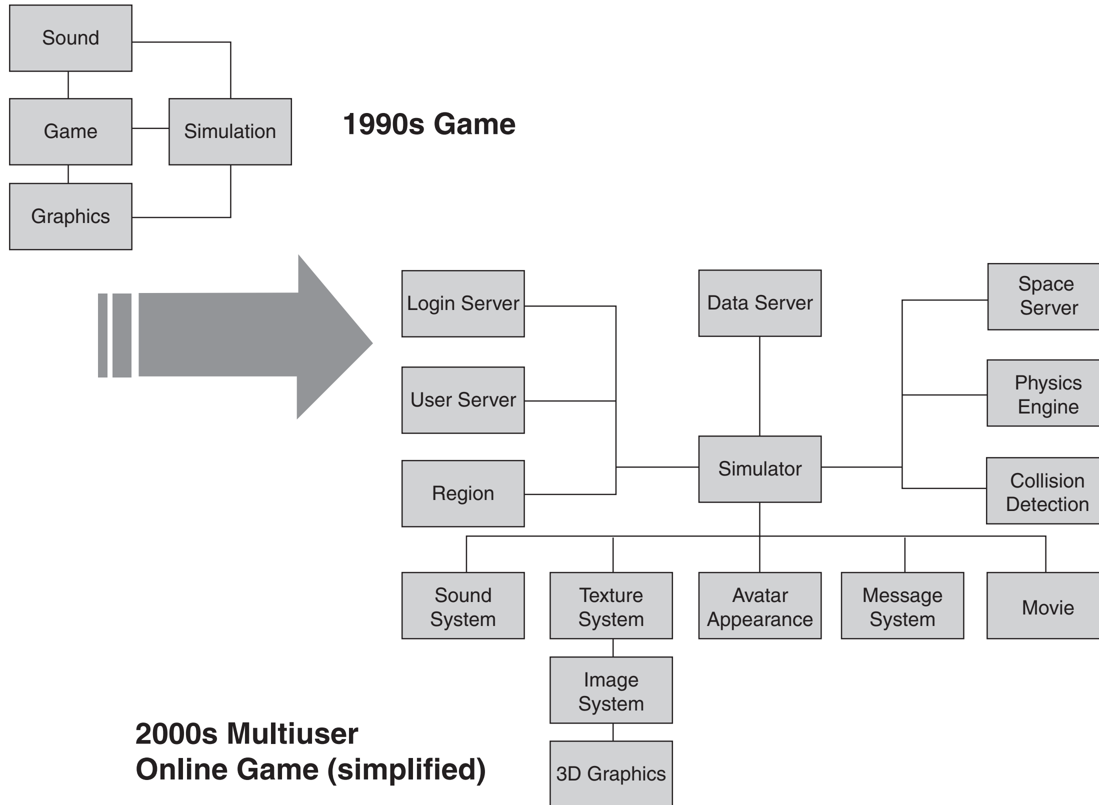

Figure 6.4 Games architectures have matured immensely over the last 20 years.

The current sophisticated, shared online games of the early twenty-first century exhibit greater descriptive power compared to the basic 2D games of the 1970s. Hiding the complexity of the physics engine from the graphical rendering system, and hiding both of these from the user server and the system that stores the in-world objects, enables increasing levels of sophisticated behavior.

Abstraction has its drawbacks, however. Each level of abstraction deliberately hides a certain amount of complexity. That's fine if you start with a complete description of the problem and work your way upward, but you must remember that this isn't the way today's systems integration and architecting methods work.

These methods start from the general and the abstract, and gradually refine the level of detail from there. Eventually, they drill down to reality. This sounds good. Superficially, it sounds almost like a scientific technique. For example, physicists conduct experiments in the real world, which has a lot of complexity, imperfection, and "noise" complicating their experiments. However, those experiments are designed to define or confirm useful and accurate abstractions of reality in the form of mathematical theories that will enable them to make successful predictions. Of course, the key difference between software engineering and physics is that the physicists are iteratively creating abstractions for something that already exists and refining the abstraction as more facts emerge. The architects, on the other hand, are abstracting first and then creating the detail to slot in behind the abstraction. Figure 6.5 should make the comparison clearer.

The IT approach should strike you as fundamentally wrong. If you need some convincing, instead of focusing on the rather abstract worlds of physics or IT, let's first take a look at something more down to earth: plumbing.

Plumbing the Depths

The IT and plumbing industries have much in common. Participants in both spend a great deal of time sucking their teeth, saying, "Well, I wouldn't have done it like that," or, "That'll cost a few dollars to put right." As in many other professions, they make sure that they shroud themselves in indecipherable private languages, acronyms, and anecdotes.

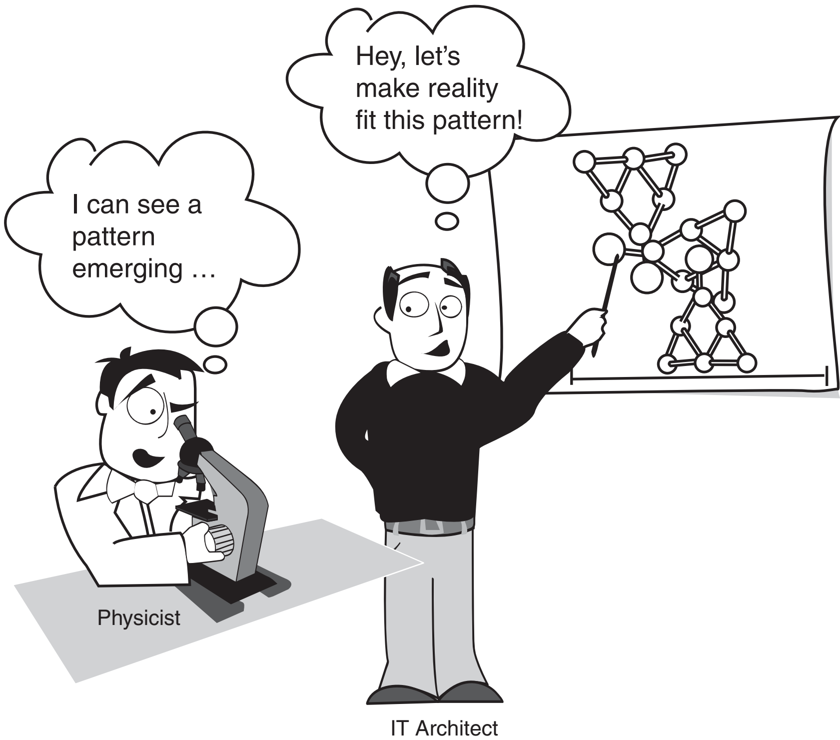

Figure 6.5 Who's right? Physicists or IT architects?

Imagine for a moment a heating engineer who has been asked to install a radiator in a new extension. He has looked at the plans and knows how he's going to get access to the pipes. From the specifications he's read, he knows what fixtures he needs. After doing some pretty easy calculations based on room size, window area, and wall type, he even got hold of the right size radiator to fit on the wall that will deliver the right amount of heat for the room. It's an hour's work, at most.

The job is done and he leaves a happy man. A few days later, the homeowner is complaining that the room is still cold. Only when the plumber arrives back on-site and investigates the boiler does he find out that the output of the boiler is now insufficient for the needs of the house. He recommends that the homeowner order a new 33-kilowatt boiler and arranges to come back in a week.

A week later, he's back to begin fitting the new boiler. Right at the start of the task, it becomes obvious that the old boiler was oil-fired and the new one is gas. This is slightly inconvenient because the property is not connected to the gas main, even though it runs past the property.

Another few weeks pass while the homeowner arranges for the house to be connected to the gas supply. On the plumber's third visit, everything is going swimmingly. Then he notices that there are no free breaker slots on the electricity circuit board to attach the new boiler. A week later, he replaces the circuit board. The boiler is installed, but another problem arises: Although the heat output of the boiler is sufficient, a more powerful pump is required to distribute the heat throughout the house.

And that's when the problems really start.

Don't Abstract Until You See the Whole Elephant

Judging from the architect's top-level view, the solution seemed pretty obvious. Only when the job was almost done was it obvious that it hadn't worked. Those other aspects of the problem—the supply, the pump, and the circuit board—were invisible from the Level 0 perspective the plumber received, so he ignored them in doing his analysis.

After all, nothing was fundamentally wrong with the plumber's solution; he just didn't have a good specification of the problem. The process of abstracting the problem to the single architectural drawing of the new room meant that he had no visibility of the real problem, which was somewhat bigger and more complex. He simply couldn't see the hidden requirements—the environmental constraints—from his top-level, incorrectly abstracted view of the problem.

Unfortunately, abstractions, per se, always lack details of the underlying complexity. The radiator was a good theoretical solution to the problem, but it was being treated as a simple abstract component that, when connected to the central heating system, would issue the right amount of heat. Behind that simple abstraction lays the real hidden complexity of the boiler, gas main, and circuit board that leaked through and derailed this abstracted solution.[2] Will such complexity always leak up through the pipe and derail simple abstract solutions?

Well, imagine for a moment that the abstraction was absolute and that it was impossible to trace backward from the radiator to the source of the heat. Consider, for example, that the heat to each radiator was supplied from one of a huge number of central utilities via shared pipes. If the complexity of that arrangement was completely hidden, you would not know who to complain to

if the radiator didn't work. Of course, on the positive side, the utility company supplying your heat wouldn't be able to bill you for adding a new radiator!

Is this such an absurd example? Consider today's IT infrastructures, with layers of software, each supposedly easier to maintain by hiding the complexities below. Who do you call when there is a problem? Is it in the application? The middleware? Maybe it is a problem with the database?

If you become completely insulated from the underlying complexity—or if you simply don't understand it, then it becomes very difficult to know what is happening when something goes wrong. Such an approach also encourages naïve rather than robust implementations. Abstractions that fully hide complexity ultimately cause problems because it is impossible to know what is going wrong.

Poorly formed abstractions can also create a lack of flexibility in any complex software architecture. If the wrong elements are chosen to be exposed to the layers above, people will have to find ways around the architecture, compromising its integrity. Establishing the right abstractions is more of an art than a science, but starting from a point of generalization is not a good place to start—it is possibly the worst.

Successful Abstraction Does Not Come from a Lack of Knowledge

In summary, abstraction is probably the single most powerful tool for the architect. It works well when used with care and when there is a deep understanding of the problem.

However, today's methods work from the general to the specific, so they essentially encourage and impose a lack of knowledge. Not surprisingly, therefore, the initial abstractions and decompositions that are made at the start of a big systems integration or development project often turn out to be wrong. Today's methods tend to ignore complexity while purporting to hide it.

The Ripple Effect

Poor abstractions lead to underestimations and misunderstandings galore. Everything looks so simple from 10,000 feet. On large projects, a saying goes that "All expensive mistakes are made on the first day." From our experience, it's an observation that is very, very true.

Working with a lack of information makes abstraction easy but inaccurate.

All projects are most optimistic right at the start. These early stages lack detailed information; as a result, assumptions are made and the big abstractions are decided.

Assumptions are not dangerous in themselves—as long as they are tracked. Unfortunately, all too often they are made but not tracked, and their impact is not understood. In some ways, they are treated as "risks that will never happen." Assumptions must always be tracked and reviewed, and their potential impact, if they're untrue, must be understood. Chances are, some of them will turn out to be false assumptions—and, chances are, those will be the ones with expensive consequences.

We need to move away from this optimistic, pretty-diagram school of architecture, in which making the right decisions is an art form of second guessing based on years of accumulated instinct and heuristics.[3] We need a more scientific approach with fewer assumptions and oversimplifications. A colleague, Bob Lojek, memorably said, "Once you understand the full problem, there is no problem."

Fundamentally, we need to put more effort into understanding the problem than prematurely defining the solution. As senior architects for IBM, we are often asked to intervene in client projects when things have gone awry. For example:

An Agile development method was being used to deliver a leading-edge, web-based, customer self-service solution for a world-leading credit card processor. The team had all the relevant skills, and the lead architect was a software engineering guru who knew the modern technology platform they were using and had delivered many projects in the past.

Given the new nature of the technology, the team had conformed strictly to the best-practice patterns for development and had created a technical prototype to ensure that the technology did what they wanted it to do. The design they had created was hugely elegant and was exactly in line with the customer requirement.

A problem arose, though. The project had run like a dream for 6 months, but it stalled in the final 3 months of the development. The reporting system for the project recorded correctly that 80 percent of the code had been written and was working, but the progress meter had stopped there and was not moving forward. IBM was asked to take a look and see what the problem was.

As usual, the answer was relatively straightforward. The levels of abstraction, or layering, of the system had been done according to theoretical best practice, but it was overly sophisticated for the job that needed to be done. The architecture failed the Occam's Razor test: The lead architect had induced unnecessary complexity, and his key architectural decisions around abstraction (and, to some extent, decomposition) of the problem had been made in isolation of the actual customer problem.

Second, and more important, the architect had ignored the inherent complexity of the solution. Although the user requirements were relatively straightforward and the Level 0 architecture perspectives were easy to understand, he had largely ignored the constraints imposed by the other systems that surrounded the self-service solution.

Yes, the design successfully performed a beautiful and elegant abstraction of the core concepts it needed to deal with—it's just that it didn't look anything like the systems to which it needed to be linked. As a result, the core activity for the previous 3 months had been a frantic attempt to map the new solution onto the limitations of the transactions and data models of the old. The mind-bending complexity of trying to pull together two mutually incompatible views of these new and old systems had paralyzed the delivery team. They didn't want to think the unthinkable. They had defined an elegant and best-practice solution to the wrong problem. In doing so, they had ignored hundreds of constraints that needed to be imposed on the new system.

When the project restarted with a core understanding of these constraints, it became straightforward to define the right levels of abstraction and separation of concerns. This provided an elegant and simple solution with flexibility in all the right places—without complicating the solution's relationship with its neighbors.

—R.H.

As a final horror story, consider a major customer case system for an important government agency:

> We were asked to intervene after the project (in the hands of another systems integrator) had made little progress after 2 years of investment.
>
> At this point, the customer had chosen a package to provide its overarching customer care solution. After significant analysis, this package had been accepted as a superb fit to the business and user requirements. Pretty much everything that was needed to replace the hugely complex legacy systems would come out of a box.
>
> However, it was thought that replacing a complete legacy system would be too risky. As a result, the decision was made to use half of the package for the end-user element of the strategic solution; the legacy systems the package was meant to replace would serve as its temporary back end (providing some of the complex logic and many of the interfaces that were necessary for an end-to-end solution).
>
> The decision was to eat half the elephant. On paper, from 10,000 feet, it looked straightforward. The high-level analysis had not pointed out any glitches, and the layering of the architecture and the separation of concerns appeared clean and simple.
>
> As the project progressed, however, it became apparent that the legacy system imposed a very different set of constraints on the package. Although they were highly similar from an end user and data perspective, the internal models of the new and old systems turned out to be hugely different—and these differences numbered in the thousands instead of the hundreds.
>
> Ultimately, the three-way conflict between the user requirements (which were based on the promise of a full new system), the new package, and the legacy system meant that something had to give. The requirements were deemed to be strategic and the legacy system was immovable, so the package had to change. This decision broke the first rule of bottom-up implementations mentioned earlier.

> Although the system was delivered on time and budget, and although it works to this day for thousands of users and millions of customers, the implementation was hugely complicated by the backflow of constraints from the legacy systems. As a result, it then proved uneconomic to move the system to subsequent major versions of the package. The desired strategic solution became a dead end.
>
> —K.J. and R.H.

In each of these cases, a better and more detailed understanding of the overall problem was needed than standard top-down approaches could provide. Such an understanding would have prevented the problems these projects encountered.

Each of these three problems stems from a basic and incorrect assumption by stakeholders that they could build a Greenfield implementation. At the credit card processor, this assumption held firm until they tried to integrate it with the existing infrastructure. The government department failed to realize that its original requirements were based on a survey of a completely different site (the one in which the legacy system was cleared away), resulting in large-scale customization of the original package that was supposedly a perfect fit.

Fundamentally, today's large-scale IT projects need to work around the constraints of their existing environment. Today's IT architects should regard themselves as Brownfield redevelopers first, and exciting and visionary architects second.

Companies that try to upgrade their hardware or software to the latest levels experience the same ripple effect of contamination from the existing environment. Despite the abstraction and layering of modern software and the imposed rigor of enterprise architectures, making changes to the low levels of systems still has a major impact on today's enterprises.

As we mentioned before, no abstraction is perfect and, to some extent, it will leak around the edges. This means there is no such thing as a nondisruptive change to any nontrivial environment. As a supposedly independent layer in the environment changes—perhaps a database, middleware, or operating system version—a ripple of change permeates around the environment.

As only certain combinations of products are supported, the change can cascade like a chain of dominoes. Ultimately, these ripples can hit applications, resulting in retesting, application changes, or even reintegration.

Thus, to provide good and true architectures, we need to accept that we need a better understanding of the problem to engineer the right abstractions. Additionally, we need all the aspects of the problem definition (business, application, and infrastructure) to be interlinked so that we can understand when and where the ripple effect of discovered constraints or changes will impact the solution we are defining.

Do We Need a Grand Unified Tool?

The problem definition is too big for one tool or person to maintain, so there appears to be a dilemma. The full complexity of the problem needs to be embraced, and an understanding is required of everything that's around, including the existing IT and business environments. But all that information needs to be pulled together so that the Views aren't discrete or disconnected.

Many people have argued for tool unification as a means to achieve this, to maintain all these connected Views in a single tool and, thus, enable a single documented version of the truth to be established and maintained. But that is missing a vital point about Views.

As explained in Chapter 2, "The Confusion of Tongues," Views need to be maintained by people in their own way, in their own language. Imposing a single tool will never work. Simply too many preferred perspectives, roles, and prejudices exist within our industry to believe that everyone is going to sit down one day and record and maintain their Views in one specific tool.[4] If such combinations of Views into single multipurpose tools were possible, desirable, and usable, then it is arguable that Microsoft® Office user interfaces Word®, PowerPoint®, and Excel® would have merged long ago.

Moreover, these integrated approaches that have been at the heart of traditional tooling are usually pretty poor at dealing with ambiguity or differences of opinion. On large projects with many people working on the same information, it is not unusual to have formal repositories that enable people to check out information, make changes to it, and then check it back in. Such systems prevent two people from updating the same information at the same time, which would result in confusion and conflicts. The upshot of this

approach, however, is that the information that is checked into the repository is the information that everyone else is then forced to use. The implications of your changes are not always apparent to you—or perhaps immediately to your colleagues, either. Maintaining a single source of truth when hundreds of people are changing individual overlapping elements is less than straightforward. A change made by one individual can have serious consequences for many other areas of the project, and no mechanism exists for highlighting or resolving ambiguity—whoever checks the information into the repository last wins!

In summary, grand unified tools are to software engineering what grand unified theories are to modern physics—tricky to understand, multidimensional, and elusive, often involving bits of string. No one has created a single tool to maintain the full complexity of a complex IT project. Likewise, no one will do so unless the tool enables people to maintain Views in their own way, in their own language, and to identify and deal with ambiguity cooperatively.

The Connoisseur's Guide to Eating Elephants

This chapter set out to define the kinds of things the Elephant Eater must do, the kinds of problems it needs to deal with, and the kinds of environments with which it must cope. We've covered a lot of ground, so it's worth recapping the key requirements that we have established—a connoisseur's guide to eating elephants.

The Elephant Eater machine must recognize that the environment imposes many more constraints beyond functional and nonfunctional requirements. We rarely work on Greenfield sites anymore; the elephant-eating machine must be at home on the most complex Brownfield sites—the kind of Brownfield sites that have had a lot of IT complexity built up layer on layer over many years.

The Elephant Eater must also address the lack of transparency that is inherent within our most complex projects. This will enable us to x-ray our elephant to see the heart of the problem. To achieve this transparent understanding, the Elephant Eater must acknowledge the fundamental human limitation of the View and enable us to break down the problem into smaller chunks.

However, we suspect that a one-size-fits-all approach to maintaining Views is doomed to failure. A high-level business process View will always look very different than a detailed data definition. Therefore, an elephant-eating machine that relies on a single tool for all users is pretty impractical.

In addition, we now know that, despite the best efforts of architects to keep them insulated and isolated via abstractions and enterprise architectures, many of these Views are interlinked. Therefore, the only way to understand the problem properly is to make the interconnections between Views explicit and try to make them unambiguous. We should also note, however, that establishing a consolidated picture of all these Views needs to be a process of cooperation and communication—one View cannot overwrite another one, and ambiguity must be dealt with within its processing. We also know that the View should cover the entire solution (business, application, and infrastructure).

By using the formal View and VITA approach introduced in Part I, "Introducing Brownfield," it should be possible to see how the Elephant Eater proposed can address these requirements. The following facets are an intrinsic part of Brownfield development.

Our Brownfield abstractions—and, therefore, architectures—will be a good fit for the problem: Those decisions will be made based on detailed information fed in via a site survey instead of vague generalization. This adopts an engineer's approach to the solution instead of the artisan's heuristics and intuition.

We will be able to preempt the ripple effect, often understanding which requirements are in conflict or at least knowing the horrors hiding behind the constraints. Therefore, the requirements can be cost-effectively refined instead of the abstractions of the solution or its code. Resolving these problems early will have significant economic benefit.

The solution will become easier to create due to a deeper understanding of the problem. A precise and unambiguous specification will enable the use of delivery accelerators such as these:

- Global delivery and centers of excellence
- Code generation via Model Driven Development and Pattern Driven Engineering because the precise specification can be used to parameterize the generation processes
- Iterative delivery as possible strategies for appropriate business and IT segmentation of the problem become clearer

Therefore, the Brownfield approach conceptually solves many of the problems presented in this chapter and previous chapters, avoiding the early, unreliable, and imprecise abstractions and decompositions of existing approaches. In the remaining chapters, we examine how Brownfield evolved and how it can be deployed on large-scale IT projects.

Endnotes

1 Booch, Grady. "The BCS/IET Manchester Turing Lecture." Manchester, 2007. http://intranet.cs.man.ac.uk/Events_subweb/special/turing07/.

2 Splolsky, Joel. "Joel on Software." www.joelonsoftware.com/articles/LeakyAbstractions.html.

3 Maier, Mark W. and Eberhardt Rechtin. *The Art of Systems Architecting*. CRC Press, Boca Raton, Florida, 2000.

4 For example, IBM's Rational Tool Set.

Index

A

Abstract Syntax Trees (ASTs), 171
abstraction, 113, 118, 131
complexity, 122-124
complexity example, 121-122
drawbacks of, 120
ripple effect, 124-128
software engineering, 120-121
systems integration, 113-118
accelerated delivery on Brownfield sites, 156-159
Acceptance Phase, Brownfield development approach, 163
adding information, 199
agile development methods, 135
agile methods, 144-151
approach to waterfall problems, 151
versus waterfall methods, 145
Albrecht, Allan, 16
ambiguity
Elephant Eaters, consuming the environment, 41-43
Views, 29-30
Anderson, Chris, 102
Architects' Workbench, 187
architecture, 75-76
abstraction, 118, 122-124
complexity example, 121-122
ripple effect, 124-128
software engineering, 120-121
Elephant Eaters, 48-49
Artifacts, 52-55
Inventory, 50-51
Transforms, 51-52
Views, 49-50
precision architectures, merging Views, 190-194
Artifacts, 198
Elephant Eater architecture, 52
consistent configuration artifacts, 53
documentation, 53
efficient execution, 53-54
testing transforms, 54-55
generating, 198
paying your own way, 198
testing, 199-200
assumptions, 124
ASTs (Abstract Syntax Trees), 171

B

Babel Fish, 91-93
Backus, John, 170
Backus-Naur form (BNF), 170
bad news diodes, 8
BAs (business analysts), 140-141
Berners-Lee, Sir Tim, 91, 99, 135
big-mouthed superhero, 40
BNF (Backus-Naur form), 170
Boehm, Barry, 15, 144
Booch, Grady, 90, 104
bottom-up approach to systems integration, 113
BPEL (Business Process Execution Language), 189
bridging business/IT gap, 79-83
touring the model, 84-87
use cases, 79
Brook, Jr., Frederick P., 7, 25
Brooks' Law, 25

Brownfield, 14, 25, 60, 91
CASE, 138-139
death of, 105
deciding to switch from Greenfield, xxii-xxiii
evolution, 141-142
legacy code, 170-172
MDA (Model Driven Architecture), 139
business analysts (BAs), 140-141
evolution, 141-142
moving to, 204
creating Elephant Eaters, 204-205
empowering business change, 205-206
interfaces, 207
site surveys, 20-21
sources of, 134-135, 138
testing, early testing, 156
versus other techniques, 136
VITA, 166
Brownfield Beliefs, 47, 60-61
bridging business/IT gap, 64
embracing complexity, 62
establishing truth, 64
iteratively generating and refining, 63
language, 63
making business and IT indivisible, 61
reuse, 62
Brownfield development approach, 158
phases and outputs of, 159-160
subphases and outputs of, 161-162
Brownfield lifecycle, 57-59, 162
Brownfield movement, 168
Brownfield sites, accelerated delivery, 156-159
business analysts (BAs), 140-141, 206
business attractors, 104-105
business change, empowering when moving to Brownfield, 205-206
business options, software archaeology, 93-96
business process definitions, 67
Business Process Execution Language (BPEL), 189
business/IT gap
bridging, 79-81, 83
touring the model, 84-87
use cases, 79
Brownfield Beliefs, 64
language speciation, 32-34

C

CAD (Computer Aided Design), 75
CAD/CAM (Computer Aided Design/Computer Aided Manufacturing), 135
CASE (Computer Aided Software Engineering), 76, 135, 138
Brownfield, 138-139
change management, risk areas of project failure, 9
CHAOS report, xxvi
chaos theory, 105
checkpoints, quality assurance checkpoints, 144
choreography, 211
CIM (Computation Independent Model), 152
circular references, 178
class diagrams, 169
communication, 25
context, 42-47
formal versus informal, 68
gaps in, 67-72
PowerPoint, 70-72
problems
language speciation, 31-34
Views, 26-27
semantics, 42-47
syntax, 42-43

complexity
abstraction, 121-124
Brownfield Beliefs, 62
environmental complexity, 13-16, 18
effects of, 18-20
induced complexity, 9-10
Component Model, 68
Computation Independent Model (CIM), 152
Computer Aided Design (CAD), 75
Computer Aided Design/Computer Aided Manufacturing (CAD/CAM), 135
Computer Aided Software Engineering (CASE), 76, 135, 138
Configuration Artifacts, 198
considerations for Elephant Eaters
conflicting goals, 111-112
interactions, 112
transparency, 110-111
consistent configuration artifacts, 53
constraints, 14-15, 49-50
consuming the environment, Elephant Eaters, 41
overcoming inconsistency and ambiguity, 41-47
context, 42
Elephant Eaters, consuming the environment, 43-47
overcoming inconsistency and ambiguity, 43-47
customers, 103-104

D

Data Definition Language (DDL), 166
data sources for forming Inventories, 168
DDL (Data Definition Language), 166
decomposition of complex problem space, 118
DeMarco, Tom, 30
developing tools, 207
diagrammatic views, 169
diagrams, 67-68, 71
documentation, 53
Documentation Artifacts, 198
Domain Object Model (DOM), 149
dynamic aspects, considerations for Elephant Eaters, 112
dynamic services, 100-103

E

Eclipse, 171
Eclipse Modeling Framework (EMF), 187
Elephant Eaters
in action, 55-57
generating and refining, 59-60
architecture, 48-49
Artifacts, 52-55
Inventory, 50-51
Transforms, 51-52
Views, 49-50
Brownfield Beliefs, 60-61
bridging business/IT gap, 64
embracing complexity, 62
establishing truth, 64
iteratively generating and refining, 63
language, 63
making business and IT indivisible, 61
reuse, 62
considerations for
conflicting goals, 111-112
interactions, 112
transparency, 110-111
consuming the environment, 41
overcoming inconsistency and ambiguity, 41-47
creating, 204-205

environment, 130
portrait of, 200-201
elephant-eating strategies, 39-41
EMF (Eclipse Modeling Framework), 187
Engineer Phase, Brownfield development approach, 162
enterprise architectures, evolving, 212-213
Enterprise Service Buses (ESBs), 209-211
environment, 129
environmental complexity, 13-18
effects of, 18-19
ripple effect, 18-20
ESBs (Enterprise Service Buses), 211
building, 209-211
evolution, Brownfield, 141-142
Executable Artifacts, 198
execution, artifacts, 53-54
exploring Inventory manually, 73-75
extracting information, merging Views, 189-190
extracts, 198

F

function point analysis, 16
functional requirements, 11

G

gaps in communication, 67-72
generating Artifacts, 198
paying your own way, 199
generation faults, identifying, 200
Gerstner, Lou, 30
globalization, 6
goals, conflicting goals, 111-112
Greenfield, deciding to move to Brownfield, xxii-xxiii

H

Haasjes, Geert-Willem, 100
Hilbert space, 72-73
Inventory, exploring manually, 73-75

I

IBM
Inventory structures, 179
patents for implementation of Brownfield, 205
System/360, xxi
IBM islands, 84
IBM Rational Software Architect (RSA), 189
identifying
generation faults, 200
missing or incorrect information, 186-187
patterns, 169-170
IFPUG method, 16
inconsistency
Elephant Eaters, consuming the environment, 41-47
Views, 27-28
induced complexity, risk areas of project failure, 9-10
inference, 190
innovation capacity, IT spending, 18
interactions, considerations of Elephant Eaters, 112
interfaces
building, 207-209
moving to Brownfield, 207
Inventory
data sources, 168
Elephant Eater architecture, 50-51
exploring manually, 73-75
importers, 57
OWL, 180, 183

structure of, 173
triples, 173-180
Inventory optimizers, 105
IT, 79
IT spending, 18
iterative development, 93

J-K

JAD (Joint Application Design/Development), 135

L

language, Brownfield Beliefs, 63
language speciation, 31-32
business/IT gap, 32-33
making business and IT indivisible, 34
languages
choosing, 205
ontologies, 98-99
OWL. *See* OWL
legacy code, 170-172
lifecycles, Brownfield lifecycle, 57-59
lifetimes, 73
Lister, Timothy, 30
Logical Data Model, 68
Lojek, Bob, 124
long tail, 102

M

mashups, 134
MDA (Model Driven Architecture), 152-153
Brownfield, 139
BAs (business analysts), 140-141
evolution, 141-142
Pattern Driven Engineering, 153-154
reversing, 155-156
MDA/MDD (Model Driven Architecture/Model Driven Development), 135
MDD (Model Driven Development), 64
memory techniques, 67
merging Views, 183, 185
extracting information, 189-190
identifying missing or incorrect information, 186-187
precision architectures, 190-194
time dimensions, 187-189
transforms, 195-197
metadata, 87
metaphors, 84
Microsoft PowerPoint, 70-72
middleware, 212
Model Driven Architecture (MDA), 151-153
Brownfield, 139
business analysts (BAs), 140-141
evolution, 141-142
Model Driven Architecture/Model Driven Development (MDA/MDD), 135
Model Driven Development, 57, 62
models, bridging business/IT gap, 84-87
mosaic language zones, 32
moving to Brownfield, 204
creating Elephant Eaters, 204-205
empowering business change, 205-206
interfaces, 207
***Mythical Man Month, The*, 25**

N

Naur, Peter, 170
nonfunctional requirements, 11

O

Occam's Razor, 10
OMG (Open Management Group), 152
On Demand, 95
ontologies, 98-99

Open Management Group (OMG), 152
organization, risk areas of project failure, 7
outputs of Brownfield development approach, 159-162
OWL (Web Ontology Language), 138, 180
  Inventory, 180, 183

P

Palmisano, Sam, 95
parable of the blind men and the elephant, 24
parochialism, 47
  Views, 30-31
parsing Views, 169-170
Pattern Driven Engineering, MDA (Model Driven Architecture), 153-154
patterns, 125
  identifying, 169-170
Peopleware, 30
phases of Brownfield development approach, 159-160
physically separated teams, 30
PIM (Platform Independent Model), 152
planning, risk areas of project failure, 7
Platform Independent Model (PIM), 152
Platform Specific Model (PSM), 152
plumbing, comparison, 121
PowerPoint, 70-72
precision architectures, merging views, 190-194
presentations, 71-72
private languages, 32
process flows, 67-68
project reporting, risk areas of project failure, 7-8
PSM (Platform Specific Model), 152

Q

quality assurance checkpoints, 144

R

RAD (Rapid Application Development), 135
RDF (Resource Description Framework), 138, 180
RDF graphs, 183
RDF/XML extract, 180
regulatory compliance, IT spending, 18
reporting on projects, risk areas of project failure, 7-8
representing triples, 72
requirements
  conflicting requirements, 111
  functional requirements, 11
  nonfunctional requirements, 11
  risk areas of project failure, 11-13
Resource Description Framework (RDF), 180
reuse, Brownfield Beliefs, 62
reversing MDA (Model Driven Architecture), 155-156
ripple effect, 18, 131
  abstraction, 124-128
  effects of environmental complexity, 19-20
risk areas of project failure
  change management, 9
  globalization, 7
  induced complexity, 9-10
  organization and planning, 7
  project reporting, 7-8
  requirements, 11-13
RSA (Rational Software Architect), 189
rules, 93

S

Sarbanes-Oxley, 18
Second Life, 77
semantic technologies, 90
semantic web, 99-100, 135, 183
semantics, 42
  Elephant Eaters, consuming the environment, 43-47
  overcoming inconsistency and ambiguity, 43-47
Service Oriented Architecture (SOA), 33
services
  customers, 103-104
  dynamic services, 100-103
singing pigs, 70
site surveys, 14, 110, 141, 168
  Brownfield sites, 20-21
skills, developing for Elephant Eaters, 204
SOA (Service Oriented Architecture), 33
software archaeology, 91-93
  business options, 93-96
  structures, 96-97
software engineering, abstraction, 120-121
sources of Brownfield, 134-135, 138
splitting, 113
  systems integration, 113-118
Standish Group, xxvi
static testing, 163
steady state, IT spending, 18
Stock, Gregory, 105
stove-pipe systems, 16
strange attractors, 104
strategies for elephant-eating, 39-41
structure of Inventory, 173
structures, software archaeology, 96-97
subphases of Brownfield development approach, 161-162
Survey Phase, Brownfield development approach, 162
syntax, 42
  overcoming inconsistency and ambiguity, 42-43
System/360 (IBM), xxi, 16
systems integration
  bottom-up approach, 113
  splitting or abstraction, 113-118
  top-down approach, 116

T

TADDM (Tivoli Application Dependency Discovery Manager), 97, 168
taxonomy, 114
teams, physically separating, 30
Test Artifacts, 198
testing
  Artifacts, 199-200
  Brownfield, early testing, 156
  static testing, 163
  transforms, artifacts, 54-55
text messages, 32
The Open Group Architecture Framework (TOGAF), 114-115
three-dimensional displays, 76-78
time dimensions, merging Views, 187-189
time slices, 73
Tivoli Application Dependency Discovery Manager (TADDM), 97, 168
TOGAF (The Open Group Architecture Framework), 114-115
tool unification, 128-129
tools, developing, 207
top-down approach, systems integration, 116
transformations, 195

Transforms
creating, 197-198
Artifacts, 198
Elephant Eater architecture, 51-52
merging Views, 195-197
testing, 54-55
transparency, lack of, 110-111
triples, 51
Inventory, 173-180
representing, 72
truth, Brownfield Beliefs, 64
Turner, Richard, 144

U

UML (Unified Modeling Language), 43, 139
UML diagrams, 169
unified tools, 128-129
updating information, 200
use cases, 66, 79

V

Venn diagrams, 99
Views, 26, 166
communication, 26-27
ambiguity, 29-30
inconsistency, 27-28
parochialism, 30-31
Elephant Eater architecture, 49-50
merging, 183, 185
extracting information, 189-190
identifying missing or incorrect information, 186-187
precision architectures, 190-194
time dimensions, 187, 189
transforms, 195-197
one-size-fits-all approach, 130
parsing, 169-170
software archaeology, 91
splitting, 114
virtual worlds (v-worlds), 77, 135
Second Life, 77
visualizations, 87
VITA (Views, Inventory, Transforms, and Artifacts), 48, 166-167

W

W3C (World Wide Web Consortium), 179-183
waterfall development, 144
waterfall methods, 146-151
agile approach to problems, 151
versus agile methods, 145
Web 2.0, 91
Web Ontology Language. *See* OWL
WebSphere Business Modeler, 187, 204
World Wide Web Consortium (W3C), 180, 183

X-Y

XML, 91

Z

Zachman Framework, 115